12-16-2020

Methods for Improving Time and Accuracy in Deep Learning and Its Applications

Xueli Xiao

METHODS FOR IMPROVING TIME AND ACCURACY IN DEEP LEARNING AND ITS APPLICATIONS

by

XUELI XIAO

Under the Direction of Yi Pan, Ph.D.

ABSTRACT

Deep learning has achieved great performance in various areas, such as computer vision, natural language processing, and speech recognition. In this research, we design methods to improve the prediction performance and decrease training time of deep learning models. We first propose an efficient evolutionary algorithm (EA) to automatically tune hyperparameters in a deep learning model in Chapter 2. We use a variable length genetic algorithm (GA) to systematically and automatically tune the hyperparameters of a Convolutional Neural Network (CNN) to improve its performance. Experiment results show that our algorithm can find good CNN model hyperparameters efficiently. In Chapter 3, we propose a method to intelligently freeze layers during the training process to decrease training time. Our method involves designing a formula to calculate normalized gradient differences for all layers with weights in the model and then use the calculated values to decide how many layers should be frozen. We implemented our method on top of stochastic gradient descent (SGD) and performed experiments on standard image classification dataset CIFAR-10. Results show that our method can accelerate training on VGG nets, ResNets, and DenseNets while having similar test accuracy. Next, in Chapter 4, we propose to incorporate prior knowledge into the training process to improve classification accuracy. We incorporate class similarity knowledge into CNN models using a graph convolution layer. We evaluate our method on two benchmark image datasets: MNIST and CIFAR-10 and analyze the results on different data and model sizes. Experimental results show that our model can improve classification accuracy, especially when the amount of available data is small. In Chapter 5, we map Electronic Health Records (EHRs) to images and feed them to CNNs for feature relationship learning. The relationships between EHR features are quantitatively measured before mapping to images. We add this relationship-learning part as a boosting module on the original machine learning model. Experimental results show that our proposed models have better performance compared with the baseline models. In summary, this research proposes various methods to improve the training time and performance of deep learning models.

INDEX WORDS: Deep Learning, Convolutional Neural Networks, Hyperparameter Optimization, Training Speed, Prior Knowledge, Electronic Health Records

METHODS FOR IMPROVING TIME AND ACCURACY IN DEEP LEARNING AND ITS APPLICATIONS

by

XUELI XIAO

A Dissertation Submitted in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

in the College of Arts and Sciences

Georgia State University

2020

METHODS FOR IMPROVING TIME AND ACCURACY IN DEEP LEARNING AND ITS APPLICATIONS

by

XUELI XIAO

DEDICATION

To my beloved family.

ACKNOWLEDGEMENTS

This research would not have been possible without the support of many people. I would love to express my deepest gratitude to my advisor, Dr. Yi Pan, for his continuous support and guidance through this journey. Dr. Pan has been very encouraging, and his innovative thinkings and visions are especially inspirational. I share my gratitude to all the dissertation committee members, Dr. Yanqing Zhang, Dr. Rolando Estrada, and Dr. Yichuan Zhao, for their advice, comments that significantly improved the research. I thank all colleagues from my research group who provided insight and expertise that greatly assisted the research. I thank all professors and staff at Georgia State University that supported me in various ways. I take this opportunity to thank Dr. Raj Sunderraman, who gave me support when I transitioned to Computer Science along with other valuable advice. I express great gratitude to Dr. K. N. King, who gave me important suggestions after I obtained my master's and shared valuable experiences for teaching. I thank Tammie, the best academic administration specialist I have ever met, for always being so patient and available for students.

I would also like to show my gratitude to my beloved family and all my friends for their support and understanding. Special thanks to my parents, who always believe in me, and my husband Shangguo Zhu for his company and support.

This research is supported by the Department of Computer Science and the Molecular Basis of Diseases (MBD) research program at Georgia State University. This research uses computational resources from various institutions. I acknowledge the high performance computing resources at Georgia State University (https://ursa.research.gsu.edu/high-performance-computing) for providing GPU resources. I thank NVIDIA Academic Hardware Grant for the donation of the Tesla K40 GPU. I thank the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number ACI-1548562. Specifically, I used the Bridges system, which is supported by NSF award number ACI-1445606, at the Pittsburgh Supercomputing Center (PSC).

TABLE OF CONTENTS

ACKNOWLEDGEMENTS . . . . **v**

LIST OF TABLES . . . . **x**

LIST OF FIGURES . . . . **xii**

PART 1 INTRODUCTION **1**

PART 2 EFFICIENT EVOLUTION OF HYPERPARAMETERS **6**

2.1 Background 6

2.1.1 Convolutional Neural Networks 6

2.1.2 Genetic Algorithms 7

2.1.3 Hyperparameter Optimization Algorithms 8

2.2 Method 10

2.2.1 Variable Length Genetic Algorithms 11

2.2.2 Encoding Scheme 13

2.2.3 Fitness of individuals 15

2.3 Experiments and Results 16

2.3.1 Datasets 16

2.3.2 Search Space 17

2.3.3 Experiment Setup 19

2.3.4 Genetic Algorithm Settings 20

2.3.5 Stopping Condition 20

2.3.6 Results 21

2.4 Summary and Future Work 24

PART 3 FAST DEEP LEARNING TRAINING THROUGH INTELLIGENTLY FREEZING LAYERS . . . 28

3.1 Background . . . 28

3.1.1 Deep Learning Models . . . 29

3.1.2 Optimizing Deep Neural Networks . . . 30

3.1.3 Time Reduction Methods . . . 30

3.1.4 Layer-wise and Parameter-wise Learning Rate . . . 32

3.1.5 Parameters' Participation in Training . . . 33

3.1.6 Gradient Information Used . . . 33

3.2 Method . . . 34

3.2.1 Layer Level Freezing . . . 34

3.2.2 Layer Freezing Rate . . . 35

3.2.3 Layers to Freeze . . . 39

3.2.4 Overhead . . . 41

3.2.5 Freezing Scheme . . . 41

3.3 Experiments and Results . . . 43

3.3.1 VGG Models . . . 43

3.3.2 ResNets . . . 44

3.3.3 DenseNets . . . 45

3.4 Summary and Future Work . . . 45

PART 4 INCORPORATING PRIOR KNOWLEDGE IN DEEP LEARNING . . . 48

4.1 Background . . . 48

4.1.1 Classification Accuracy Improvement Methods . . . 48

4.1.2 Knowledge incorporation in deep learning . . . 49

4.1.3 Class Similarity . . . 50

4.1.4 Graph Convolutional Networks . . . 50

4.2 Preliminary Studies . . . 51

4.2.1 Fourier Descriptor 52

4.2.2 Matching with Templates 52

4.2.3 Incorporate knowledge into Training 53

4.3 Preliminary Results 57

4.3.1 Exclude classes deterministically 57

4.3.2 Exclude classes stochastically 57

4.4 Incorporate Class Similarity Knowledge 58

4.4.1 Represent Class Similarities Using Misclassification Graph 59

4.4.2 Overall Model Architecture 59

4.5 Experiments and Results 62

4.5.1 Datasets 63

4.5.2 Baseline 63

4.5.3 Results on MNIST Dataset 63

4.5.4 Results on CIFAR-10 Dataset 64

4.6 Summary and Future Work 64

PART 5 MAPPING NON-IMAGE DATA TO IMAGES FOR FEATURE RELATIONSHIP LEARNING **73**

5.1 Backrgound 73

5.2 Method 74

5.2.1 Mapping Electronic Health Records to images 74

5.2.2 Data Augmentation 80

5.2.3 Convolutional Neural Networks for image mapping classification . 81

5.2.4 Overall Model Architecture 83

5.3 Experiments and Results 83

5.3.1 Chronic Lymphocytic Leukemia Dataset 83

5.3.2 Disease progression prediction on CLL dataset 85

5.4 Summary 86

PART 6 CONCLUSIONS AND FUTURE WORKS **88**

6.1 Conclusions 88

6.2 Future Works 89

6.2.1 Extend Layer Freezing to Freeze Arbitrary Layers 89

6.2.2 Use More Metrics for Model Evaluations 89

6.2.3 Combining Improvements from Various Methods 89

6.2.4 Use Prior Knowledge to Generate Soft Targets 90

6.2.5 Use Graph Attention Networks to Learn Class Similarities 90

LIST OF TABLES

Table 2.1 The default search space used by our variable length GA. 21

Table 2.2 Accuracy Comparison of Discovered models when using similar time (less than 30 hours). 25

Table 3.1 The mean running time and test accuracy for VGG, ResNet, and DenseNet architectures on CIFAR-10. Comparison is between the SGD and our freezing method on top of SGD. The mean and standard deviation is calculated based on 3 runs. VGG-19 Experiments were done in a controlled environment on a single NVIDIA Tesla K40 GPU. ResNet-18 and Resnet-50 experiments were run on a single NVIDIA Tesla V100 GPU on a HPC cluster. ResNet-101 experiments were run on a single NVIDIA Tesla P100 GPU on a HPC cluster. 44

Table 3.2 We reduce the freezing frequency in the second learning state (the learning rate is 0.01) due to the observations made. Experiments were run three times for both the freezing SGD and vanilla SGD. Mean and standard deviation for time and test accuracy is calculated. Results show that the SGD with freezing now have slightly higher accuracy. 45

Table 4.1 We evaluate the templates quality using the percentage of images for which the target class is within the included classes. 56

Table 4.2 Comparison between basic softmax and deterministic exclusion. . 57

Table 4.3 Test accuracy of stochastic exclusion with various scale factors . . 58

Table 4.4 Accuracy comparison on MNIST of the original CNN model and PK-GCN models with various data sizes. ‘X’ Means no accuracy improvement is observed. 65

Table 4.5 Test accuracy comparison on CIFAR-10 of the original CNN model and PK-GCN models with various data sizes. 66

Table 5.1 Precision comparison on the CLL dataset with different K values. The CNN boosted Model outperforms the baseline models. 87

LIST OF FIGURES

Figure 2.1 The hyperparameter optimization steps using our variable length GA. 13

Figure 2.2 The initial chromosome (Phase 0). In the chromosome, what type of activation function to be used for the whole model is encoded. It also includes hyperparameter information for the first two convolutional layers: the size of the convolutional windows, number of output channels, and whether or not to include batch normalization for the layer. For the whole two-layer block, there are three additional hyperparameters: whether to add a pooling layer after the first convolutional layer, what is the pooling type, and if a skip connection should be included. 14

Figure 2.3 An example of a skip connection. This block contains two convolutional layers: Layer a and Layer b. The skip connection performs a 1x1 convolution on the input, and its output is added with the output of Layer b. Stride size and number of filters of the skip connection is setup such that its output tensor has the same shape as that of Layer b. 15

Figure 2.4 Encoding for Phases after Phase 0. Chromosome in the current phase is an extension of the one from the previous phase plus some new fields. Chromosomes are longer, and the represented models have more layers. The field with gray background indicate the chromosome from the previous phase. The rest defines the properties of new layers in the current phase. In the current, two convolutional layers can be added, with the flexibility of letting the algorithm to decide whether to add one or two. This is achieved by adding a “Include Layer b?” field, the one with yellow background, in the chromosome. 16

Figure 2.5 The model grows deeper when a new phase begins. The models in the new phase are built based on the best model in the previous phase. To the left is an example of the best model found in Phase 0, and to the right is a model in Phase 1. 17

Figure 2.6 The corresponding chromosomes for models in Figure 2.5. Chromosomes grows longer when a new phase is entered. The chromosome in the new phase is extended from the best chromsome in the previous phase. 18

Figure 2.7 Weight initialization scheme for deeper models. A deeper model of the current phase is built on top of the best model from the previous phase. Part of its weights is transferred from the previous best model, part is initialized randomly. 19

Figure 2.8 The size of the search space grows exponentially as the number of layers in the deep learning model increases. In the initial phase of our algorithm, models have two layers and the search space is 156,800. As the model goes deeper, this number grows exponentially. When the number of layers reaches 14, the search space grows to about 10^{35}. 20

Figure 2.9 The evolutionary process stops when the stopping condition is met. In this example the algorithm stops at Phase 14 because the fitness drop is larger than the threshold. The best individual, in this case at Phase 13, is picked to receive further training. 22

Figure 2.10 The fitness of 20 individuals evolved for 5 generations in each phase. The algorithm stops at Phase 14 when the stopping condition is met. The best individual found in this evolutionary process is chosen to receive more training. The overall evolution time is less than 25 hours. 23

Figure 2.11 We implemented classical GA and ran the experiment with various population sizes and 30-hour time constraint. We sort the fitness of individuals in each generation and plot their fitness. (a)(b)(c) are plots for 50, 20, and 10 population respectively. 26

Figure 2.12 We implemented large scale evolution on top of a public repository, and ran the experiment with different population setting for 30 hours. (a)(b)(c) are plots for 30, 50, and 80 population respectively. The fitness of the best discovered individuals are slightly above 0.5 when time is restricted to 30 hours on 1 Nvidia Tesla V100 GPU. . . . 27

Figure 3.1 Mean gradient magnitudes of ResNet-18 for different layers. The model is trained for 350 epochs with step learning rate decay. The plot shows how the gradient changes as the training progresses. 35

Figure 3.2 Freezing rates for all layers in ResNet-18. The model is trained for 350 epochs with step learning rate decay. The plot describes how the freezing rates change as the training goes on. We are going to use this information in our freezing process. To better see the freezing rates for the first 17 layers, refer to Figure 3.3. 37

Figure 3.3 To better illustrate the freezing rate values for the first 17 layers, we remove the last layer in ResNet-18 in this plot, so that more details can be seen. 38

Figure 3.4 For the purpose of time saving, frozen layers should always start at the first layer and be consecutive. If there is an input to a frozen layer that is unfrozen, gradient calculation is still required for the frozen layer. For example in (a), Layer 1, 2, 3 are frozen and the rest are unfrozen. The frozen layers do not have inputs that requires gradients. Thus we do not need to calculate gradients for Layer 1, 2, 3. In (b), Layer 1, 3 and Layer n-1 are frozen. However, since Layer 2 is unfrozen, we still need to calculate gradients for Layer 3 in order to know how to update weights in Layer 2. Similarly, gradient calculation is required for Layer n-1. In this case, there will not be much time saving. . . 40

Figure 3.5 We train ResNet-50 using SGD with freezing and vanilla SGD. The test accuracy for SGD with freezing outperforms or is similar to vanilla SGD in the first two learning stages (when the learning rate is 0.1 and 0.01). However, the accuracy is slightly lower when it comes to the last stage (the learning rate is 0.001). This suggests that there might be too much freezing in the second learning stage. 46

Figure 4.1 Fourier descriptor distance between the candidate image and the templates. 54

Figure 4.2 The original softmax versus softmax with knowledge mask. 67

Figure 4.3 Excluding classes deterministically using k=5. 68

Figure 4.4 Class exclusion probabilities. 68

Figure 4.5 Classes are excluded stochastically based on exclusion probabilities. 69

Figure 4.6 A Misclassification Graph example on the MNIST Dataset. 69

Figure 4.7 Overall model architecture. In Stage 1 training, mislassification graph is obtained from the the validation data. The graph represents class similarity information and is fed into the next stage. In Stage 2 training, a graph convolution layer is added to incorporate class similarity knowledge into the training. 70

Figure 4.8 The GCN takes in the data embedding concatenated with class embeddings, performs neighborhood aggregation, and outputs new node representations. 70
Figure 4.9 Obtain data embedding from the model. 71
Figure 4.10 Obtain class embeddings for all classes from the model. 71
Figure 4.11 Two ways of incorporating the graph convolution layer. 72
Figure 5.1 An example of how related features are positioned as neighboring pixels inside of an image. 76
Figure 5.2 The genetic representation of feature positions inside of an image is the unrolling of the image. 77
Figure 5.3 Modified crossover operation. 78
Figure 5.4 Mutation operations swaps two values. 78
Figure 5.5 A 3x3 neighborhood example. 79
Figure 5.6 To generate a synthesize positive patient image, a positive patient's feature is overlaid with part of a negative patient's features. . . . 82
Figure 5.7 The overall architecture of the proposed model. 84
Figure 5.8 Actual examples of mapping CLL diagnosis and treatments to images. 85
Figure 5.9 Overall model performance. 86

PART 1

INTRODUCTION

Deep learning has achieved great performance in various areas, such as computer vision [28][99], natural language processing [33], speech recognition [32], and Covid-19 data analysis [44][104][109]. Neural network models are constructed and trained to map the data to the corresponding solutions. Researchers have been working on various methods to improve the accuracy of deep learning models. However, improving accuracies is not an easy task: it takes expert knowledge and trials and errors. One way of improving model performance without human experience is by using automatic hyperparameter optimization. Another problem with deep learning models is that the training time is very long. In this research, we focus on methods to improve deep learning model accuracies and decrease training time.

CNNs have achieved great performance in the area of computer vision. Designing the architecture of CNNs so that it produces good results on a task is challenging. It involves decisions on various hyperparameters of the network, such as how many layers are in the network, how many filters are in a layer, what are the sizes of convolutional windows, and so on. All the different combinations of the hyperparameters create a huge number of possibilities for CNN models. Traditionally, the selection of hyperparameters is done by hand tuning, i.e., manually testing different sets of parameters that may work well. This requires a certain level of knowledge and expertise in deep learning, and the process of trial and error is tedious, time-consuming, and maybe discouraging to the novices. It is thus beneficial to automatically search for a CNN architecture that suits the needs of the task. Searching for a good deep learning model is challenging, very computationally intensive, and time consuming. The lengthy search time is caused by many factors: the evaluation time for a single deep learning model is long, the search space grows exponentially large as the model depth increases, and the access to GPU resources is limited – many researchers only have access to one or a few GPUs. Given the above mentioned reasons, it is certainly not practical to traverse every single possibility to find the best model. Thus we use an efficient EA to automatically and systematically tune hyperparameters to find models that give good accuracy.

Deep learning models are often optimized using SGD [62] or other optimization algorithms. One drawback of these algorithms is that the training time is very long. Depending on the task and available hardware, this time could be hours or even weeks. Even with the availability of high-performance computing systems, some deep neural networks require several days for training. For example, it takes six days to train AlexNet [48] on two GTX 580 3GB GPUs. As a solution for this, SGD is usually implemented with learning rate decay. A higher learning rate helps speed up training but may miss the optima or fail to converge. A lower learning rate is more reliable but makes the training speed suffer. Usually, in the earlier epochs of training, higher learning rates are used to make training faster. Then as the training goes on and the model weights get closer to the optimum, the learning rate decreases to help with convergence. There are many kinds of research on training time saving, and in our work, we intelligently freeze layers to save training time.

In multi-class image classification, image features are extracted and learned through convolutional kernels and eventually mapped to one class among some other classes. Typically during this process, CNNs learn from the information contained within the images only, and no prior knowledge is considered into the training. We first do a preliminary study that excludes classes that are not similar to the target class during training. Then we incorporate class similarity knowledge into deep learning training by using a graph convolutional layer. The performance of classifications can greatly be impacted by the similarity between classes. For example, a cat would have a larger chance of being misidentified as a dog, than as an airplane. By learning the similarity among classes and incorporating them in our models, we could potentially improve classification performances. We propose to incorporate class similarity knowledge into deep learning models to improve multi-class classification performance. Specifically, we experiment on image classification using CNNs. Our method is not task specific, and has the potential of being applied to the classification on other types of data besides images. There are some prior works that take class similarity into consideration. Lee et al. proposed Dropmax [51], a method which randomly drops classes adaptively, to improve the accuracy of deep learning models. During the class dropping, classes that are more similar to the input data have larger chances of being kept. Chen et al. [13] used word semantic similarity to calculate the underlying structures between labels and used a graph convolutional network (GCN) augmented classifier to do classification. Their method needs external information to define a label graph and initiate node representations. Using external information can have issues. For example, in many cases, labels' word semantic similarity cannot reflect the real similarity of the data. And to get the embeddings for classes, we need to rely on other machine learning models. Inspired by Chen et al.'s method [13], we propose our method that does not rely on any external information and can be applied to much wider ranges of classification problems. A misclassification graph derived from the validation data set can be used for class similarity, and class embeddings can be extracted from model weights. The goal of our work is not to beat the state-of-arts on the benchmark image classification dataset. Instead, we use the datasets to evaluate the effect of our method on various data and model sizes and analyze under which scenarios does our method work the best.

My research focuses on improving the accuracy and training time of deep learning models, especially in four categories: using automatic hyperparameter optimization to improve model accuracies, freeze layers to increase training time, incorporate prior knowledge in the training process to improve model performance, and mapping non-image data to images for local feature similarity learning. The contributions of my research are as follows:

- Our hyperparameter optimization method can efficiently discover deep learning models with variable depths. It makes hyperparameter optimization more feasible for researchers with limited computing resources.
- Our algorithm does not have a constraint on the depth of deep learning models, which means our method can handle problems of different sizes.

- We incorporated crossover operation in our model evolution process along with efficient model evaluation techniques to make the model discovering process more effective.
- Our intelligent freezing method opens a new way of accelerating the training of deep learning models while having similar model accuracy. With further exploration and experiments on this method, there is more room for improvement.
- A formula is designed to calculate freezing rates for layers in the model. Then the maximum subarray sum is calculated on the array of freezing rates to decide how many layers from the beginning layer should be frozen. This paves the way for the future design of intelligent freezing.
- In order to reduce the computation time overhead, we selectively and periodically calculate the gradient information during the training process.
- Our prior knowledge incorporation method adds class similarity knowledge into deep learning models to improve classification accuracy.
- We use a novel way of defining class similarities using misclassification graphs on the validation dataset. In our definition, similarities have directions.
- We propose a novel two-stage training model. The first stage training obtains class similarity knowledge, and the second training stage combines the original model's output with the convoluted class similarity graph outputs and improves classification performance.
- Our method does not require extra external information. Class similarity knowledge is extracted through the dataset itself. Class and data embeddings are both obtained from trained network weights.
- We analyze the effect of adding class similarity knowledge with different model and dataset sizes.

- We use a CNN to learn the relationship among medical entities in EHR data and perform a supervised prediction task.
- We propose a novel method that maps EHR patient journeys (diagnosis, procedures, and prescribed drugs) to images. The underlying structure: relations among the medical entities are taken into consideration.
- Deep learning models require balances among classes. In the actual data, the number of actual positives is small. We design a novel EHR data augmentation method that upsamples the positive class.
- We utilize a hybrid machine learning model architecture that includes a base machine learning model and a CNN boosting model. The base machine model can learn additional features that are not mapped to images.

The details of my research progress are described in the following chapters ,and the future works are listed at the end of this research.

PART 2

EFFICIENT EVOLUTION OF HYPERPARAMETERS

2.1 Background

There are various methods for hyperparameter optimization. We focus on methods based on EAs. EAs have been used in hyperparameter optimization of deep learning models. Young et al. [97] used a GA to optimize the hyperparameter of a 3-layer CNN. The algorithm is not suitable for situations where we do not know how many layers are needed there. Real et al. [61] used a mutation only EA and gradually grows the deep learning model to find a good set of combinations. The evolutionary process is slow due to the mutation only nature.

Inspired by the work of Young et al. [97] and Real et al. [61], we propose a variable length GA to efficiently optimize the hyperparameters in CNNs. Models of various hyperparameters settings are first created and evolved using the algorithm. We do not put a constraint on the model depth, and various techniques such as crossover operation in the evolving process make the algorithm more efficient. Experiment results show that our method can efficiently find good hyperparameter combinations even when the search space becomes exponentially large. Hyperparameter optimization is itself a time-consuming process. Having an efficient algorithm is especially helpful when the access to GPU hardware is limited, and the search space is very large.

2.1.1 Convolutional Neural Networks

A CNN is a type of deep artificial neural network, and it is assumed to be used for images. Classical CNNs contain convolutional layers, pooling layers, ReLU activations, and fully connected layers. Convolutional layers have convolutional windows that can extract features from images. These windows are usually called filters or kernels. The kernels move through the images, performing convolutions on local image data and produce feature maps.

Pooling is used to reduce the number of parameters in a CNN. It is also called downsampling or subsampling. It can help reduce computation complexity. If a feature map is of size 2020, using pooling of size 22 and step of size 2, the dimension can be reduced to 1010. There are different types of pooling layers: max pooling, average pooling, fractional max pooling, and so on. Max pooling takes the pixel of the biggest value in the pooling window, whereas average pooling calculates the average of all the pixels in the window. These are the two commonly used pooling methods in CNNs. Activation function ReLU stands for Rectified Linear Unit. It introduces nonlinearity into the network. ReLU activation keeps all positive input and makes negative input to be zero: f(x)=Max(0,x). Some of the other types of activation functions include tanh, Exponential Linear Unit (ELU) [19], Scaled Exponential Linear Unit (SELU) [47], and so on. A fully connected layer has full connections to neurons in the previous layer.

2.1.2 Genetic Algorithms

GAs are biologically inspired. It learns from Darwin's evolutionary theory. In Darwin's theory of natural selection, the fittest individuals are selected, and they produce offsprings. The characteristics of these individuals are passed to the next generations. If the parents have higher fitness, their offsprings tend to be fitter and have more chances of survival.

GAs learn from this idea. They can be used to solve optimization and search problems. In GAs, candidate solutions are evolved to generate better ones. A set of solutions form a search space, and the goal is to find the best ones among them. This is similar to finding the fittest individual in a population. GAs start with a population that contains random solutions within the search space.

Each solution has a chromosome, which contains properties of the solution. These chromosomes can be altered. A typical GA has three bio-inspired operators that can be used on a chromosome: selection, crossover, and mutation. Selection means to select a portion of the population as candidates to produce offsprings and generate more solutions. Usually, the fitter individuals are selected. The fitness of a solution can be calculated using a fitness function, and it reflects how good the solution is. The crossover combines the chromosomes of two parents and produces a chromosome for the offspring. The offspring's chromosome inherits the properties of both parents. The Mutation operator is like a biological mutation: it changes one or more values in the chromosome. This introduces more diversity in the population.

GAs are simple yet effective. They have been applied on various research problems such as vehicle routing problem [3], neuroscience search problems [88][89], deep learning hyperparameter optimizations [61][97], neural network weight optimizations [26][58], dynamic channel assignment [25], and so on. GAs with variable chromosome length have also been applied to many problems [60][71].

2.1.3 Hyperparameter Optimization Algorithms

Grid search and random search [9] are two popular methods for hyperparameter optimization. Grid search selects the best model among many models built on predefined hyperparameter settings. It evaluates all the models to guarantee the best one is found. As the number of hyperparameters grow and search space gets increasingly larger, the amount of time it takes for grid search grows exponentially, making it impractical to be used. Random search [9] takes less time than grid search because unlike grid search, it does not search for all possibilities exhaustively. Instead of trying out all possibilities, it randomly selects models with different hyperparameter combinations. There is a trade-off between search time and the quality of discovered models. It is not guaranteed that the optimal hyperparameter combination within the search space will be found. It turns out random search can achieve similar results compared with grid search while being more efficient.

Besides grid search and random search, there are other approaches, such as Bayesian optimization, gradient-based methods, reinforcement learning based methods, and EAs. Bayesian optimization methods [65][70][76] use an acquisition function to smartly decide where the algorithm should explore next in the search space. A probabilistic model is built to balance exploration and exploitation. Gradient-based methods can be used if the hyperparameters are continuous. Luketina et al. [54] and Fu et al. [24] use gradient descent to optimize hyperparameters. Reinforcement learning based methods are used to automatically discover CNN architectures. Zoph et al. [111] used a Recurrent Neural Network to generate CNN model structures and applied reinforcement learning to improve the generated architecture. Baker et al. [4] used a Q-learning agent to discover a network layer by layer.

Another large category of hyperparameter optimization method is based on EAs. Research work under this category differs in their EAs used, the search space, the encoding scheme for CNN model architecture, ways of reducing computational cost, and so on. Young et al. [97] performed GAs to optimize hyperparameters in CNNs. A three-layer fixed-architecture CNN is used, and only six hyperparameters are tuned. The problem with a fixed layer architecture is that the models may be too small to fit the problem, leading to a high bias. Real et al. [61] evolved complex CNN architectures from very simple individuals by applying different kinds of mutation operations. Model structures are encoded as graphs in [61]. While the method by Real et al. [61] can find accurate models for challenging tasks, the computational cost is very high: 250 workers are used, and the searching time is over 250 hours. A mutation only algorithm can contribute to high computational cost since the evolutionary process is very slow. Many hyperparameter optimization methods use the model's accuracy on the held-out validation dataset as an evaluation of the model. Albelwi et al. [1] took a different approach to evaluate how good a CNN model is. Images are reconstructed from the learned filters using Deconvnet, and the similarity between the original image and the reconstructed image is used as the fitness of a CNN architecture. There are various ways of defining models' structures too. Suganuma et al. [74] used a cartesian genetic programming encoding scheme to define CNN models. It enables them to incorporate modules such as tensor concatenation into the model architecture. Baldominos et al. [5] used the backus-naur form to define a grammar for model topologies, allowing more flexible representations of CNN architectures.

2.2 Method

EAs are inspired by the process of natural selection. They are commonly used to produce high-quality solutions when the search space is large. The idea of EA is to produce an initial population that is made up of diverse solutions and evolve them for many generations to produce better ones. The genes of the better solutions will survive and pass on to further generations, whereas the worse solutions die off.

Both Young et al. [97] and Real et al. [61] use EA, and their methods have limitations. For the purpose of optimizing hyperparameters for deep learning models, no constraints should be put on the network layers. Limiting the number of layers could make the network size too small for the problem, leading to underfitting. Young et al. [97] used a classical fixed length GA, the most popular type of GA, to optimize a CNN model for the CIFAR-10 dataset [48]. A fixed length GA means the CNN model has a predefined fixed number of layers. However, for a given problem, one never knows how many layers are enough before experiments are done. Having a too large network, on the other hand, can lead to overfitting. Thus it makes sense to start from a small network and gradually grow it as needed. Real et al. [61] does so by using a mutation only EA. Although high accuracies are obtained on the CIFAR-10, and CIFAR-100 dataset, the computational cost of their method is very high. The whole process takes 250 parallel computers and over 250 hours to complete. Those who rely on automatic hyperparameter optimization tools usually do not have that kind of computational resource.

Because of the above-mentioned limitations, we propose a variable length GA for deep learning hyperparameter optimization. It starts from a small network model and gradually builds on top of it. Crossover operation, along with other techniques, are used in this EA to accelerate the optimization process, making it affordable for those with limited resources.

2.2.1 Variable Length Genetic Algorithms

Classical GA requires fixed chromosome length. Since the number of convolutional layers varies in different CNN models, and more hyperparameters are involved as the model grows deeper, a variable length GA is more suitable for the purpose of our task.

A chromosome in the variable length GA contains parameters that define a solution, in this case, a hyperparameter configuration for a network model. How the solution is encoded in the chromosome is discussed in Section 2.2.2.

Figure 2.1 describes the overall procedure for the variable length GA. The initial population with two convolutional layers is first produced, and the individuals/solutions are initialized with random hyperparameters. The individuals are then evaluated and sorted based on how fit they are. The fitness of an individual is made to be the corresponding CNN model's accuracy on the validation dataset. Then the fitter individuals get directly picked and survive into the next generation. The rest of the next generation is produced by individuals from the current generation using crossovers and mutations, common operations of a GA. In the crossover, two individuals are selected as parents to produce a child that inherits the parents' properties. In mutation, some properties of the individual are changed to other values. After the 2-layer models have been evolved for some generations, the algorithm enters the next phase that enables more layers and hyperparameters.

A new phase in the algorithm simply means the chromosome length increases, bringing more hyperparameters in, and the corresponding CNN models have more layers.

The steps of our hyperparameter optimization method are as follows:

1. Initialize the number of populations p and generations g according to user input.

2. Create the initial population. Each individual in the population has randomized hyperparameters. The size of the population is p as specified by the user.

3. Evaluate the fitness of each individual in the current generation. The fitness is the accuracy of the individual on the validation dataset.

4. Sort each individual according to their fitness value, from high to low.
5. Select the fittest individuals. Select a portion of the population that has the highest fitness values, and let them survive into the next generation. The percentage of this survival rate can be set manually.
6. Allow some less fit individuals to survive. For the rest of the individuals in the current population, give them a small chance to survive into the next generation. This probability is set manually.
7. Randomly mutate some individuals in the next generation. The mutation rate is manually configured. If an individual is chosen to mutate, one of its hyperparameter's value will be changed.
8. Produce new individuals. Randomly select two individuals from the next population to serve as parents, and perform crossover operation to produce a child. Each hyperparameter in the child is randomly set to be one of its parent's hyperparameter value. Repeat the crossover operation on different random individuals many times until the next generation has p individuals. Now the next generation becomes the current generation.
9. Repeat Step 3 - 8 until the number of generations has reached g.
10. Select the best individual from the current population.
11. Produce population with longer chromosome length. For each individual of the population, one part of the chromosome is from the best individual in Step 10; the other part is randomly generated.
12. Repeat Step 9 - 11 until convergence.
13. Select the best individual and train for more epochs until convergence.

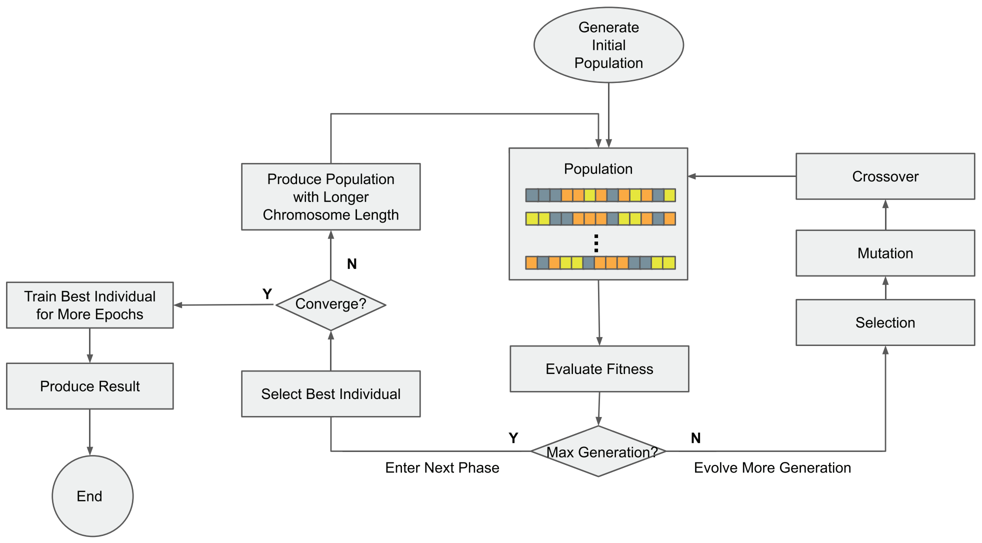


Figure (2.1) The hyperparameter optimization steps using our variable length GA.

2.2.2 Encoding Scheme

The hyperparameter information of each individual, or CNN model, is encoded into what's called a chromosome in GA. The chromosome may have different representations, such as a bit string that contains 1's and 0's. For the purpose of hyperparameter optimization, the chromosome of our experiments has many fields that contain different values. It defines the hyperparameters of a CNN architecture, such as the number of neurons, activation function type, and so on. Once a chromosome is known, a CNN model can be built according to the chromosome.

In the initial phase (Phase 0) of the algorithm, there are two convolutional layers, a and b. Relevant hyperparameters are encoded in the chromosome, as shown in Figure 2.2. For each convolutional layer, the hyperparameters include the number of output feature maps, the size of the convolutional windows, and whether or not to include batch normalization. For this two-layer block, there are three additional hyperparameters: whether to include pooling in this block, what is the pooling type, and whether or not to include a skip connection. If there is a pooling layer for this block, it is going to be added after the first convolutional layer. If a skip connection exists, it is going to have a 1x1 convolution, and its output is

No. of Output Channels for Layer a	Conv kernel Size for Layer a	Activation Type	Include Pool?	Pool Type	Include BN for Layer a?	No. of Output Channels for Layer b	Conv Kernel Size for Layer b	Include BN for Layer b?	Include Skip?

Figure (2.2) The initial chromosome (Phase 0). In the chromosome, what type of activation function to be used for the whole model is encoded. It also includes hyperparameter information for the first two convolutional layers: the size of the convolutional windows, number of output channels, and whether or not to include batch normalization for the layer. For the whole two-layer block, there are three additional hyperparameters: whether to add a pooling layer after the first convolutional layer, what is the pooling type, and if a skip connection should be included.

going to be added with the output of the whole block. Figure 2.3 is an example of the skip connection. There is an additional "activation type" field in the chromosome for Phase 0. This defines what type of activation function is going to be used for the whole model. Figure 2.2 is the encoding scheme of Phase 0, the initial phase that represents a model with two convolutional layers.

For every phase after Phase 0, we allow the algorithm to still add two convolutional layers (Layer a and Layer b) in each phase, but we give the flexibility and let the algorithm decide whether to add one or two convolutional layers in the phase. The chromosome encoding for a Phase after Phase 0 is as shown in Figure 2.4. New fields are concatenated to the chromosome of the previous phase. The new part is very similar to the encoding for Phase 0, except that it does not have an "Activation Type" field and has a new field, "Include Layer?" that can take two values: 0 or 1. If the value is 1, there will be two convolutional layers in this phase, and if 0, there will be only one.

Going into a new phase means that the chromosomes are longer, and models being discovered by the algorithm grow deeper. And part of the hyperparameters in the deeper model comes from the best model in the previous phase. An example can be found in Figure 2.5. Assume to the left is the best model discovered in Phase 0. The part in the dashboard box is encoded in Phase 0 chromosome. When the algorithm enters Phase 1, the chromosomes grow longer, and the corresponding CNN models are deeper. The deeper

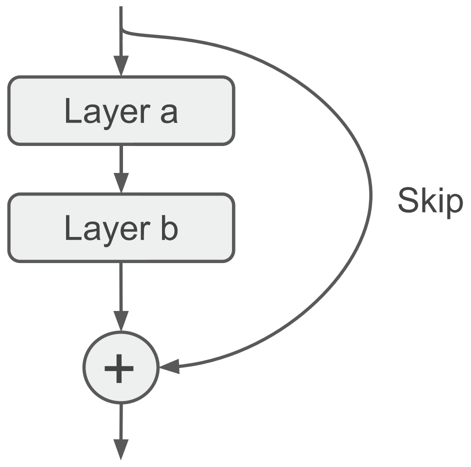


Figure (2.3) An example of a skip connection. This block contains two convolutional layers: Layer a and Layer b. The skip connection performs a 1x1 convolution on the input, and its output is added with the output of Layer b. Stride size and number of filters of the skip connection is setup such that its output tensor has the same shape as that of Layer b.

models share some common properties: part of their hyperparameter settings come from the best model in Phase 0. See the right part in Figure 2.5 for an example of Phase 1 model. The part bounded by the dashed box is being searched in Phase 1, and the gray part above is from Phase 0. Figure 2.6 shows the corresponding chromosomes for the two models in Figure 2.5. In Figure 2.6, the yellow fields indicate hyperparameters that are being searched in that phase, and gray fields indicate hyperparameters from previous phases.

2.2.3 Fitness of individuals

The fitness of an individual is its accuracy on the validation dataset. One way is to train the models to converge and then compare. While this is more accurate in terms of comparing the models' performance on the test set, this process takes very long time and needs a great amount of computing power. To save time, models can be trained for just a few epochs to compare their relative fitness with each other. Models that are fitter in the first few training epochs tend to also perform better later. In the experiments, the number

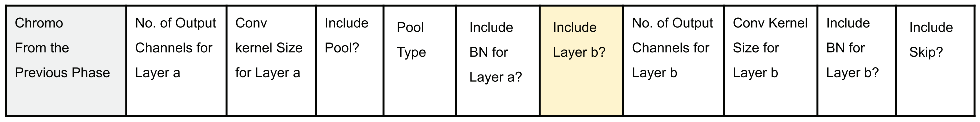

Chromo From the Previous Phase
No. of Output Channels for Layer a
Conv kernel Size for Layer a
Include Pool?
Pool Type
Include BN for Layer a?
Include Layer b?
No. of Output Channels for Layer b
Conv Kernel Size for Layer b
Include BN for Layer b?
Include Skip?

Figure (2.4) Encoding for Phases after Phase 0. Chromosome in the current phase is an extension of the one from the previous phase plus some new fields. Chromosomes are longer, and the represented models have more layers. The field with gray background indicate the chromosome from the previous phase. The rest defines the properties of new layers in the current phase. In the current, two convolutional layers can be added, with the flexibility of letting the algorithm to decide whether to add one or two. This is achieved by adding a "Include Layer b?" field, the one with yellow background, in the chromosome.

of training epochs for comparing relative fitness is set to be 5.

This way of evaluation is unfair when the size of the model differs greatly. A deeper model might have similar or even worse test accuracy compared with a shallower model when both are only trained for a small number of epochs, such as five. To solve this problem, we introduce weight transfer from shallow models to deeper ones. In Figure 2.7, when the algorithm enters a new phase, it searches for deeper models (the right part). New layers are added to the best model from the previous phase (the left part). Instead of initializing all the weights randomly in the new models, some of their weights are transferred from the previous best model, and some are done through random initialization. This takes advantage of the previously trained models and saves some training time. To evaluate its fitness, the deeper model is simply trained for five epochs since part of it already received some training.

2.3 Experiments and Results

2.3.1 Datasets

Experiments were done on the CIFAR-10 dataset [48]. This dataset contains 60,000 color images of size 32X32 in 10 classes: each class has 6,000 images. There are 50,000 images for training and 10,000 images for testing.

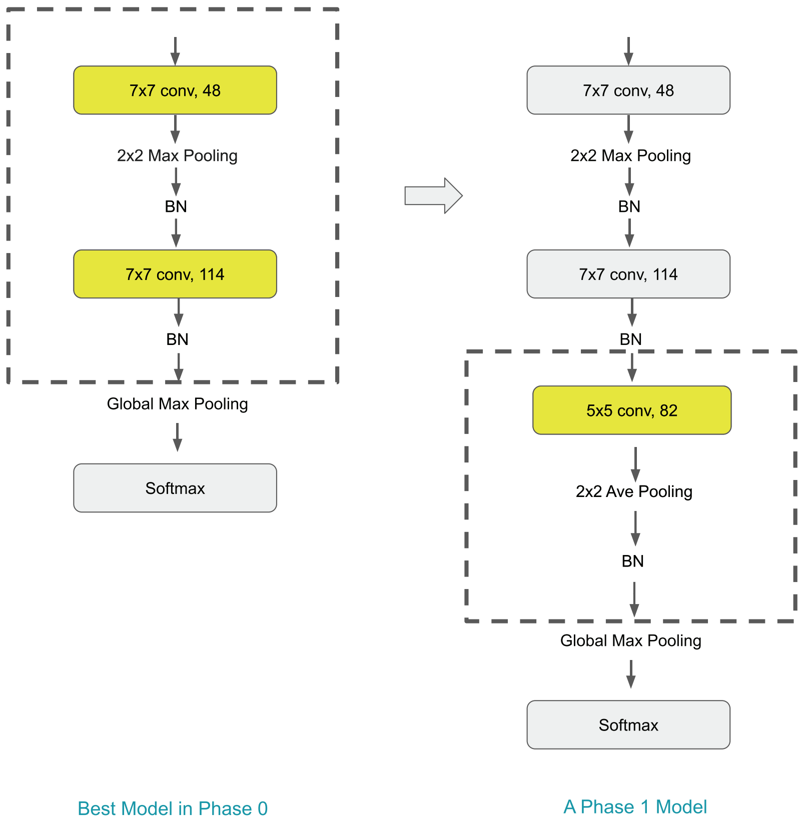


Figure (2.5) The model grows deeper when a new phase begins. The models in the new phase are built based on the best model in the previous phase. To the left is an example of the best model found in Phase 0, and to the right is a model in Phase 1.

2.3.2 Search Space

The search space defines all the possible solutions that can be searched. A relatively small search space may take the algorithm less time to find a satisfying CNN model, but may be constrictive due to the limited number of possible models included in this space. A larger one, on the other hand, is more time-consuming and requires more computational power.

In our experiments, the search space includes the number of output feature maps in each convolutional layer, convolutional filter size, activation function type, pooling type, whether to have a skip connection, whether to include batch normalization in a layer, and the total

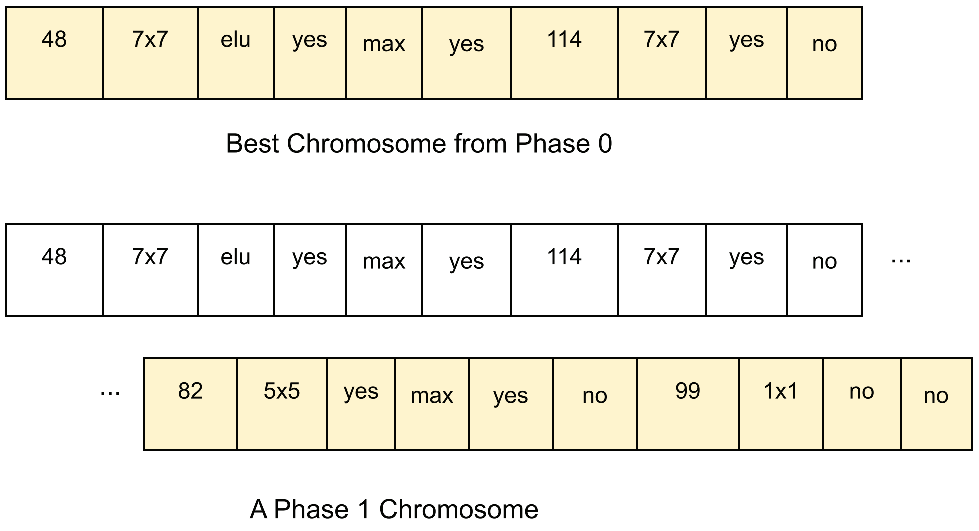


Figure (2.6) The corresponding chromosomes for models in Figure 2.5. Chromosomes grows longer when a new phase is entered. The chromosome in the new phase is extended from the best chromsome in the previous phase.

number of layers. Most of the hyperparameters, such as the number of output feature maps and convolutional filter sizes, need to be decided in every convolutional layer. Thus as the model grows deeper, the search space becomes exponentially larger and the searching more difficult. Most of the previous works do not include pooling types in their search space. In our work, however, various pooling layers can exist in a single model. Table 2.1 describes the default search space of our experiments. The left column is the types of hyperparameters, and the right column is their choices. Unless otherwise noted, our variable length GA uses this space. Figure 2.8 shows how the search space of the algorithm grows as the number of layer increases. Initially, models have two layers and the search space contains 156,800 different combinations of hyperparameters. When the number of layers increases to 14, the search space grows to about 10^{35}.

The possible options for each hyperparameter type are carefully chosen. ReLU, Tanh, ELU, and SELU are possible choices for the activation functions. Tanh is chosen in favor of Sigmoid because it is zero centered.

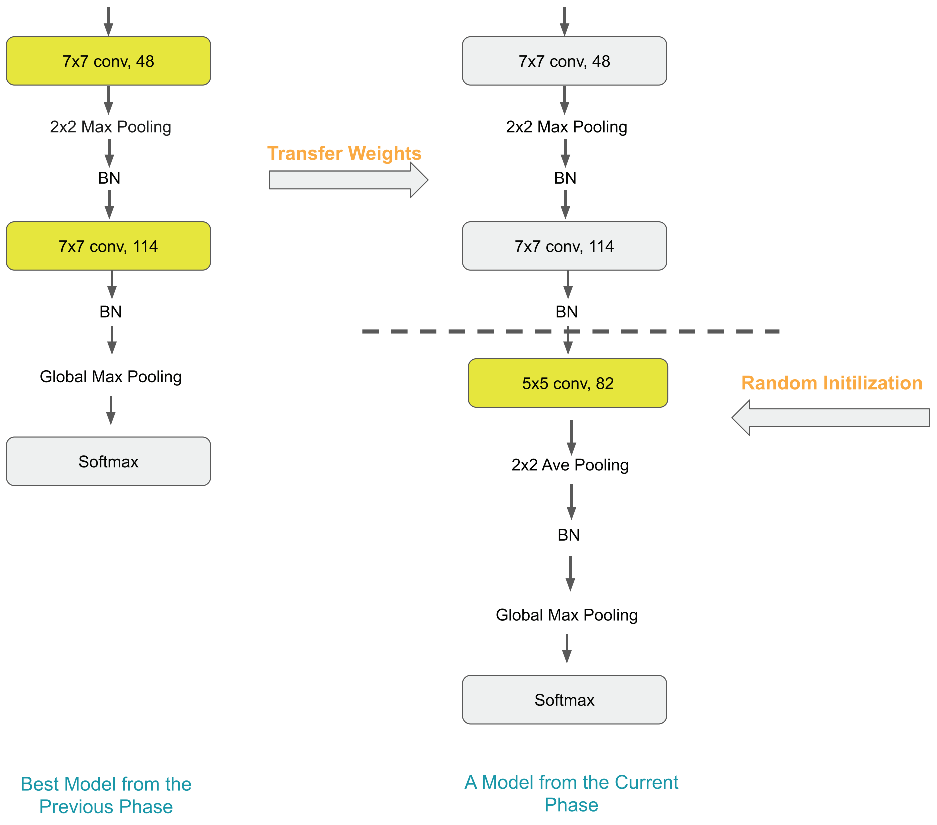


Figure (2.7) Weight initialization scheme for deeper models. A deeper model of the current phase is built on top of the best model from the previous phase. Part of its weights is transferred from the previous best model, part is initialized randomly.

2.3.3 Experiment Setup

Our proposed method is implemented in Python, and the CNN models are created using Keras. Experiments were run on a single GPU. To save time, the fitness of each individual is made to be the test accuracy after five training epochs. Weight transfer is also used to save the experiment running time. If a model survives into a new generation, it does not need to be trained again because all of its weights are saved. If the algorithm enters a new phase where the new population with longer chromosomes is built on top of the previous stage's best individuals, the old best model's trained weights are transferred into the new models, and the new ones are trained for another five epochs to get their fitness value.

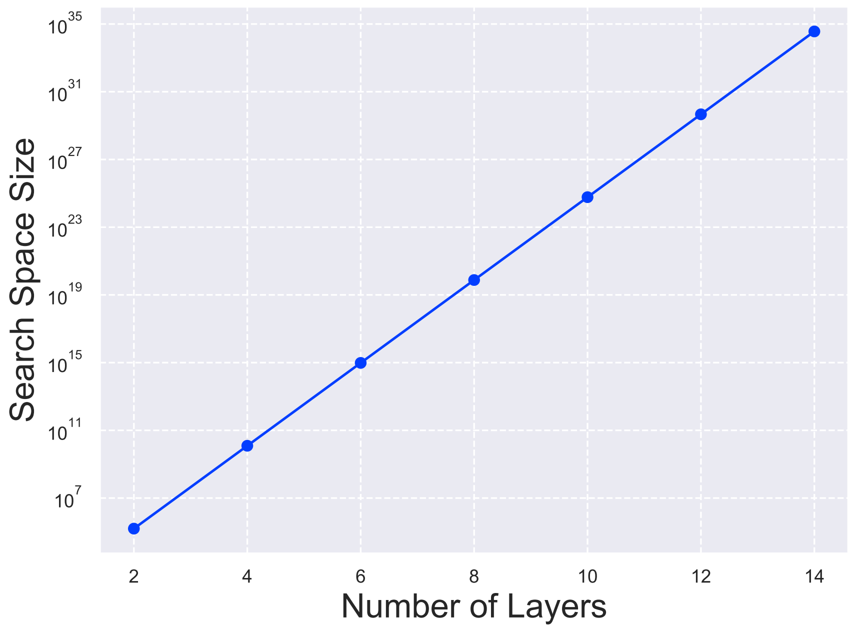


Figure (2.8) The size of the search space grows exponentially as the number of layers in the deep learning model increases. In the initial phase of our algorithm, models have two layers and the search space is 156,800. As the model goes deeper, this number grows exponentially. When the number of layers reaches 14, the search space grows to about 10^{35}.

2.3.4 Genetic Algorithm Settings

There are parameters that need to be set up in our variable length GA: the size of the population, number of generations, number of phases, mutation rate, and survival chance of less fit individuals. In the experiments, 50% of the fittest individual survive into the next generation, the survival rate of less fit individuals is set to be 0.2, and the mutation rate is set to be 0.2. We do not set a predefined number of phases in our method. Instead, a stopping condition is set to stop the algorithm. Section 2.3.5 discusses the stopping condition in details.

2.3.5 Stopping Condition

The evolutionary process stops when there is no more improvement in validation accuracy when more layers are added. A threshold value is set for the stopping condition: when the best fitness in the new phase is smaller than the best fitness in the previous phases

Table (2.1) The default search space used by our variable length GA.

Hyperparameter	**Choices**
Number of Output Channels	8, 16, 32, 64, 128, 256, 512
Convolutional Filter Size	1x1, 3x3, 5x5, 7x7, 9x9
Activation Function Type	ReLU, Tanh, ELU, SELU
Pooling Type	Max pooling, Average pooling
Skip Connection	Yes, No
Batch Normalization	Yes, No
Number of Layers	≥ 2

by more than 0.01, the evolving process stops, and the individual with the best fitness is returned. This individual then receives further training. In Figure 2.9, the evolutionary process stops at the 14th phase. The best individual fitness at Phase 14 is 0.8070, whereas the best in the previous phases is 0.8233 (at Phase 13). The evolutionary process thus stops, and the best individual at Phase 13 is picked to receive further training.

2.3.6 Results

Figure 2.10 shows the result of 20 individuals evolved for 5 generations in each phase. Each individual is trained for 5 epochs to evaluate the fitness. The evolution stops at Phase 14 when the stopping condition is met. The number of convolutional layers increases in each phase. Initially, when the algorithm enters a new phase, there is a great improvement in the accuracy of the individuals. This improvement gradually declines in the later phases. This shows the increase of convolutional layers can have a great influence on the model performance, especially when the model is relatively shallow. When the network is shallow, its size could be too small to fit the problem. As the number of layers increases, the size of

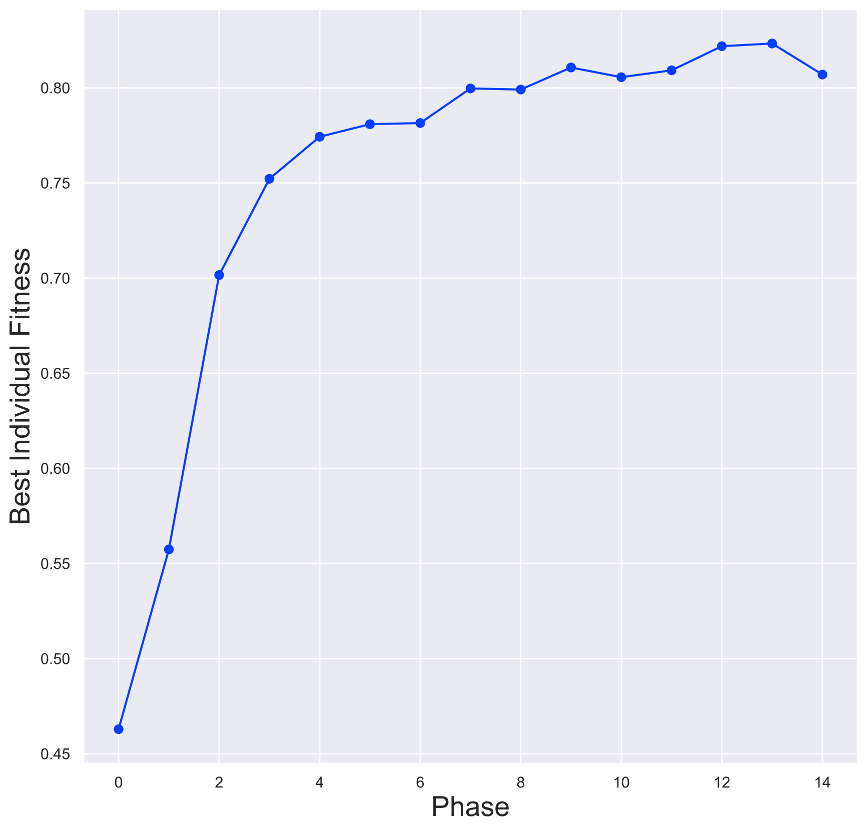


Figure (2.9) The evolutionary process stops when the stopping condition is met. In this example the algorithm stops at Phase 14 because the fitness drop is larger than the threshold. The best individual, in this case at Phase 13, is picked to receive further training.

the network grows, and so is the capability of the network.

We compare the results of our variable length GA with random search, classical GA [97], and large scale evolution [61].

Unlike our method that allows for a dynamic search space that grows as the number of layers increases, random search requires that the search space to be fixed, which means the maximum number of layers need to be specified. We make the number of layers for random search in the range $[2, 10]$. Other search space settings are similar to our method.

We implemented classical GA [97] with the same GA settings mentioned in the paper, except that instead of using a population of size 500, we use much smaller population sizes. Because the fitness score is the model's test accuracy after being trained for 4,000 iterations, it takes a very long time to evaluate one model. Evaluating the initial 500 models would go

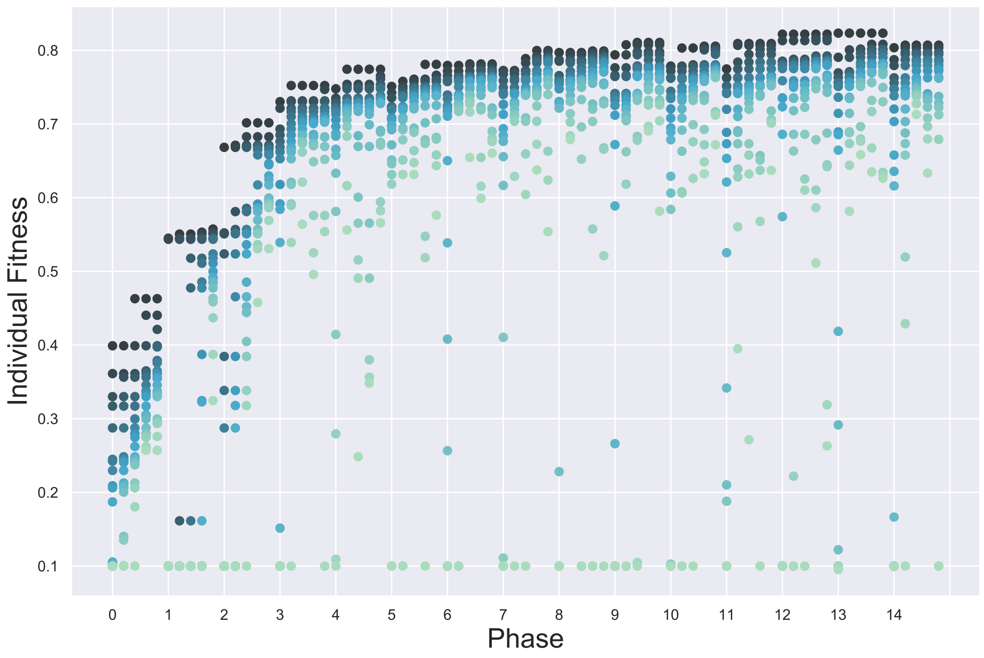


Figure (2.10) The fitness of 20 individuals evolved for 5 generations in each phase. The algorithm stops at Phase 14 when the stopping condition is met. The best individual found in this evolutionary process is chosen to receive more training. The overall evolution time is less than 25 hours.

far beyond 30 hours. If a 30-hour limit is enforced, the final result would simply be one of the models from the initial population, without any evolution involved. Figure 2.11 shows the results produced by the classical GA. A 30-hour time constraint only allows evaluation for around 75 models. Thus the more individuals in a population, the fewer generations the population will evolve. If there are 50 individuals in the population, the algorithm will stop in the middle of the second generation, and we do not have many benefits from evolution.

We implemented large scale evolution [61] on top of a publicly available implementation [55]. We ran the implementation with different population sizes: 30, 50, and 80. The fitness of discovered individuals is shown in Figure 2.12. Large scale evolution can produce good models when given a great amount of computing hardware and time. However, when there is limited time and computational hardware available, it is not an ideal algorithm to use. As can be told from Figure 2.12(a), when time is restricted to 1 GPU and 30 hours, the fitness of discovered individuals are relatively low. When the population size is 30, the fitness starts to plateau. This could be due to the lack of variety in the chromosome because the population size is small. When the population size is 80, there is still room for the fitness to improve, but because of a mutation only evolution, it does not grow as fast as our variable length GA shown in Figure 2.10.

Table 2.2 shows the comparison between our method and random search, traditional GA [97], and large scale evolution [61]. In our method, we sample 20 individuals from the initial search space and evolve for 5 generations in each phase. The best model discovered has 88.92% test accuracy, and the total time is around 25 hours. We ran random search, classical GA [97], and large scale evolution [61] with the 30-hour constraint and reported the best result found. When we have limited GPU resources, within a similar amount of time, large scale evolution gives a model with the lowest accuracy. This is because large scale evolution starts with very small models, and there are only mutation operations on chromosomes. Without crossover, the evolution process is much slower. Large scale evolution gives very good results when hardware and time is not a problem. However, it is not practical for the average research lab. Classical GA gives a model with relatively high accuracy. This is because the defined search space is much smaller: the number of convolutional layers is fixed to be three, according to the paper. It is necessary to fix the layer number because classical GA has fixed length chromosomes. Setting a predefined number of layers is not very practical when we encounter a new problem because we usually do not have much idea of how many layers would be a good fit for the problem. Our method can find better models compared with the other three when there are limited computational resources available.

2.4 Summary and Future Work

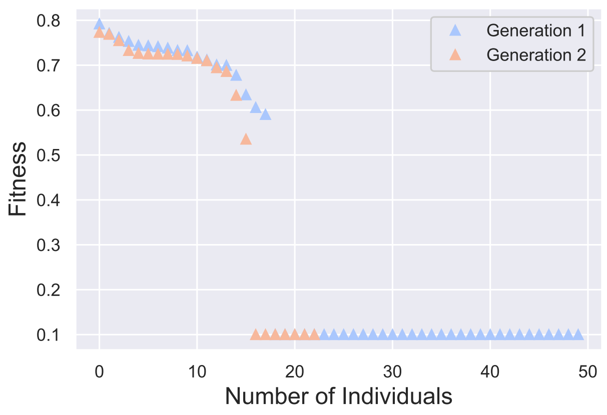
In this work, we propose to use a variable length GA to efficiently find good hyperparameter settings for deep CNNs. Experiments were performed on CIFAR-10 dataset, and results are compared with random search [9], classical GA [97], and large scale evolution [61]. Experimental results show that our algorithm can find much better models within a time constraint.

Table (2.2) Accuracy Comparison of Discovered models when using similar time (less than 30 hours).

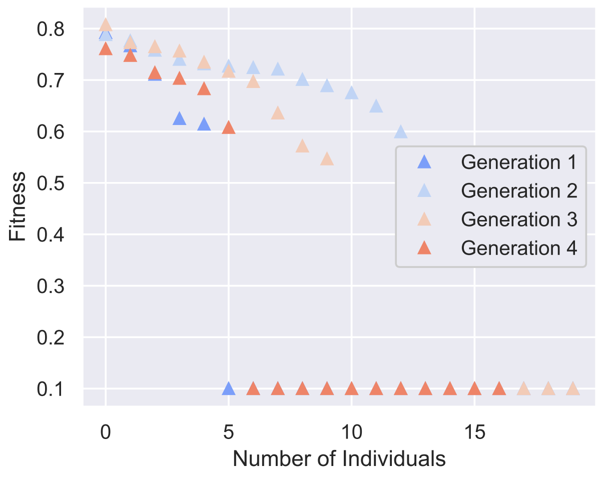
Method Name	**Accuracy and Time Comparison**	
	Accuracy	***Time(hrs)***
Random Search	58.66%	30
Classical GA	80.75%	30
Large Scale Evolution	51.90%	30
Variable Length GA (Ours)	88.92%	24.55

One limitation of our approach is that our method searches a sequential deep learning architecture and can only produce sequential models. Sequential architectures are among the most popular and work really well. Optimizing hyperparameters for these kinds of models is already a hard problem. There exist novel deep learning models with more flexible architectures. And using our current chromosome encoding and evolutionary strategies, we cannot discover those models.

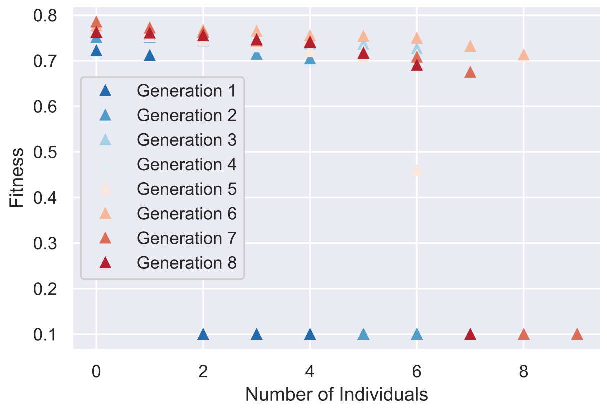
There are ways of making our algorithm more efficient. For example, right now, in each phase, all models are trained for 5 epochs for fitness evaluation, even if a model has poor hyperparameter configurations. In the future, we can detect these bad configurations in the early stage of fitness evaluation, so we do not spend more computational resources in training them. Other EAs such as ant colony can also be applied to this problem [83][84]. We have used fuzzy set theory and evolutionary computing to design a genetic fuzzy routing algorithm in wireless networks [53]. We have also designed a deep fuzzy neural network for cancer detection [6]. In the future, we plan to extend our method to more deep learning areas such as optimizing the architectures of deep fuzzy models.



(a)

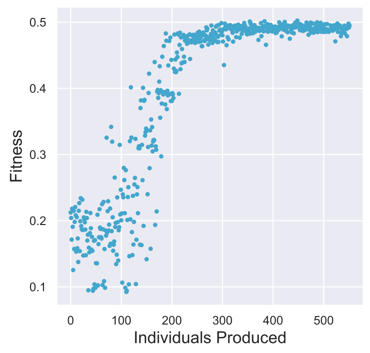


(b)

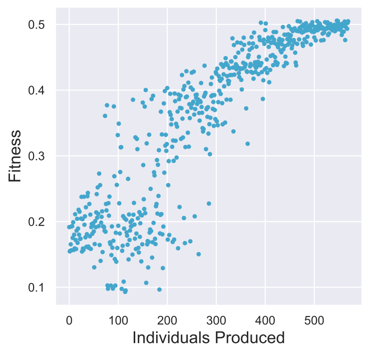


(c)

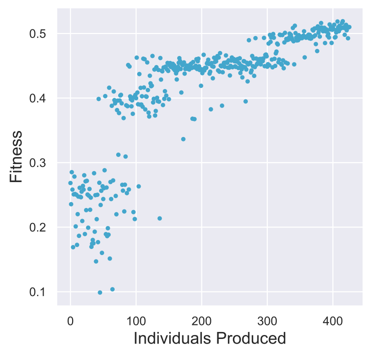
Figure (2.11) We implemented classical GA and ran the experiment with various population sizes and 30-hour time constraint. We sort the fitness of individuals in each generation and plot their fitness. (a)(b)(c) are plots for 50, 20, and 10 population respectively.



(a)



(b)



(c)

Figure (2.12) We implemented large scale evolution on top of a public repository, and ran the experiment with different population setting for 30 hours. (a)(b)(c) are plots for 30, 50, and 80 population respectively. The fitness of the best discovered individuals are slightly above 0.5 when time is restricted to 30 hours on 1 Nvidia Tesla V100 GPU.

PART 3

FAST DEEP LEARNING TRAINING THROUGH INTELLIGENTLY FREEZING LAYERS

3.1 Background

One problem with deep learning models is that the training time is very long. This work attempts to shorten the training time of deep learning models while having similar test accuracy. It is known that layers in a deep neural network learn at different speeds [93][105], and it is unnecessary to have all parameters participate in training at all times [10]. This inspired us to intelligently freeze layers to accelerate training. Freezing a layer avoids modifying its weights. It means that if we do not want to modify the weights of a layer, backward passes to that layer can be avoided, and speedups can be gained. This is achieved by identifying the layers which require less training using a mathematical formula. Our method is different compared with other simple freezing based on a schedule [10]. Time is reduced by intelligently freezing certain layers in the training processes. The contribution of our work is as follows:

- Our method opens a new way of accelerating the training of deep learning models while having similar model accuracy. With further exploration and experiments on this method, there is more room for improvement.
- A formula is designed to calculate freezing rates for layers in the model. Then the maximum subarray sum is calculated on the array of freezing rates to decide how many layers from the beginning layer should be frozen. This paves the way for the future design of intelligent freezing.
- In order to reduce the computation time overhead, we selectively and periodically calculate the gradient information during the training process.

3.1.1 Deep Learning Models

Deep CNN is a class of deep learning models that specifically deals with images. A typical CNN consists of convolutional layers, pooling layers, fully connected layers, and ReLU activation layers. The convolutional layers are the core components of a CNN. They act as feature extractors and can preserve the local connectivity information of an image. Pooling layers help reduce the number of parameters in the network and also control overfitting. A fully connected layer has full connections to the activations in the previous layer. And ReLU adds non-linearity to the model. Many recent deep learning models have batch normalization layers [41] to help with training speed. One difficulty of training deep models is that the distribution of the inputs to layers is not fixed, and it may change with weights updates in each batch. To further clarify, the weights of the model are updated layer-wise in backward using an estimate of error. This error estimate, such as gradient, tells how to update each weight in a layer under the assumption that input to this layer is the same as the previous. That means weights are updated with the expectation that the output of the previous layer has the same distribution. This slows down the learning process by requiring smaller learning rates and may require more epochs to converge. Batch normalization standardizes the inputs of a layer for each batch, so the input distribution will not be much varied. Thus it will stabilize the learning and reduce the time and number of epochs required.

There are many different CNN architectures. The VGGNet [68] is a deep CNN architecture that involves linearly stacking various components/layers. More recent architectures tend to have more complicated structures. Deep Residual Networks (ResNets) by He et al. [28] features skip connections in the network architecture. This allows feeding the output of a layer to another layer by skipping a few layers in between. Dense CNN (DenseNet) [37] is an extension of the ResNets with more skip connections involved. All the above models are with millions of parameters, and the basic units are non-linear units. This makes the progression in optimization very slow, as millions of parameters on multiple non-linear units need to be tuned in order to get good performance. On the other hand, training requires some hyperparameters to be set up, such as the learning rate and its decay, mini-batch size, and so on. These hyperparameters are difficult to decide, and their effect is critically important for the convergence. The above-mentioned training process always demands heavy computations and a great amount of time.

3.1.2 Optimizing Deep Neural Networks

SGD is one of the most popular optimization algorithms used for training deep architectures. Gradients of the loss function are calculated with respect to all the weights in the network. A learning rate is specified by the user, and the weights are updated using the gradient information and the learning rate. To help with the convergence of the network, an update rule is usually used to decrease the learning rate over time. SGD is usually used together with backpropagation, an efficient method for computing gradients for expressions. It involves iterative applications of the chain rule. On the high level, an error is calculated at the loss function, and then this information is propagated backward through the network layers. Gradients are calculated, and weights are updated along the way.

3.1.3 Time Reduction Methods

There are various researches on how to reduce training time for deep learning models, for example, adjusting the learning rate, parallelizing the training, reducing the model size, modifying the computations for convolutions, using batch normalizations, and so on.

Working on the learning rate can help accelerate the training or improve accuracy. Instead of being manually set, learning rates can be dynamically computed. Also, parameters in the network can have different learning rates. Our freezing method also falls into this category. For the details of learning rate related prior art, refer to Section 3.1.2 - 3.1.6.

Training can be parallelized in order to gain speedups. Miranda et al. [57] partitioned the training task into multiple training subtasks with multiple sub-models, which can be performed independently and in parallel. After training, sub-models are then merged to provide a pre-trained initial set of weights for the original model. Harlap et al. [27] proposed PipeDream, a training system that parallelizes computations by pipelining executions.

Experiments were performed on five different deep learning models using various numbers and types of machines. Results show that it is 5x faster in time-to-accuracy compared with data-parallel training. Zlateski et al. [110] introduced ZNN, a software package for CNN training on multi-core and many-core CPU machines. You et al. [96] implemented averaged SGD, an algorithm with fast convergence, in an asynchronous manner. The goal is to further reduce training time compared with distributed algorithms on traditional SGD. Landola et al. [39] presented FireCaffe, which parallelizes training of deep learning across a cluster of GPUs. Hardware with high bandwidth is selected, efficient communication algorithms are used, and batch size is increased to make communication more efficient. Results show that FireCaffe can achieve a near-linear speedup on popular deep learning models.

Speedups can also be achieved by reducing the size of the model. Kim et al. [45] proposed an effective method to compress CNN networks and evaluated their method on smartphones. Significant time and energy reduction can be obtained with a small loss in model accuracy. He et al. [29] proposed an algorithm to prune channels in deep neural networks. Compelling speedups can be achieved with some accuracy loss.

Faster implementations can be used to accelerate the convolution computations. Mathieu et al. [56] presented a method for the training and inference of CNNs in the Fourier domain. Their method greatly outperforms two spatial domain-based convolution implementations, the state-of-the-art at the time, in terms of speed. Lebedev et al. [50] used a group-wise brain damage method to speed up CNNs. The method reduces the computation of convolutions to thinned matrix multiplications. Lavin et al. [49] presented fast algorithms to compute convolutions for small filters and small batch sizes.

There are many other time-reducing methods. For example, momentum [75], a widely applied technique used with SGD, helps the algorithm to move quickly towards the minima by using the accumulation of past gradients. Batch normalization [41] normalizes the input for each batch in each layer to help speeding up convergence. Salimans et al. [64] introduced weight normalization, a method that reparameterizes weights in a neural network, to speed up the convergence of SGD. Wang et al. [82] accelerated deep learning training by using inconsistent SGD to dynamically adjust training efforts on batches. Van Grinsven et al. [80] decreased the training time of CNNs on medical analysis tasks by dynamically focusing on training samples with greater difficulty. Transfer learning [94] can reduce the time to train a model on a new problem by reusing components of readily trained models on a different but similar problem.

3.1.4 Layer-wise and Parameter-wise Learning Rate

In the typical SGD, a global learning rate is specified, which means all layers update using the same step size. However, it is observed that layers in deep learning models learn at different speeds [93][105]. Instead of using a universal learning rate for all network parameters, Singh et al. [69] proposed to have different learning rates for different layers to speed up convergence. A function was designed to calculate the learning rate adaptively according to the two-norm of the gradient vector of a layer. Learning rates for layers with small gradients are scaled up so as to escape low curvature points. Experiments were performed on relatively simple networks such as AlexNet [48]. It is unclear how the algorithm performs on deeper and more complicated models and whether it could hurt accuracy on such models. You et al. [95] designed an algorithm to calculate the learning rate for each layer. Using the proposed method, they can scale up the training batch size without losing accuracy. And this allows more computational units to be added when using data-parallel synchronous SGD. Zhang et al. [105] proposed to train feedforward neural networks with layer-wise learning rate by using the approximation of back-matching propagation. Their method is tested on VGGNets of different depths using CIFAR-10 and CIFAR-100 [48]. Experiment results show that their method gets better accuracy compared with vanilla SGD and two other layer-wise learning rate method. However, overhead information is not provided, and it is unclear how much time tradeoff they have.

Instead of the whole layer sharing the same learning rate, individual parameters can have their own learning rates too. AdaGrad and Adadelta both have learning rates specific to each parameter in the network. A global learning rate is set, and a scaling factor is calculated for every parameter to obtain parameter-wise learning rates.

On the extreme, learning rates for some parameters can be zero. This has the same effect as freezing – not participating in the backpropagation training process. Section 3.1.5 talks about parameters' participation in training in detail.

3.1.5 Parameters' Participation in Training

Previous work has shown that it is not necessary to have every parameter participated in every training step. Srivastava et al. proposed dropout [72], a technique that randomly drops units during the training process. This method greatly reduces overfitting. Huang et al. [36] proposed stochastic depth, a method to randomly drop layers during training. It allows the test error to be reduced furthermore by increasing model depth. Their method only works on models with skip connections. Hettinger et al. [31] proposed FowardThinking, which trains neural networks one layer at a time. The previously trained weights are frozen during the training process. Brock et al. [10] proposed FreezeOut, which speeds up training by progressively freezing out layers. Frozen layers are excluded from the backward propagation, and weights are not updated for them. Different schedules have been experimented with on the FreezeOut technique. Their method has no improvements with VGG models.

3.1.6 Gradient Information Used

Gradient information is usually used to decide the learning rate for a specific network component, or the learning rate for the whole network at a certain training time. There are various methods on how the gradient information is used. Nesterov [59] adopted a first-order method to adjust the gradient of the current iteration based on the prediction of the gradient on the next iteration. This method has a better convergence rate in some situations compared with the vanilla gradient descent. Duchi et al. proposed AdaGrad [23], a method that adapts the learning rate to each parameter in the network. To calculate the specific learning rates, the l_2 norm of all the gradients on all previous iterations is used. Zeiler proposed Adadelta [101], an extension of AdaGrad. The exponential decaying average of squared gradients from previous iterations is used for the learning rate calculation. This restriction prevents the gradients from becoming infinitesimally small, a problem that exists in AdaGrad. Zhang et al. [105] used layer-wise adaptive learning rates calculated based on the back-matching propagation, which basically gives a value that is the gradient scaled by the inverse of a matrix. This helps get a scaling factor for the learning rate for each layer.

3.2 Method

In the typical training of deep neural networks models, all layers participate in the training process at all times. However, prior works [10][36][72] have shown that this may not be necessary. Different layers learn at different speeds. From observing the gradient magnitude for each layer, we can tell how fast it is learning. By using the gradient information, we can also investigate whether a layer is doing much useful work. If not, it can be frozen in order to save training time. The basic idea is to use a formula to calculate the freezing rates for all layers and then obtain how many layers from the beginning should be frozen using the maximum subarray sum on the array of standardized freezing rates. We then explore how frequent layers should be frozen.

3.2.1 Layer Level Freezing

To freeze a layer means to exclude it from the backpropagation process. Since parameters do not need to be updated at all times during training, certain layers can be frozen during some training epochs. Our freezing method works at the layer level because it is observed that gradients have similar magnitudes in the same layer [69]. Meanwhile, the inconsistency of gradients is mainly across layers [93]. Figure 3.1 shows the mean gradient magnitude of a ResNet-18 model trained with SGD using step learning rate decay.

Mean gradient magnitude, however, may not be a good measurement for deciding how much we want to freeze a layer. Intuition is that we want to freeze layers more when they are close to the optimum. This means that the gradients for those layers are relatively small. However, small gradients do not always indicate the closeness to optimum. The landscape of the function that we are trying to minimize can contain many saddle points and high loss plateaus, and they can lead to small gradients too.

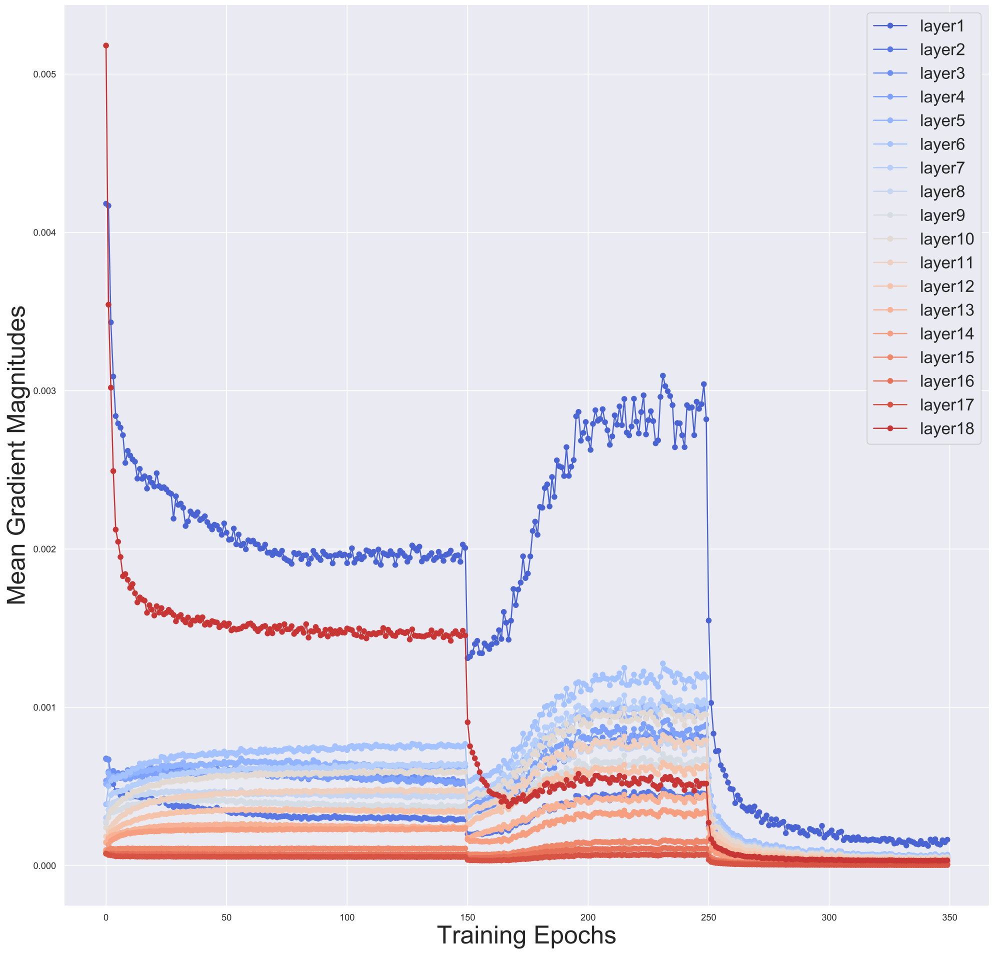


Figure (3.1) Mean gradient magnitudes of ResNet-18 for different layers. The model is trained for 350 epochs with step learning rate decay. The plot shows how the gradient changes as the training progresses.

3.2.2 Layer Freezing Rate

Directly using gradients to indicate how much we want to freeze a layer may not be a good idea based on the above-mentioned reasons. Instead, we consider the normalized difference of gradients as an important measurement. If the weights update in both positive directions and negative directions, and the updates eventually cancel each other, time is wasted doing all the updates. These weights can therefore be frozen because the gradients cancel each other anyway. This is the intuition behind using a normalized difference of gradients.

To better explain how the normalized gradient difference is calculated in our method, we introduce the concept of freezing rate in our framework. They will later be used to decide which layers should be frozen during training. The freezing rate of a layer describes how much we want to freeze it. The following formula defines the freezing rate for a layer:

$$F_l^{(k)} = \begin{cases} 1 & \text{if } g_l = 0 \\ 1 - \frac{\sum_{i=1}^{N} |\sum_{j=1}^{M} g_{lij}^{(k)}|}{\sum_{i=1}^{N} \sum_{j=1}^{M} |g_{lij}^{(k)}|} & \text{otherwise} \end{cases} \tag{3.1}$$

where $F_l^{(k)}$ is the freezing rate of layer l at k-th epoch. N is the number of weights in layer l, M is the number of iterations in the epoch, and $g_{lij}^{(k)}$ is the gradient of the i-th weight in layer l at j-th iteration in epoch k. According to Equation 3.1, if the gradients of weight change signs a lot, and cancel each other across iterations, $F_l^{(k)}$ will be large, which means the layer has a high freezing rate at that epoch. If most of the gradients update in the same direction across iterations, $F_l^{(k)}$ will be small, and the layer has a low freezing rate. The range of $F_l^{(k)}$ is between 0 and 1:

$$0 \leqslant F_l^{(k)} \leqslant 1 \tag{3.2}$$

When gradient updates are always in the same direction for every weight in the layer, $F_l^{(k)}$ is 0. On the other hand, when gradients completely cancel each other across iterations, $F_l^{(k)}$ becomes 1. Freezing rates indicate how much we want to freeze layers. Overall, we want to let layers that have lower freezing rate values receive more training and those with higher freezing rates more freezing.

Figure 3.2 shows the freezing rates for all the layers of a ResNet-18 model trained with SGD. Step decay is used: the learning rate is set to 0.1 for the first 150 epochs and decays to 0.01 for the next 100 epochs, and finally goes to 0.001 for the last epochs. As can be seen from Figure 3.2, different layers have different freezing rates. When the learning rate becomes smaller, layers generally have lower freezing rates. It can also be observed that the freezing rate for the last layer becomes very low towards the end of training compared with other layers.

To better visualize the relative freezing rates between layers, Figure 3.3 shows the freezing rates for all with the last layer removed.

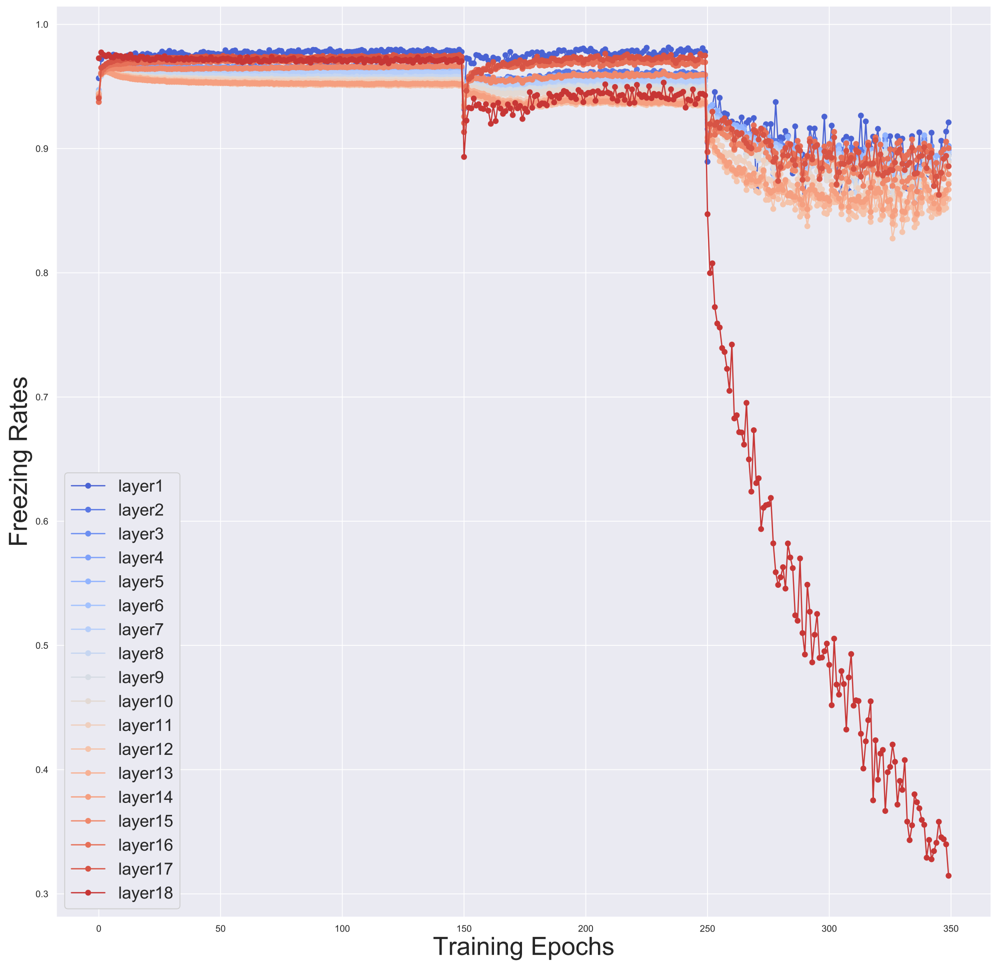


Figure (3.2) Freezing rates for all layers in ResNet-18. The model is trained for 350 epochs with step learning rate decay. The plot describes how the freezing rates change as the training goes on. We are going to use this information in our freezing process. To better see the freezing rates for the first 17 layers, refer to Figure 3.3.

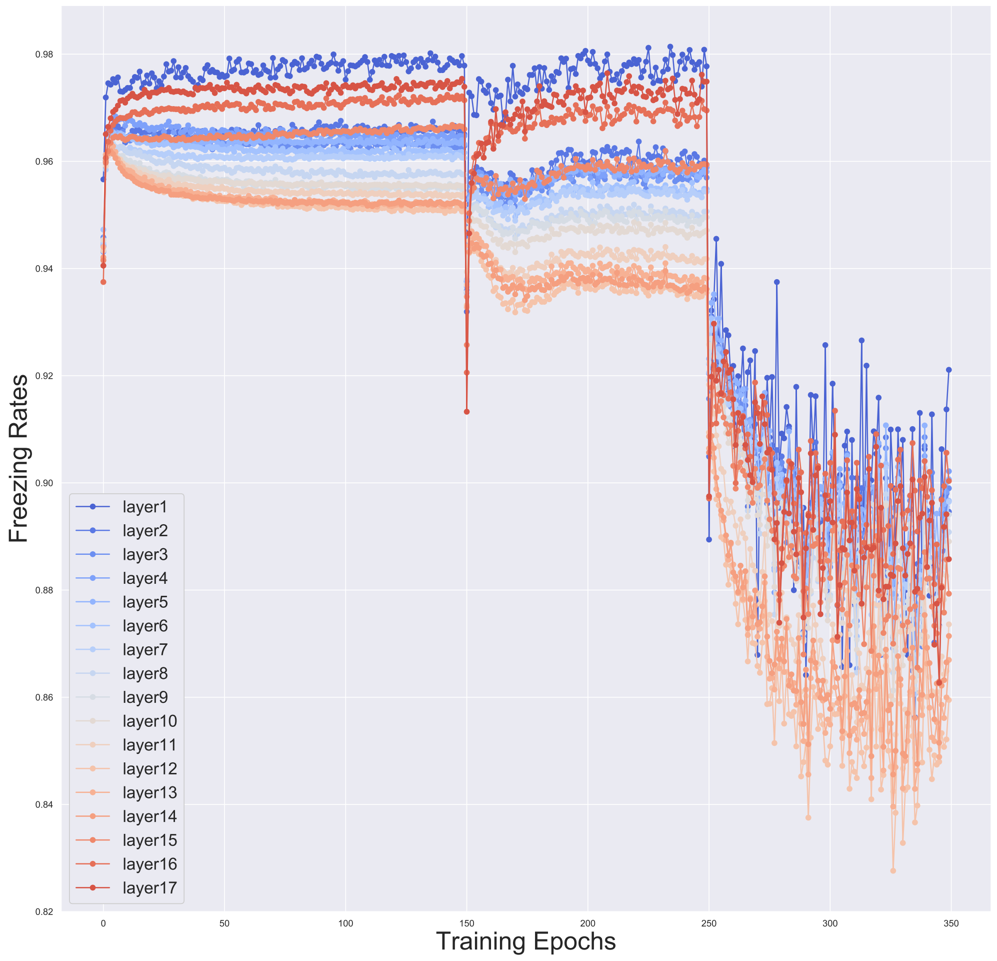


Figure (3.3) To better illustrate the freezing rate values for the first 17 layers, we remove the last layer in ResNet-18 in this plot, so that more details can be seen.

3.2.3 Layers to Freeze

Freezing rates cannot be directly used to decide which layers to freeze. If there is an input to a frozen layer that is not frozen, the gradients to the frozen layer are still calculated in order to get the gradients for the previous unfrozen layers. This means for the purpose of time reduction, freezing should start at the first layer and be consecutive, as shown in Fig 3.4(a). If frozen layers are not consecutive, gradients will still be calculated for those that have unfrozen layers before them. In Fig 3.4(b), Layer 1, Layer 3, and Layer n-1 are frozen. Since Layer 2 is unfrozen, its weights are still updated. To obtain their gradients, all of the following layers, including Layer 3 and Layer n-1, require gradient calculation. There is not much time reduction in this case.

If freezing is simply done using layer freezing rates, it is very likely that the above requirement cannot be satisfied. To handle this, we calculate the maximum subarray sum for the freezing rates. First of all, freezing rates for all layers, except for the first layer and last layer, are standardized to zero mean and unit variance using Equation 3.3:

$$F^{'} = \frac{F - \overline{F}}{\sigma} \tag{3.3}$$

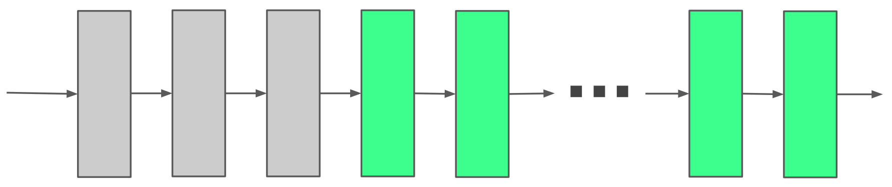
where F is the original freezing rates vector obtained using Equation 3.1, $\overline{F}$ is the mean of vector F, and σ is its standard deviation. $F^{'}$ is the standardized freezing rate vector.

The standardized freezing rates now form an array that contains both positive and negative numbers. Positive numbers indicate that we want to freeze these layers, and negative means otherwise. The goal is to find the maximum subarray sum starting from the first array element and return the end index.

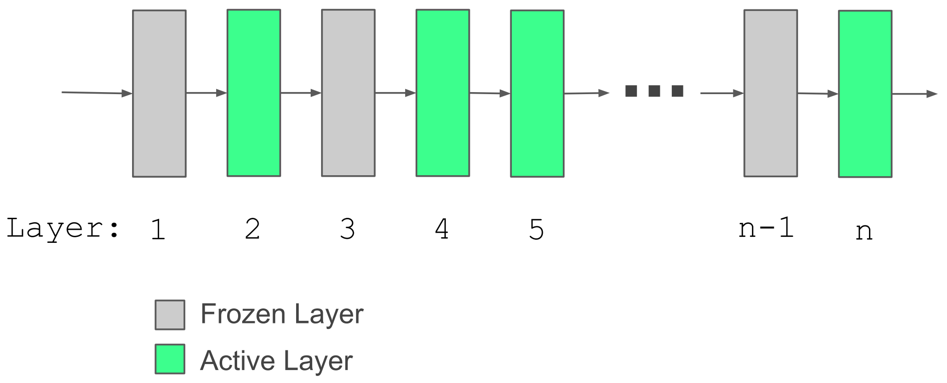
The end index n is calculated using Equation 3.4:

$$n = \arg\max_{k} \sum_{l=1}^{k} F_l^{'} \tag{3.4}$$

where $F_l^{'}$, obtained using Equation 3.1 and Equation 3.3, is the standardized freezing



(a)



(b)

Figure (3.4) For the purpose of time saving, frozen layers should always start at the first layer and be consecutive. If there is an input to a frozen layer that is unfrozen, gradient calculation is still required for the frozen layer. For example in (a), Layer 1, 2, 3 are frozen and the rest are unfrozen. The frozen layers do not have inputs that requires gradients. Thus we do not need to calculate gradients for Layer 1, 2, 3. In (b), Layer 1, 3 and Layer n-1 are frozen. However, since Layer 2 is unfrozen, we still need to calculate gradients for Layer 3 in order to know how to update weights in Layer 2. Similarly, gradient calculation is required for Layer n-1. In this case, there will not be much time saving.

rate for layer l. As a result, the layers that should be frozen will be from the first layer to the layer with end index n.

3.2.4 Overhead

Due to the calculation of freezing rates, time overhead is introduced. Since the goal is to reduce the training time, we want to keep the overhead small.

It is observed that the relative freezing rates for layers remain roughly the same per learning rate decay step. For example, in Figure 3.2, Layer 1 has the largest freezing rate compared with other layers in the first learning rate decay step (from epoch 0 to epoch 150). It remains so until the learning rate is decayed in the next step. Based on this observation, the layer-wise freezing rate can be calculated just once per learning rate decay step. Overhead is reduced this way.

3.2.5 Freezing Scheme

Our goal is to find a freezing scheme that reduces training time while having similar test accuracy. How many layers from the beginning should be frozen can be decided using the method mentioned in Section 3.2.3. We still need to explore how frequently those layers should be frozen.

To let things stabilize, layers are trained freely without freezing for some epochs before the freezing starts. Here, we tested a freeze-and-unfreeze method for all layers. This method requires several hyperparameters to be set up: how frequent are layers being frozen during each learning rate stage. In our experiments, we find the layers to freeze using the maximum subarray sum for the standardized freezing rates (3.4), and have them freeze using a specified freezing frequency, for example, every other epoch. To allow for more flexibility, we also find layers to freeze using the second maximum subarray sum and have these layers freeze at a different but lower frequency, such as every six epochs. The details on more hyperparameter setups are listed in Section 3.3.

Algorithm 1 describes the details of training deep models using SGD with intelligent freezing. For simplicity, a single fixed learning rate η is used in Algorithm 1. When training using SGD with step decay, the steps in Algorithm 1 is simply repeated for each different learning rate there is.

Algorithm 1: The training process with intelligent freezing using a fixed learning rate η

```
$\omega$ is network parameters.
$\eta$ is the learning rate.
$e_1$ is the hyperparameter for the number of training epochs without freezing.
$e_2$ is the number of training epochs with freezing.
$F_r$ is the freezing rates for all layers.
$L$ is the layers that should be frozen.
$f = [f_1, f_2]$ is the hyperparameter for how frequent layers should be frozen.
begin
    $optimizer$ = sgd_optimizer($\omega$, $\eta$)
    for $epoch = 0$ to $e_1$ do
        train()
        test()
    end
    Compute freezing rates for all layers according to equation 3.1.
    $F_r$ = get_freezing_rates($\omega$)
    $L$ = get_layers_to_freeze($F_r$)
    for $epoch = e_1 + 1$ to $e_2$ do
        $optimizer$ = get_optimizer($\omega$, $L$, $f$)
        train()
        test()
    end
end
```

3.3 Experiments and Results

We evaluate our method on a popular image classification dataset: CIFAR-10 [48]. The dataset contains 60,000 color images, with 50,000 for training and 10,000 images for testing. There are in total of 10 classes, and each class has 6,000 images.

Experiments were performed on various deep architectures: VGG models [68], ResNets [28], and DenseNets [37]. Most tests were run on a high performance computing server, with other programs running on it. For some tests, we were able to perform in a controlled environment, where there are no other programs running on it. To make sure we have a fair comparison, experiments were run multiple times, and the average is taken. Details can be found in the following sections.

3.3.1 VGG Models

Experiments were performed on VGG models in an environment with no other programs running. A computing node with a single NVIDIA Tesla K40 GPU was used. Both the vanilla SGD and our method trains the model for 350 epochs. Learning rate decay is used during training. The learning rate is 0.1 for the first 150 epochs, then 0.01 for the next 100 epochs, and 0.001 for the last 100 epochs. Accuracy on the test set and running time of the two algorithms are compared.

We first evaluate our method on a modified VGG-19 net. This model only has one fully connected layer of size 512. Other settings are the same as the 19-layer VGG net mentioned in [68]. Layers-to-freeze are obtained using the maximum subarray sum on standardized freezing rates for all layers according to Equation 3.4, and they are made to freeze every other epoch. We use the second maximum subarray sum to obtain another set of layers, and they freeze once every six epochs. Both our freezing method and the vanilla SGD were run three times for comparison. As shown in Table 3.1, for modified VGG-19, a 9.5% speedup can be achieved.

Table (3.1) The mean running time and test accuracy for VGG, ResNet, and DenseNet architectures on CIFAR-10. Comparison is between the SGD and our freezing method on top of SGD. The mean and standard deviation is calculated based on 3 runs. VGG-19 Experiments were done in a controlled environment on a single NVIDIA Tesla K40 GPU. ResNet-18 and Resnet-50 experiments were run on a single NVIDIA Tesla V100 GPU on a HPC cluster. ResNet-101 experiments were run on a single NVIDIA Tesla P100 GPU on a HPC cluster.

Network Name	**Comparison Between SGD and Our Method**			
	SGD Time (hrs)	***SGD Test Accuracy***	***Ours Time (hrs)***	***Ours Test Accuracy***
Modified VGG-19	6.21±0	93.72±0.26	5.62±0 (-9.5%)	93.73±0.16
ResNet-18	1.62±0.076	95.30±0.02	1.33±0.04 (-17.9%)	95.30±0.33
ResNet-50	5.40±0.13	95.35±0.17	5.03±0.01 (-7.1%)	95.27±0.09
DenseNet-121	9.03±0.17	95.66±0.07	7.91±0.01 (-12.3%)	95.38±0.16

3.3.2 ResNets

We tested our method on ResNet architectures. Experiments were run on a high-performance computing cluster. A computing node with a single NVIDIA GPU was used. We made sure that the experiments for the same deep architecture are done on the same GPU. The training setting is the same as in the VGG experiment.

ResNet-18 experiments were run on a single NVIDIA Tesla V100 GPU. As shown in Table 3.1, for ResNet-18, a 17.9% speedup can be achieved. ResNet-50 experiments were also conducted on a single NVIDIA Tesla V100 GPU. For ResNet-50, our method can reduce the training time by 7.1%.

There is a slight decrease in test accuracy for freezing ResNet-50. In order to explore the possible causes for this decrease, we plot the test accuracy for training ResNet-50 using SGD with freezing and vanilla SGD. As shown in Figure 3.5, in the first and second learning stage (when the learning rate is 0.1 and 0.01), SGD with freezing has similar or better test accuracy compared with vanilla SGD. However, in the last stage (when the learning rate is 0.001), the accuracy for SGD with freezing is not as good. This suggests that there might be too much freezing during the second learning stage and this is having an impact on the third stage. We thus lower the freezing frequency on the second stage, and observe how this affects the test accuracy. Layers obtained using the maximum subarray sum of freezing rates are frozen once every 4 epochs, and layers obtained using the second largest subarray sum are frozen once every 8 epochs. Table 3.2 shows the results for SGD with freezing using reduced freezing frequency. The test accuracy for ResNet-50 with reduced freezing frequency based on three runs is higher than that of the vanilla SGD. A speedup of 8.2% can be achieved.

Table (3.2) We reduce the freezing frequency in the second learning state (the learning rate is 0.01) due to the observations made. Experiments were run three times for both the freezing SGD and vanilla SGD. Mean and standard deviation for time and test accuracy is calculated. Results show that the SGD with freezing now have slightly higher accuracy.

Network Name	**Comparison Between SGD and Our Method**			
	SGD Time (hrs)	***SGD Test Accuracy***	***Ours Time (hrs)***	***Ours Test Accuracy***
ResNet-50	5.40±0.13	95.35±0.17	4.96±0.11 (-8.2%)	95.45±0.11

3.3.3 DenseNets

For DenseNet, we also use the same freezing frequency hyperparameter as VGG nets and ResNets. DenseNet-121 experients were run on a single NVIDIA Tesla P100 GPU. The results for DenseNet-121 can be found in Table 3.1. Experiments were repeated three times for both the vanilla SGD and SGD with freezing. A 12.3% time reduction can be achieved.

3.4 Summary and Future Work

In this work, we propose a method to intelligently freeze deep learning layers to reduce training time. A formula to calculate the normalized difference of gradients is designed to

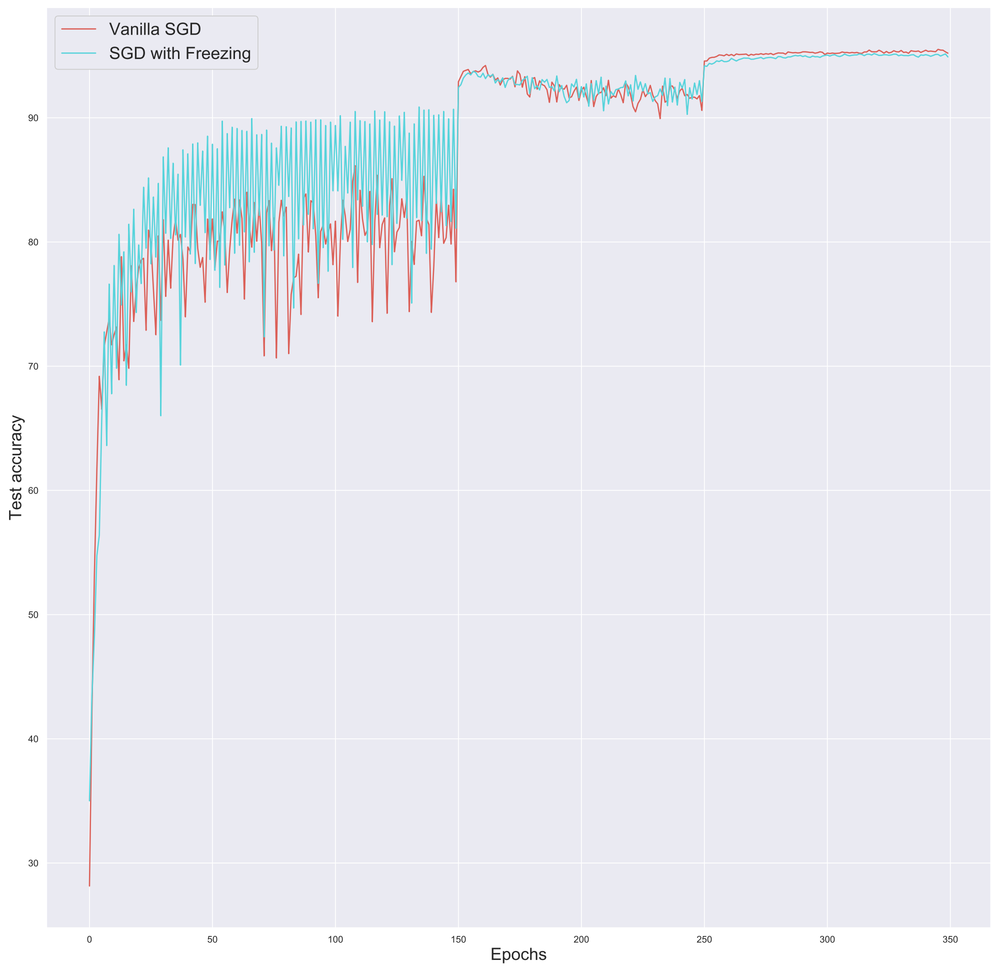


Figure (3.5) We train ResNet-50 using SGD with freezing and vanilla SGD. The test accuracy for SGD with freezing outperforms or is similar to vanilla SGD in the first two learning stages (when the learning rate is 0.1 and 0.01). However, the accuracy is slightly lower when it comes to the last stage (the learning rate is 0.001). This suggests that there might be too much freezing in the second learning stage.

calculate the freezing rates for all layers with weights. And the maximum subarray sum of the freezing rates array is used to decide how many layers starting from the beginning should be frozen. Experiments on VGG nets, ResNets, and DenseNets show that our method can speed up the training process while having similar test accuracy.

There are still many different freezing schemes that remain to be explored, and there is room for further time reduction. For example, instead of using freezing frequency as a hyperparameter, a freezing probability can be calculated based on freezing rates. Also, instead of freezing layers, it would be interesting to see how the method performs on individual network parameters.

PART 4

INCORPORATING PRIOR KNOWLEDGE IN DEEP LEARNING

4.1 Background

4.1.1 Classification Accuracy Improvement Methods

There are various ways of improving classification accuracy: tune model hyperparameters [61][90][97], design novel architectures [92][99], find more data to train the model, data augmentation [67], add more features [100], transfer learning [85], model ensemble [63], and so on.

Model hyperparameters play an important role in model performances. The size of the model decides how well the model can fit the data. Learning rates need to be tuned carefully to help the model converge to a better optimum [7]. The convolution window decides how much local information is examined at a time. And there are many more hyperparameters that are essential to models' performance. Deciding a good set of hyperparameters is difficult work, and there are many research works in this area [4][5][9][61][74][90][97].

Deep learning models also rely on the abundance of data. And data augmentation, a method that manufactures data with existing data, can greatly help with model performance. Popular image augmentation methods include flipping, translating, and rotating images. And there are novel image augmentation methods such as overlaying two images [40] and masking part of the image [77].

A model ensemble takes advantage of multiple diverse models and combines the predictions of various models on the task. It is a very powerful technique. In Kaggle competition, it is common to see the top results utilizing ensemble learning [63]. Popular model regularization technique, dropout [72], also behaves like training an ensemble of submodels.

4.1.2 Knowledge incorporation in deep learning

Deep learning is mostly data-driven. We want to investigate ways of incorporating human knowledge into the training process and observe how that can help with the model. Adding information/features could also improve model performances. Sometimes the information does not come from the data itself but from other sources.

There are various ways of incorporating prior knowledge into deep learning. Our previous work [42] in infant cry classification uses prior knowledge to select the categories in corresponding models to improve performance. Our previous work [42] in infant cry classification uses prior knowledge to select the categories in corresponding models to improve performance. Diligenti et al. [21] used first-order logic rules and translated them to constraints, which are incorporated during the backpropagation process. Towell et al. [79] proposed a hybrid training method: first translate logic rules into a neural network and then use the neural network to train classified examples. Xu et al. [91] derived a semantic loss function that bridges deep learning outputs and logic constraints. Hu et al. [35] proposed a method that iteratively distills information in logic rules into weights of neural networks. Ding et al. [22] integrated prior knowledge in indoor object recognition problems. Color knowledge for indoor objects and object appearance frequencies are generated in the form of vectors and used to modify deep learning model outputs. Jiang et al. [43] incorporated semantic correlations between objects in a scene for object detection. Stewart et al. [73] performed supervised learning to detect objects using domain knowledge instead of data labels. This can be applied to problems where labels are scarce.

Lee et al. proposed Dropmax [51], a method that randomly drops classes adaptively, to improve the accuracy of deep learning models. Learning from this idea, we investigated the effect of incorporating prior knowledge into classification problems by reducing the problem to a subset of classes. This was a preliminary study we did on knowledge incorporation. Later on, we studied similarity knowledge among classes and incorporated that into CNNs using a graph convolutional layer.

One kind of knowledge that can be incorporated is class similarity. In many works, label relations are incorporated into deep learning models. Word taxonomy can be used to improve image object recognition [38]. Associated image captions can improve entry-level labels of visual objects. Structured inferences can be made by incorporating label relations. GCN augmented neural network classifiers can be used to incorporate underlying label structures.

4.1.3 Class Similarity

There are various ways to obtain similarity between classes. Label relations can be used to define class similarities [20], where semantic similarity of labels can be computed. The problem is in many situations, label relations may not be able to fully capture the similarities between the actual data. Sometimes label relations cannot represent meaningful class similarities at all.

Class similarity knowledge does not necessarily need to come from external sources. It may be extracted from the data itself. Arino et al. [2] proposed to use misclassification ratios of trained deep neural networks to get class similarities. Their proposed method uses symmetrical similarity between classes.

4.1.4 Graph Convolutional Networks

CNNs are successful in capturing the inner patterns of Euclidean data. However, lots of data in real life scenarios exist in the form of graphs. For example, social networks are graph-based: the nodes are people, and the edges are the connections between them. Chemical molecules, atoms held together by chemical bonds, can naturally be modeled as graphs. Analyzing their graphical structure can determine their chemical properties. In traffic networks, points of intersections are linked together by roads, and we can predict the traffic of these intersections in future times. Images can be thought of as a special kind of graph, where adjacent pixels are connected together, forming a pixel grid. When images are fed through a CNN model, the nearby pixels are being convoluted, and local spatial information is retained. CNNs cannot learn from graph data with more complex relations. Similarly, graph convolution can be performed on graph data, where each node can learn the weighted average of its neighbors' information. A GCN [46] does the following graph convolution operation:

$$H^{(l+1)} = \sigma(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}) \tag{4.1}$$

Where $\tilde{A}$ is the adjacency matrix with self-loops, $\tilde{D}$ is the node degree matrix, W is a layer-specific trainable weight matrix and sigma is an activation function. $H^{(l)}$ is the output from the lth layer and $H^{(0)}$ is input. During graph convolution, each node aggregates information from its neighbors.

4.2 Preliminary Studies

In our preliminary study, we are inspired by the fact that classification problems are easier when there are less classes to predict. We want to use knowledge to reduce the complexity of a classification problem by excluding some unlikely classes and only make predictions on a subset of more likely classes.

There are different forms of knowledge. In this work, we calculate the similarity between the data and class templates to decide which classes are the more likely ones to be included in the prediction. For shape recognition, we use the Fourier descriptor to obtain the shape signatures for images, and then distances are calculated based on Fourier descriptors. Fourier descriptor is chosen because it is rotational, translational, and scale invariant.

Using Fourier descriptor distances, we can decide on which subset of classes to do classification on. First, we create templates for all classes and obtain Fourier descriptors for all of them. To do classification for a candidate image, we first calculate its Fourier descriptor and compare it with those of all the templates. This will give distance scores that tell us how similar the candidate images are to the templates. If the distances to some templates are far, it is unlikely that the candidate image belongs to that class. We can use this information in the neural network classification process. Instead of making predictions on all classes, we can just do so on a subset of classes. The Fourier descriptor distance scores give us information on which subset of classes we should make our prediction on. While the

Fourier descriptor distances may not tell us which class is the correct prediction directly, it gives us some idea on which classes the candidate is more similar to. And the neural network will then do the classification for us. In summary, compared with the original classification, the prior knowledge helps us to eliminate some unlikely classes for the candidate image, making the task easier.

4.2.1 Fourier Descriptor

Fourier descriptor is used a lot in shape recognition. They can encode shape signatures of an image. Fourier Descriptor is a preferred method to capture shape information because it is translational, scale, and rotational invariant.

Here is how a Fourier descriptor of an image is obtained:

- Apply threshold and convert the image into binary.
- Find contour based on the binary image.
- Find the coordinates of all the pixels on the boundary.
- Convert the coordinates into complex numbers.
- Take discrete Fourier Transform of the complex-valued vector

4.2.2 Matching with Templates

We match the candidate images with templates to get distance scores of an image to each class. This knowledge will give us ideas on which subset of classes to make prediction on.

For an initial attempt, we decided to use printed digits as templates for our dataset. We consider printed digits to be like concepts for hand-written digits. Handwritten digits are like variations of their concepts. For all images in the dataset and in the templates, we first use Fourier descriptor to obtain their shape signature. The Fourier descriptors are used for distance calculation between images. Then for each digit in the MNIST dataset, we calculate its distance with the 10 templates. Each digit will have a matching score with all ten templates, which is a vector of length 10. The matching score informs us which templates the digit is closer to and which templates it is further away from.

We evaluated how good are the templates when using one set of printed digits. As shown in Figure 4.1, we calculate the distance between the candidate image and all 10 templates using the Fourier descriptors. The smaller the score, the more similar the two images. The images with green scores in Figure 4.1 are 5 templates that the candidate image is more similar to based on the Fourier descriptors. And the ones with red scores are less similar to the candidate image. If the target class is within the top k most similar templates, then we consider the templates to be good for the candidate image. We calculate the distance scores between all images in the training dataset and the templates. And the above-mentioned criteria is used to judge whether the templates are good for the candidate images. We evaluate the quality of the current template and list the results in Table 4.1. From Table 4.1, we can tell how much data the target class is within the included class for different k values. For example, when $k = 5$, Only about 85% of the MNIST training set satisfies the criteria, i.e., for about 15% of the handwritten digits, the target class is among the templates that are less similar. This initial attempt does not give us good enough knowledge.

Using printed digits is the first thought when it comes to choosing templates. There are ways of improving the templates, such as use multiple templates for a single class and aggregate the distance scores. One idea of choosing additional templates is to find templates from the training dataset itself. For each class in the training set, we can do clustering to find representative images and add those to the templates.

4.2.3 Incorporate knowledge into Training

We know that the classification problem is easier when there are fewer classes to predict. The idea is to use prior knowledge to reduce the number of classes in classification.

We use prior knowledge to generate a knowledge mask that reduces the original classi-

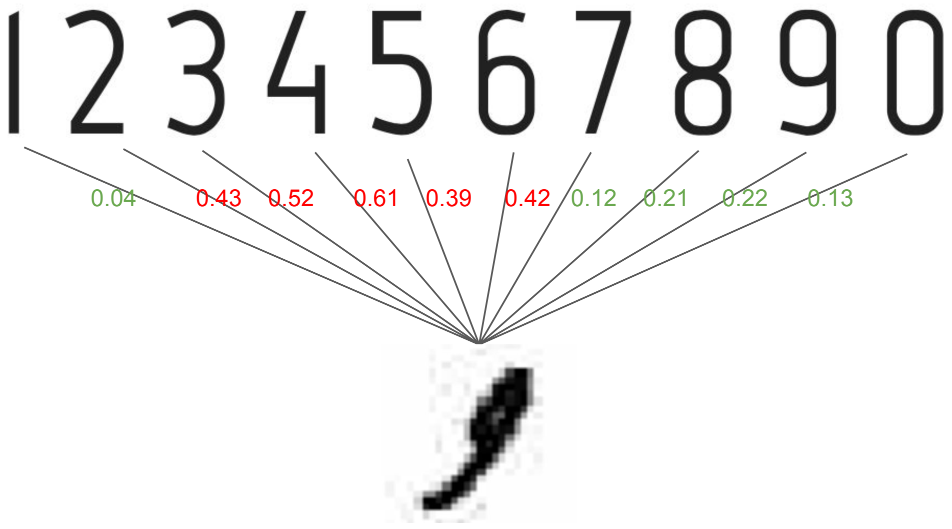


Figure (4.1) Fourier descriptor distance between the candidate image and the templates.

fication to a subset of classes. The knowledge mask tells which classes to be included and which not for each image in the dataset. It is added into the deep learning model after the softmax activation. Figure 4.2(a) shows the original softmax classification. Figure 4.2(b) is a knowledge mask applied to the original classification. The knowledge mask contains 0's and 1's: 0 means the corresponding class is unlikely to be the target, and we exclude it in classification; 1 means the class is likely to be the target class, and we include it in classification. Equation 4.2 shows how we get the network output using the knowledge mask. In Equation 4.2, $\vec{O}$ is the output generated by softmax, and $\vec{M}$ is the knowledge mask which contains 0's and 1's. Operator $\odot$ means element-wise multiplication. For example if $\vec{a} = \{a_1, ..., a_i, ...a_N\}$ and $\vec{b} = \{b_1, ..., b_i, ...b_N\}$, then $\vec{a} \odot \vec{b} = \{a_1 b_1, ..., a_i b_i, ...a_N b_N\}$. $\vec{K}$ is the results produced by the knowledge mask. It contains some zeros, which mean the corresponding classes are excluded, and no gradients are backpropagated. Only more probable classes produce non-zero probabilities.

$$\vec{K} = \frac{\vec{O} \odot \vec{M}}{\sum_i O_i M_i} \tag{4.2}$$

Algorithm 2 shows the overall procedure of how to incorporate the extracted knowledge into the training process. Note that for now knowledge is only incorporated into the training process because we do not want the target class to be excluded during testing time. If we obtain good enough templates, we will include knowledge in the testing phase and see how it affects the results.

Algorithm 2: Apply Knowledge Mask to Reduce Number of Classes in Classification

```
$k$ is the number of classes to keep in classification.
begin
    $distances$ = get_distance($data$, $templates$)
    train($distances$):
        $softmax\_output$ = model($train\_data$)
        $mask$ = get_knowledge_mask($distances$)
        $mask$ = include_targets($mask$, $train\_targets$)
        $output$ = mask_output($softmax\_output$, $mask$)
        $loss$ = cross_entropy($output$, $train\_targets$)
    test():
        $softmax\_output$ = model($test\_data$)
        $loss$ = cross_entropy($softmax\_output$, $test\_targets$)
end
```

We use two ways of excluding classes: deterministically and stochastically. This is achieved by generating deterministic and stochastic knowledge masks respectively.

- Exclude classes deterministically

  After the distances to the templates are calculated, we can directly use the ranking of distance to decide which classes to keep during training. The k classes that are most similar to the candidate image will be kept during training. For example in Figure 4.3, k is 5 and 1, 7, 8, 9, 0 are the five classes for the image to be classified. During training, this classification of this specific image will always involve these five classes only.

- Exclude classes stochastically

Instead of excluding classes deterministically using the ranking of the distances scores, we calculate the probability that each class will be excluded. For each candidate image, the probability of the classes being excluded is defined in Equation 4.3:

$$\vec{p_i} = \frac{\lambda \vec{d_i}}{max\{\vec{d_i}\}} \odot (\vec{e} - \vec{t_i}) \tag{4.3}$$

where i is the index for the candidate image, $\vec{d_i}$ is the vector of distances of image i to all templates, $max\{\vec{d_i}\}$ returns the maximum number in $\vec{d_i}$, λ is the scale factor, $\vec{t_i}$ is the one-hot encoded vector of the targets for image i, $\vec{e}$ is a vector of the same length as $\vec{t_i}$ and only contains 1's, and $\odot$ is the element-wise multiplication operator. Figure 4.4 shows an example of excluding probabilities for a candidate image. Note that the exclusion probability of the target class is 0 because we do not want to exclude the correct answer.

Based on the exclusion probability, at each iteration, the class dropped is different for a specific image, as shown in Figure 4.5. The classes that have a larger distance from the candidate image have a higher probability of being excluded, and similar classes are more likely to be kept. Excluding the classes stochastically has a regularization effect on the model similar to dropout.

Table (4.1) We evaluate the templates quality using the percentage of images for which the target class is within the included classes.

k **Value**	**Templates Quality**
5	85.0%
6	88.4%
7	90.9%
8	94.5%
9	98.7%

4.3 Preliminary Results

We perform experiments on the standard MNIST dataset. As an initial attempt, we use 1,000 images for training and 1,000 images for testing. We first perform experiments using the deterministic class excluding technique, and then the stochastic one.

4.3.1 Exclude classes deterministically

We perform deterministic class exclusion using different k values, where k is the number of classes being kept. For each setting we repeat the experiment five times and report the average and standard deviation in Table 4.2. Compared with the basic softmax activation, deterministic exclusion, in this case, does not give any accuracy improvements, instead, accuracies are lower. The fewer number of classes are kept, the lower the accuracies are. The reason for the accuracy drop could be: templates are not good enough. Or training is given a simpler task, but testing is more difficult. Thus testing has a lower score. We will improve the quality of templates to see how that affects the results.

Table (4.2) Comparison between basic softmax and deterministic exclusion.

	Test Accuracy
Softmax	94.98±0.11
Knowledge Mask k=5	92.66±0.67
Knowledge Mask k=6	93.06±0.38
Knowledge Mask k=7	93.94±0.47
Knowledge Mask k=8	94.52±0.47
Knowledge Mask k=9	94.98±0.43

4.3.2 Exclude classes stochastically

We perform stochastic class exclusion using different scale factors. The class exclusion probabilities for each image is calculated based on Equation 4.3. For each setting, we repeat the experiment five times and report the average and standard deviation in Table 4.3. We observe test accuracy improvements when stochastic class exclusion is used. Different scale factor gives slightly different accuracies. However, at this point, we are not sure how much effect does the scale factor has on the accuracies. We will need to do use statistical methods to evaluate whether there are significant differences in the mean accuracy when different scale factors are used.

Table (4.3) Test accuracy of stochastic exclusion with various scale factors

Scale Factor	**Test Accuracy**
0.8	95.44±0.30
0.9	95.32±0.36
0.95	95.52±0.19
0.99	95.50±0.12

4.4 Incorporate Class Similarity Knowledge

From the preliminary study, we learned that using human predefined knowledge may not work well. Next, we explore a data-driven way of defining knowledge.

In deep learning multiclass classification tasks, data are mapped to one of the predefined classes. The model extracts features from data, and no inter-class relationship is considered. Prior works have shown that incorporating class similarity knowledge can improve classification performances [2][51]. We incorporate class similarity knowledge into deep learning models using graph convolution to improve classification accuracy.

The class similarity knowledge is extracted directly through training from data, and no additional external information is required. Information directly learned from data should represent similarity more accurately than external ones. We train the model on the training dataset and obtain the misclassification graph on the validation dataset. The misclassification graph contains information about how often one class is misclassified as another class. If data in one class is frequently misclassified as another class, we consider that the former class is similar to the latter class.

The vector representations of classes and data are extracted from learned model weights and hidden layer outputs, respectively. Together with the class similarity graph, class and data representations are sent to a graph convolution layer. The graph convolution adjusts the results according to class similarity knowledge, and class scores are finally sent to the Softmax activation function to get the final classification results.

4.4.1 Represent Class Similarities Using Misclassification Graph

We use misclassification graphs to represent class similarities. If data in one class is often misclassified as another class, we consider the two classes have high similarity. Our misclassification graph is directed, which means similarities can be directional. Class A can be very similar to Class B, but not the other way around. This directional similarity can be observed from the misclassification information. In experiments, we can observe one class being frequently misclassified as another class, but not the reverse way.

The misclassification graph is built based on a trained model's performance on the validation dataset. Figure 4.6 is an example of a misclassification graph. A CNN model is trained on a downsampled MNIST dataset. After the model is trained, we evaluate its performance on the validation dataset. Mistakes made on the validation dataset are recorded and plotted as a graph. Each node represents a class in the dataset, and edges denote how frequent data from one class are being misclassified as another class. The thicker the edge is, the more misclassification between the two classes. As shown in Figure 4.6, there are 10 classes in the MNIST dataset: 0 - 9. Edges between some classes are thicker, for example Class 8 to Class 1. This means lots of images that are actually 8's are mistaken as 1's. Note that edges have directions. While 8's are easily mistaken as 1's, barely any 1's are misclassified as 8's.

4.4.2 Overall Model Architecture

Figure 4.7 shows the overall architecture of our model. There are two stages of training. In Stage 1, we train an original CNN model without the graph convolution layer. In the illustration, we use a CNN model with two convolutional layers and a fully connected layer as an example. The model is trained for enough epochs such that it has learned the training data well. Then a misclassification graph is obtained by feeding the validation data through the model. Data and class embeddings are also extracted from the CNN model to produce vector representations for graph nodes. After we obtain the misclassification graph and node representations, we can enter Stage 2 of training. In this stage, a graph convolution layer is added after the fully connected layer. The graph convolution contains misclassification information. This layer takes in the latent data representation and class embedding information and does convolution among the classes, and outputs new data and class embeddings with aggregated neighborhood information. The new data and class embeddings are further used to calculate class scores and finally sent to the softmax activation function to produce final results.

The graph convolution layer takes in data and class embeddings and outputs new vectors that contain aggregated neighborhood information. Figure 4.8 shows a five-class graph convolution example. The nodes in the graph represent individual classes. There are five nodes in this example corresponding to five classes. The edges between nodes represent how often one class is misclassified as another class. Edges have directions. Nodes are represented using vectors of numbers. In our case, the vectors are concatenations of data embedding and class embeddings.

Both data embedding and class embedding can be obtained from the original CNN model. Data embeddings are obtained by getting the outputs from the layer right before the Softmax classifier, as shown in Figure 4.9. Class embeddings are obtained from the weights connecting the classifier and the layer before. Figure 4.10 shows how the class embeddings are obtained. Notice that the data and class embeddings have the same dimensions. And in the original CNN model, the inner products of data and class embeddings produce class sores.

Data embedding and class embedding together form the graph node vector representations. After the node vector representations pass through the graph convolution layer, the graph convoluted outputs are further transformed to get class scores before sent to the Softmax activation function. Vector representations for data and classes are extracted from the original CNN model. The data embedding for data d is $\vec{d} = \{d_1, ...d_k, ...d_n\}$. The class embedding for class i is $\vec{c_i} = \{c_{i1}, ...c_{ik}, ...c_{in}\}$. In the original CNN model, the dot product of $\vec{d}$ and $\vec{c_i}$ produces the class i score of d. The vector representation for node i is the concatenation of $\vec{d}$ and $\vec{c_i}$: $\vec{x_i} = \{d_1, ...d_n, ...c_{i1}, ...c_{in}\}$. After graph convolution, the output for node i is: $\vec{h_i} = \{h_1, ...h_n, h_{n+1}, ...h_{2n}\}$. We need to transform the outputs from graph convolution to class scores before feeding them to softmax activation function.

We use two ways of transforming graph convoluted outputs to class scores. This will give us two variants of the graph convolution assisted model: PK-GCN-1 and PK-GCN-2. Figure 4.11 describes the difference between these two variants. In PK-GCN-1, the outputs of the graph convolution layer are directly used for producing class scores. In PK-GCN-2, a fully connected layer is added after the graph convolution layer. The fully connected layer merges the inputs and outputs of the graph convolution layer. The input to the fully connected layer is the input to the graph convolution layer concatenated with the output of the graph convolution layer.

In PK-GCN-1, we produce class score s_i for class i according to the following formula:

$$s_i = \sum_{k=1}^{k=n} d_k h_{i(n+k)} + \sum_{k=1}^{k=n} c_{ik} h_{ik} \tag{4.4}$$

where $\vec{c_i}$ is the class representation vector for node i, $\vec{d}$ is the data representation vector and $\vec{h_i}$ is the output of the graph convolution layer for node i.

For PK-GCN-2, we add a fully connected layer after the graph convolution layer. This layer merges the original data and class embeddings, and the graph convoluted output together. Let the dimension of the output of the fully connected layer be $2l$. We use the following formula to produce the class score s_i for class i:

$$s_i = \sum_{k=1}^{k=l} q_k q_{k+l} \tag{4.5}$$

where the output of the fully connected layer is $\vec{q}$, and its dimension is $2l$.

Our method can be summarized into the following steps:

1. Train a base CNN model until convergence. This is stage 1 training.
2. Feed the validation dataset through the CNN model to get the misclassification graph. The graph is weighted and bidirectional.
3. Extract vector representation of each of the m classes $\vec{c_1}$, $\vec{c_2}$, ... $\vec{c_m}$ from trained model weights. (Figure 4.10)
4. Obtain vector representations of data d from the last hidden layer outputs. (Figure 4.9)
5. Add a graph convolution layer to the base CNN model. Node i in the graph is represented by $\vec{x_i}$ which is data representation (from step 4) concatenated with class i representation (from step 5). $\vec{x_i} = \{\vec{d}, \vec{c_i}\}$.
6. (PK-GCN-2 Only) Add a fully connected layer after the convolution layer.
7. (PK-GCN-1 Only) Produce class scores using Equation 4.4.
8. (PK-GCN-2 Only) Produce class scores using Equation 4.5.
9. Continue training the model with the new layers until convergence. This is stage 2 training.

4.5 Experiments and Results

We perform our experiments on the MNIST dataset and CIFAR-10. In our experiment, different CNN model sizes and different data sizes are experimented with to evaluate how adding class similarity knowledge helps in different circumstances.

4.5.1 Datasets

The MNIST dataset contains 10 classes of handwritten digits: 0-9. The dataset provides a train-test split, with 60,000 training and 10,000 testing. The CIFAR-10 [48] dataset contains 60,000 color images of size 32X32 in 10 classes: each class has 6,000 images. There are 50,000 images for training and 10,000 images for testing.

4.5.2 Baseline

We use a two-stage training method for our models. In stage one training, a base CNN model is used to obtain the misclassification graph for class similarity. In stage two, a graph convolution layer is added to incorporate class similarity knowledge. The baselines for our method are the base CNN models we use in stage one training. We make sure the baseline and our proposed model are trained for the same number of epochs using the same optimizer.

4.5.3 Results on MNIST Dataset

We evaluate our method on the MNIST dataset. We experimented with two CNN base models. The first contains 1 convolution layer; the second contains two convolution layers. We also experiment with different train and validation data sizes. The validation dataset is obtained from the original MNIST training set. We make sure that the training and validation dataset has a balanced number of data from each class. We report the accuracy of the baseline and our models on the test dataset, which contains 10,000 images.

Table 4.4 shows the comparison of the results between our model and the original CNN. When using base model 1, the original CNN model was trained for 200 epochs. PK-GCN-1 and PK-GCN-2 were trained for 40 epochs in the first training stage and 160 epochs in the second. When we use base model 2, the original CNN model was trained for 200 epochs. PK-GCN-1 and PK-GCN-2 were trained for 80 epochs in the first training stage and 120 epochs in the second. All training uses AdaDelta optimizer with the same setup. And all models were trained for the same number of epochs for a fair comparison.

From the table, we can see that our model outperforms the base model by as much as

1.56%. And generally, the improvements are bigger when the amount of available training data is smaller.

4.5.4 Results on CIFAR-10 Dataset

We evaluate our method on the CIFAR-10 dataset. The base model we use is VGG-11. We experiment with various data sizes, and the models are evaluated on a test dataset with 10,000 images. Table 4.5 shows the comparison of the results between our model and the original CNN on CIFAR-10. The original CNN model is trained for 300 epochs. PK-GCN-1 and PK-GCN-2 were trained for 100 epochs in the first training stage and 200 epochs in the second. All training uses AdaDelta optimizer with the same setup.

When using VGG-11 as the base model, we can see that our model outperforms the baseline by as much as 2.55%.

4.6 Summary and Future Work

In our work, we define the similarity between classes using the misclassification graph produced on the validation dataset and use a graph convolution layer to incorporate that information into training. Experiment results on benchmark image classification datasets show that incorporating class similarity knowledge can improve multi-class classification accuracy, especially when the amount of available data is small.

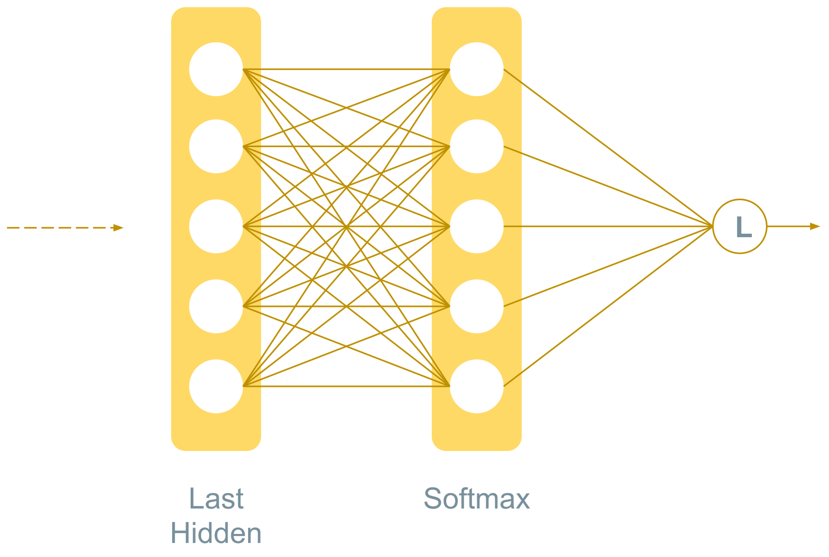
Instead of obtaining the misclassification graph from validation data, rules can be used to define class relations. The relationships between classes are fuzzy. In the future, we plan to incorporate fuzzy logic [6][34][53] and rough set theories [106][107][108] into our work to define class relations. A graph attention [81] layer can also be used in place of the graph convolution layer. The advantage of graph attention is that we do not need to know the edge information in the graph. The edges are learned through training. We could potentially study the edges learned by graph attention to see if they correlate to class similarities.

Table (4.4) Accuracy comparison on MNIST of the original CNN model and PK-GCN models with various data sizes. 'X' Means no accuracy improvement is observed.

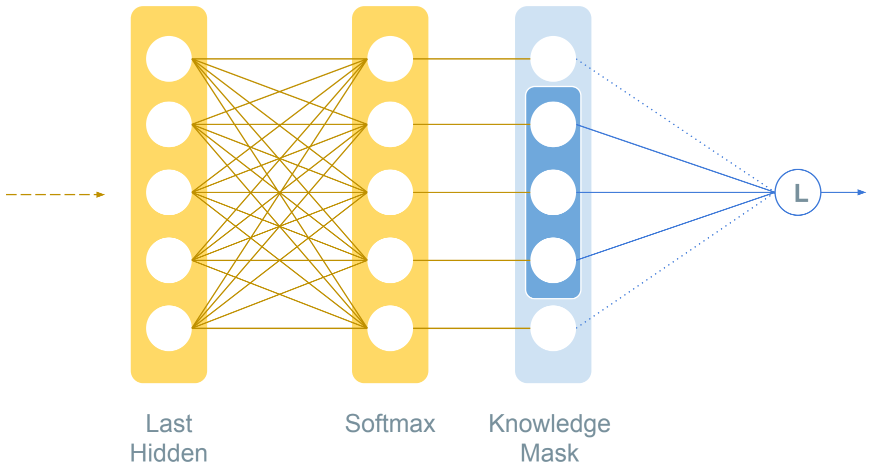
	Data size (Train|Validation)	**300| 300**	**500| 500**	**1,000| 1,000**	**1,500| 1,500**
Base model 1: 1 conv layer	Original CNN	85.30%	88.78%	93.07%	94.42%
	PK-GCN-1	85.74% (+0.44)	89.49% (+0.71)	93.97% (+0.9)	94.90% (+0.48)
	PK-GCN-2	86.28% (+0.98)	89.51% (+0.73)	94.12% (+1.05)	94.94% (+0.53))
	Data size (Train|Validation)	**2,000| 2,000**	**2,500| 2,500**	**3,000| 3,000**	
Base model 1: 1 conv layer	Original CNN	95.13%	95.78%	96.26%	
	PK-GCN-1	95.62% (+0.49)	96.03% (+0.25)	96.32% (+0.06)	
	PK-GCN-2 .	95.66% (+0.53)	95.98% (+0.2)	96.50% (+0.24)	
	Data size (Train|Validation)	**300| 300**	**500| 500**	**1,000| 1,000**	**1,500| 1,500**
Base model 2 2 conv layers	Original CNN	89.67%	92.08%	95.43%	96.00%
	PK-GCN-1	91.16% (+1.49)	93.46% (+1.38)	x	96.30% (+0.30)
	PK-GCN-2	91.23% (+1.56)	93.19% (+1.11)	x	96.26% (+0.26)
	Data size (Train|Validation)	**2,000| 2,000**	**2,500| 2,500**	**3,000| 3,000**	
Base model 2 2 conv layers	Original CNN	97.00%	97.44%	97.70%	
	PK-GCN-1	97.01% (+0.01)	97.46% (+0.02)	97.74% (+0.04)	
	PK-GCN-2	x	x	97.80% (+0.10)	

Table (4.5) Test accuracy comparison on CIFAR-10 of the original CNN model and PK-GCN models with various data sizes.

	Data size (Train|Validation)	**300| 300**	**500| 500**	**1,000| 1,000**	**1,500| 1,500**
Base model: VGG-11	Original CNN	33.34%	35.51%	44.33%	50.09%
	PK-GCN-1	35.37% (+2.03)	36.01% (+0.50)	46.88% (+2.55)	51.16% (+1.07)
	PK-GCN-2	34.93% (+1.59)	37.45% (+1.94)	45.72% (+2.39)	51.21% (+1.12)
	Data size (Train|Validation)	**2,000| 2,000**	**2,500| 2,500**	**3,000| 3,000**	
Base model: VGG-11	Original CNN	52.81%	56.50%	60.56%	
	PK-GCN-1	53.69% (+0.88)	56.98% (+0.48)	61.04% (+0.48)	
	PK-GCN-2	53.25% (+0.44)	56.54% (+0.04)	60.82% (+0.26)	



(a)



(b)

Figure (4.2) The original softmax versus softmax with knowledge mask.

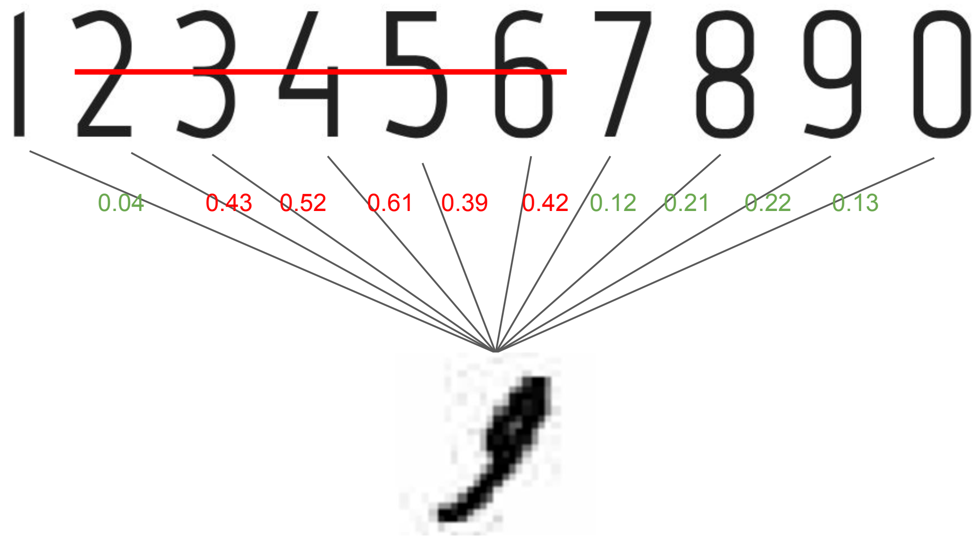


Figure (4.3) Excluding classes deterministically using k=5.

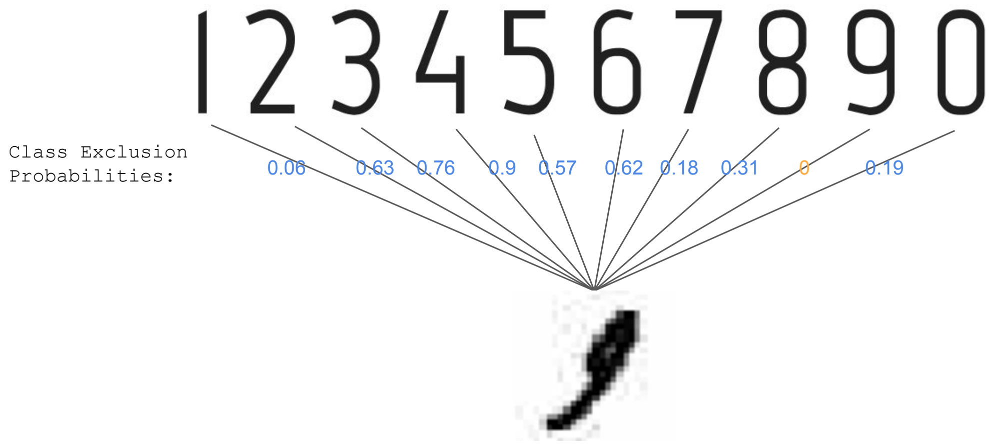


Figure (4.4) Class exclusion probabilities.

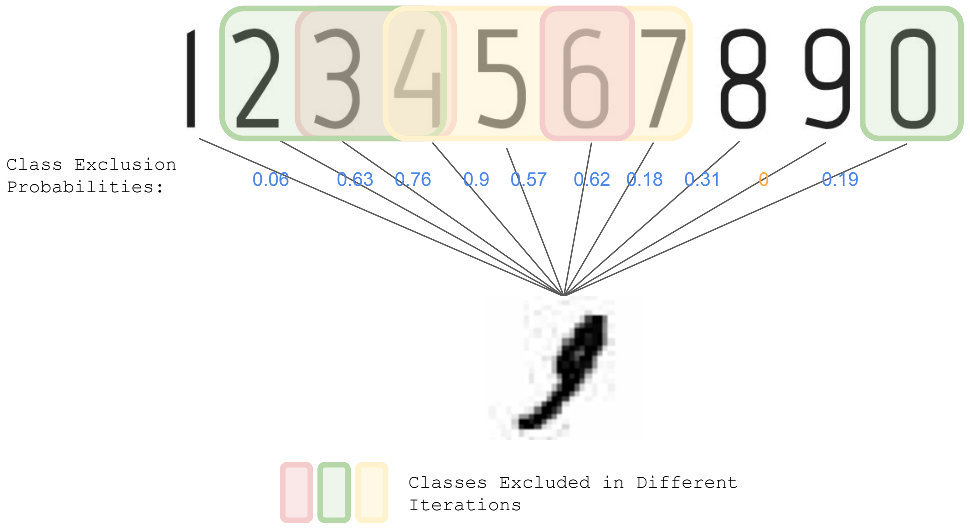

1 2 3 4 5 6 7 8 9 0
Class Exclusion Probabilities:
0.06
0.63
0.76
0.9
0.57
0.62
0.18
0.31
0
0.19
Classes Excluded in Different Iterations

Figure (4.5) Classes are excluded stochastically based on exclusion probabilities.

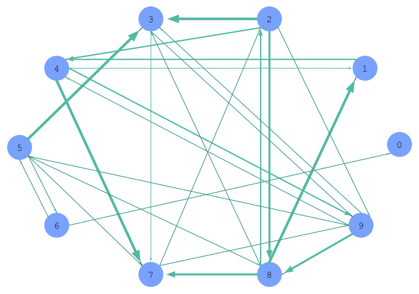


Figure (4.6) A Misclassification Graph example on the MNIST Dataset.

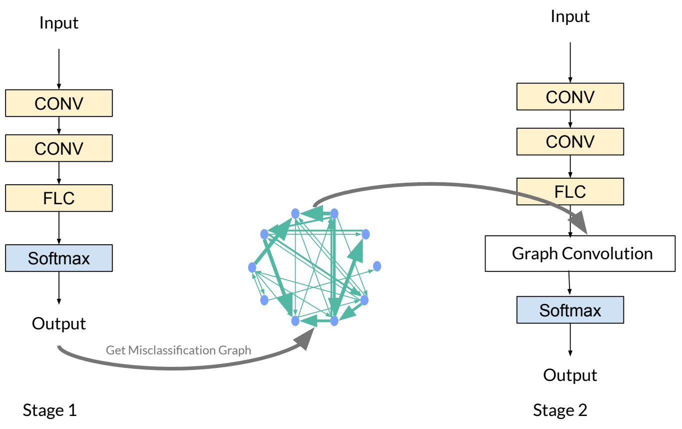


Figure (4.7) Overall model architecture. In Stage 1 training, mislassification graph is obtained from the the validation data. The graph represents class similarity information and is fed into the next stage. In Stage 2 training, a graph convolution layer is added to incorporate class similarity knowledge into the training.

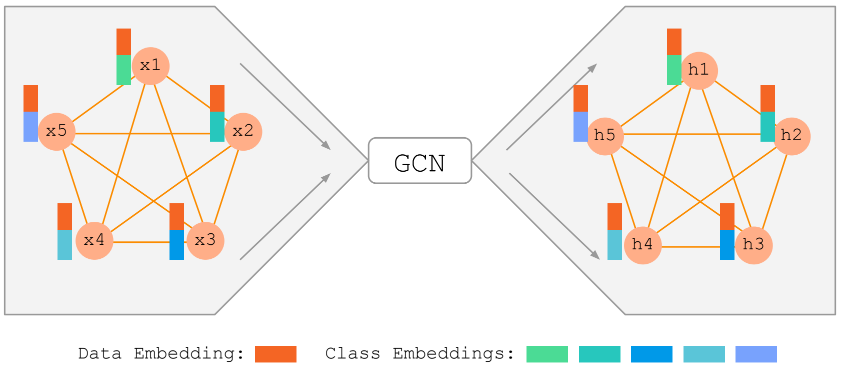


Figure (4.8) The GCN takes in the data embedding concatenated with class embeddings, performs neighborhood aggregation, and outputs new node representations.

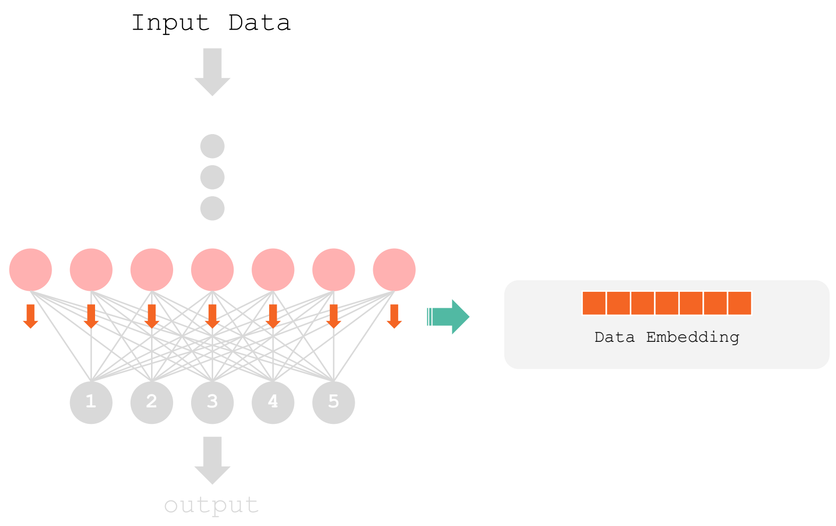


Figure (4.9) Obtain data embedding from the model.

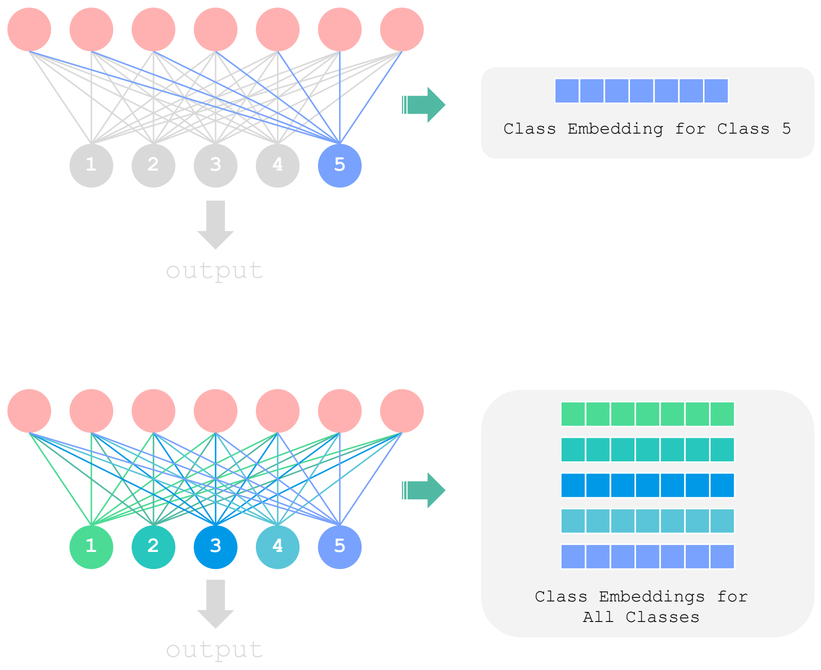


Figure (4.10) Obtain class embeddings for all classes from the model.

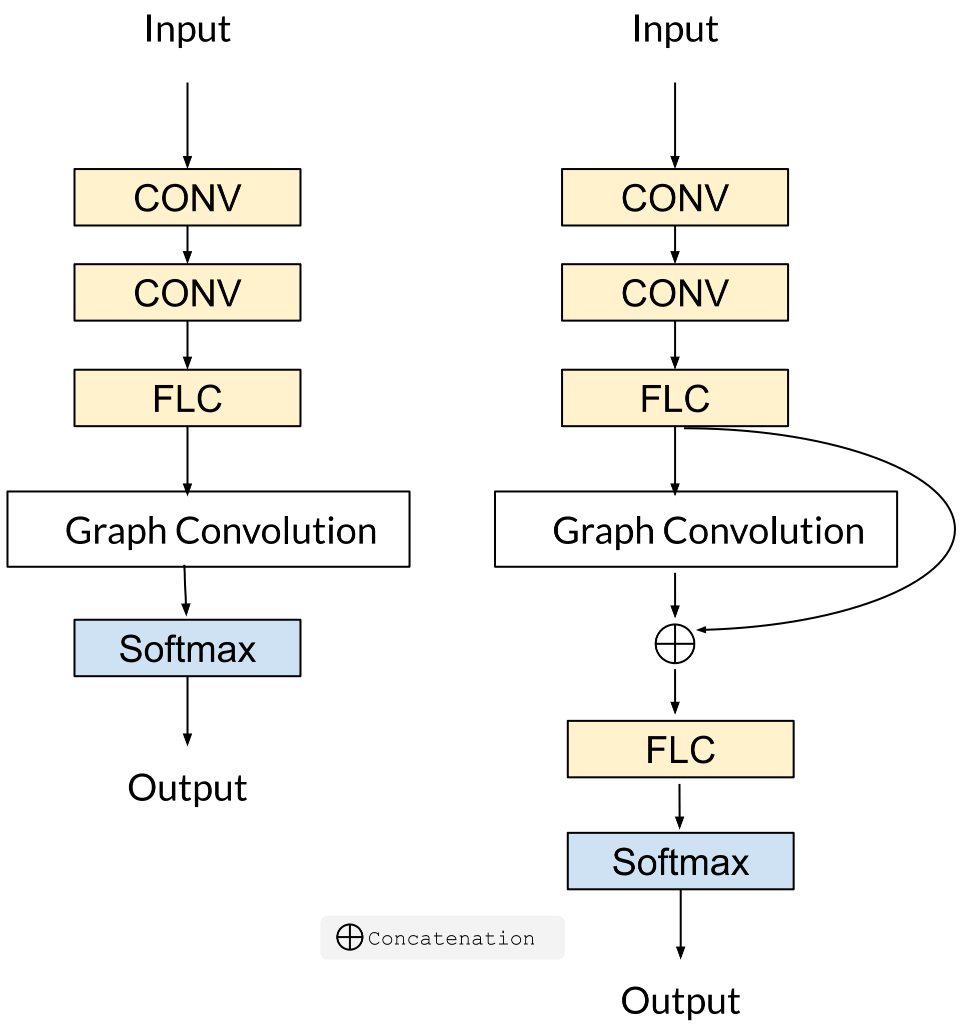


Figure (4.11) Two ways of incorporating the graph convolution layer.

PART 5

MAPPING NON-IMAGE DATA TO IMAGES FOR FEATURE RELATIONSHIP LEARNING

5.1 Backrgound

EHRs provide a great amount of digital information that can be used for applications such as health analytics and clinical informatics [78][87]. Various machine learning and deep learning models have been used on EHR data to make disease detection [17][18][52][98], disease progression [12][103], patient subtyping [8], and so on.

EHRs contain information such as patient demographics: age, gender, ethnicity, health care providers, and so on. They also contain patient journeys: a sequence of diagnosis and treatment information over time.

Traditionally, the information in an EHR is treated as a bag of features. No relationship or structural information among features is considered.

Recent works show that there exists structural information among EHR features, and incorporating this information can help with improving the prediction performance. Choi et al. [15] proposed graph convolutional transformers that learn the hidden structure of EHR while performing supervised training. Choi et al. have also proposed MiME [16], a model architecture that reflects relationships between diagnosis and treatments. Wu et al. [86] studied the relationship between patients and clinicians because patients with the same disease that visit the same clinician tend to receive similar treatments. In most EHRs, structural information among features does not already exist. Researchers reconstruct this information using various methods. For example, the relationship between a drug and a disease can be represented by how often they appear in the same medical claim. Zeng et al. [102] measure drug-drug similarity by calculating the Jaccard similarity coefficient on EHR data.

Deep CNN is a popular class of deep learning models, and they are mostly used for images. Images contain pixels that are spatially coherent. Neighboring pixels share similar information. Convolution filters in a CNN model do convolution operations and extract features among neighboring pixels. Sharma et al. [66] have transformed non-image data such as gene expressions and test data to images and used CNNs for extracting features and subsequent prediction tasks. When mapping non-image data to images, the positions of the pixels are important. If they are arbitrarily arranged, it can harm the feature extraction and prediction performances.

Inspired by previous works, we propose to map EHR features to images using relational and structural information among the features. This way, the feature relationships are taken into consideration as related features are positioned as neighboring pixels inside the images. A CNN model can then be used to extract and learn from the mapped images. Specifically, we experiment on EHR data on Chronic Lymphocytic Leukemia (CLL) patients and make predictions on whether the next line of treatments needs to be initiated.

5.2 Method

5.2.1 Mapping Electronic Health Records to images

For each patient, we take the diagnosis-treatment features from EHR data and map them to images. As a result, each patient will have a unique image that represents his/her doctor visits journey. To map diagnosis-treatment features, we need to consider: the relationships between features, the values of the features, encoding time sequence information, and how to position features inside of the image according to their relationships.

Relationships between procedure, prescription and diagnosis EHRs contain various types of information such as patient demographic information like age, gender, and ethnicity; and patient encounter information such as diagnosis codes, procedures used, and drugs prescribed. There exists structural information among the procedure, prescription, and diagnosis features. For example, doctors usually give the procedure EKG based on the diagnosis of chest pain. It means that chest pain and EKG are closely related.

In images, pixels that are physically close together share similar information. Similarly, transforming EHRs into images requires positioning similar entities together. EHRs often do not contain structural information; however, relations between the medical entities can be measured using quantitative methods. To extract these kinds of associations from EHRs, we look at how frequently two medical entities appear together in a medical claim. The more frequently they appear together, the more likely they are related. Specifically, we use conditional probability to measure the relationship between a treatment (procedure or prescription) and a diagnosis. Equation 5.1 shows how the relationship is measured:

$$P(rx|dx) = P(rx \cap dx)/P(dx) \tag{5.1}$$

P represents probability, rx is a treatment and dx is a diagnosis.

When mapping these medical entities into images, we want the related features to be positioned as neighboring pixels as much as possible. Figure 5.1 shows an example of how some medical features are positioned inside of an image. Related features, such as fever, IV fluid, and Advil are positioned as neighboring pixels. There might not be enough features to fill the image, so some pixels are empty and do not represent any features.

Values of Medical Features The value for each pixel inside of an image ranges from 0 to 1: 0 represents black, 1 represents white, and the values in between are various degrees of grays. For each diagnosis-treatment feature, we can count how many times they appear for each patient during a given time interval. And for each feature, we normalize its value to the $[0, 1]$ range.

Time information as image channels The EHR data contains patients' visits information over time. For a given time interval, we divide it into three parts, and how many times each feature appears for each part is counted. The counts are then normalized to a value in the $[0, 1]$ range. The three parts of time intervals correspond to the three channels

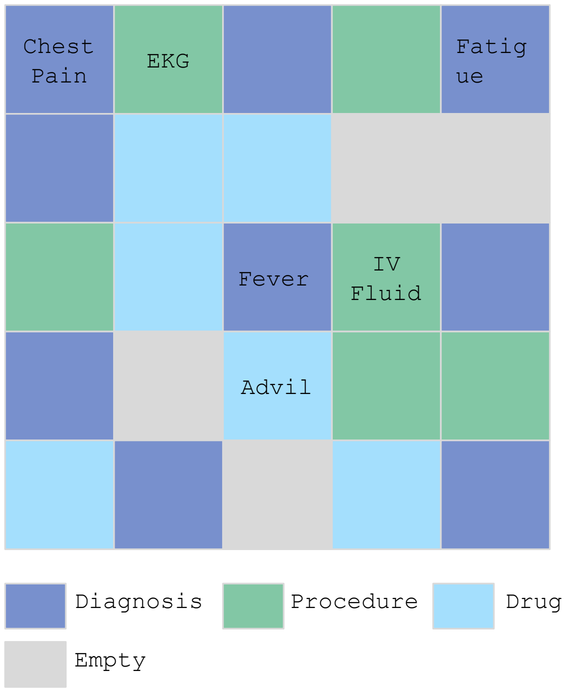


Figure (5.1) An example of how related features are positioned as neighboring pixels inside of an image.

of an image. This is how features changes over time are aggregated and encoded into the image channels.

Genetic Algorithms for optimizing entity positions How to arrange the positions of diagnosis and treatments in an image such that related entities are as close together as possible is an optimization problem. Suppose the number of diagnosis and procedure together is N, and diagnosis and treatments are mapped to an image of size $W \cdot H$, there will be $(W \cdot H)!/(W \cdot H - N)!$ ways to arrange. To traverse through each possible arrangement and evaluate how close together we are positioning the medical features is a computationally impossible task in practice because as the number of features increases, the number of ways of positioning grows super-exponentially. GAs can be used to find solutions to optimization problems efficiently. A GA is a kind of EA inspired by the process of natural selection. It is commonly used to generate high-quality solutions when the size of the search space for solutions is very large. Typically, an initial population of solutions is randomly generated

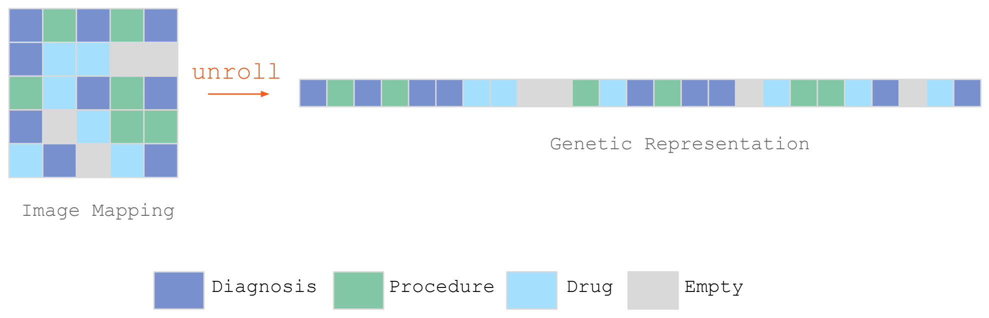


Figure (5.2) The genetic representation of feature positions inside of an image is the unrolling of the image.

and then evolved to generate a population of better solutions. Biologically inspired operators such as selection, crossover, and mutation are applied to the solutions during the evolutionary process. The algorithm lets better solutions survive into new generations and eliminate bad ones.

To use a GA, we need to have genetic representations for the solutions. For our problem, we simply unroll the image mapping to a vector that contains diagnosis and treatments, as shown in Figure 5.2. The unrolled vectors are the genetic representation of image mapping solutions.

We design modified genetic operators to generate new populations. We still use selection, crossover, and mutation operations. However, the original crossover and mutation operations can not be directly applied for diagnosis and treatment ordering because each entity appears in the chromosome for exactly one time. Using the original crossover and mutation would cause the solutions to contain some entities multiple times while missing other ones. Instead, we use modified crossover and mutation operations. Figure 5.3 shows our crossover operations. Two chromosomes, Parent a and Parent b, are recombined to generate two children. The children contain genes from both parents. To produce Child a, the front part of the chromosome from Parent a (bounded by the red box) is first taken. However, we cannot simply use the latter part of Parent b. Instead, from Parent b, we remove the genes Parent a has already passed on to the child, concatenate the remaining genes in order,

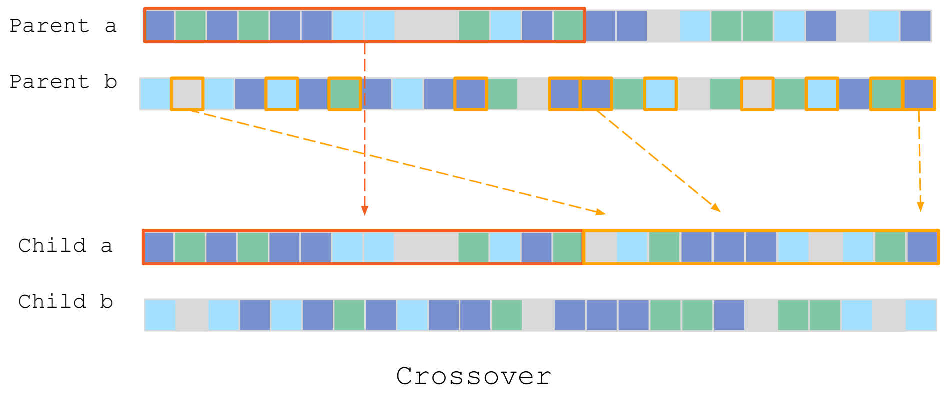


Figure (5.3) Modified crossover operation.

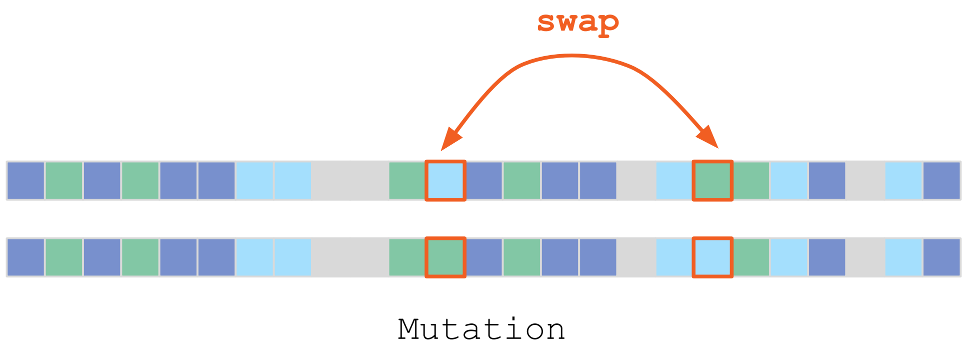


Figure (5.4) Mutation operations swaps two values.

and give that to Child a (bounded by the orange boxes). Child b is produced in a similar way as Child a. For mutation, two genes in the chromosome are randomly chosen, and their positions are swapped. These modified genetic operations guarantee that the children inherit from both parents while containing each diagnosis and treatment for exactly one time.

We also design a fitness function to evaluate how good are the solutions. For each treatment (procedure or drug), we find its N-nearest diagnosis neighbors $\{N_1, N_2, \ldots, N_n\}$ according to the condition probability scores, with N_1 being the closest diagnosis neighbor. In the image mapping, we define a neighborhood for each treatment. The neighborhood is simply the adjacent pixels to the treatment pixel. As shown in Figure 5.5, the neighborhood for drug Advil is bounded by the red box and has a size $3x3$. For each treatment i, we check if any diagnosis in $\{N_1, N_2, \ldots, N_n\}$ is within its neighborhood and assign a corresponding score T_i. The value of T_i is determined by which diagnosis is within I's neighborhood. For

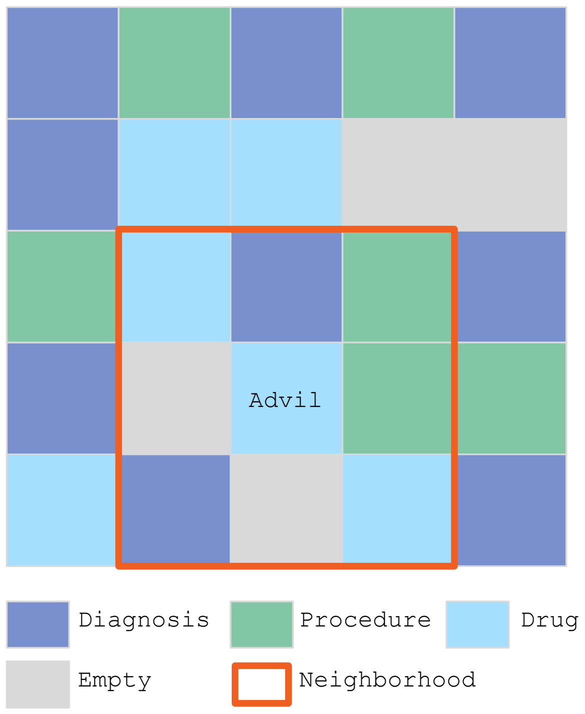


Figure (5.5) A 3x3 neighborhood example.

example, if N_1 is within I's neighborhood, T_i takes on a small value such as 1. If no diagnosis is within I's neighborhood, T_i is infinity. The fitness score of an image mapping is $sum(Ti)$ for all treatments i. We want to optimize the image mapping to have a small fitness score as possible.

The overall steps of our modified GA for diagnosis-treatment to image mapping is as follows:

Algorithm 3: Diagnosis Treatments Mapping Optimization

```
begin
    create initial population of random solutions $P$;
    choose encoding scheme to encode image mappings to individual chromosomes;
    while not stopping_condition do
        $P' = \varnothing$;
        calculate fitness $f(i)$ for each individual $i$ in $P$;
        sort $f(i)$ for all $i$ in $P$;
        select $I$ fittest individuals in $P$;
        $P' = P' \cup I$;
        randomly select $I$ less fit individuals in $P$;
        $P' = P' \cup I$;
        while $P'.size < P.size$ do
            select $p_1$ and $p_2$ from $P'$;
            crossover $p_1$ and $p_2$ to get $h_1$ and $h_2$;
            mutate $h_1$ and $h_2$ to get $c_1$ and $c_2$ $P' = P' \cup \{c_1, c_2\}$;
        end
        $P = P'$;
    end
    decode the individual $i$ with the max fitness to get best image mapping rule $M$;
    return $M$.
end
```

5.2.2 Data Augmentation

One problem for prediction tasks on EHRs is that the dataset is usually highly unbalanced. The number of positive samples is very small compared with the negatives ones. Having highly unbalanced data will cause deep learning models to perform poorly since it will be difficult for the model to learn from the minority class. Using data augmentation [67] to boost the number of positives samples is a solution to data imbalance. Popular image augmentation methods such as translation, rotation, and flipping do not suit our needs here. Since pixels of different coordinates have different meanings – they represent different diagnosis or treatments, we do not want to use translations or rotations as they change pixel locations. A popular method used for EHR data augmentation is Synthetic Minority Oversampling Technique (SMOTE) [11]. SMOTE works by selecting two samples, drawing a line between them in the feature space, and generating a new sample along the line. We found the data augmentation using SMOTE to be very slow due to a large amount of data and high feature dimensions that we have. We use a novel data augmentation technique for EHR data. The technique is inspired by image overlay data augmentation methods [40], with adaptation to the EHR domain. As shown in Figure 5.6, we take all features from a positive sample and overlay them with part/all features from a negative sample to produce a new synthesized positive sample. Intuitively, positive samples plus features from the negative samples should still generate positive samples. This technique is easy to implement and runs very efficiently in practice.

Equation 5.2 is used to generate a synthesized positive patient image.

$$I_{aug}[i,j] = min(1, I_{pos}[i,j] + Mask[i,j] \cdot I_{neg}[i,j]) \tag{5.2}$$

I_{aug} is the augmented image, I_{pos} is a positive image, I_{neg} is a negative image. i and j are coordinates for a pixel in the image. $Mask[i,j]$ is a mask for the pixel that takes either 0 or 1. This value is randomly generated according to a preset probability. The function min takes the minimum value between two values.

5.2.3 Convolutional Neural Networks for image mapping classification

We process EHR data for every patient and convert the diagnosis-treatment information to images. Every patient has one image that encodes the EHR information over a time interval. After mapping diagnosis and treatments to images, we pass the images to a CNN

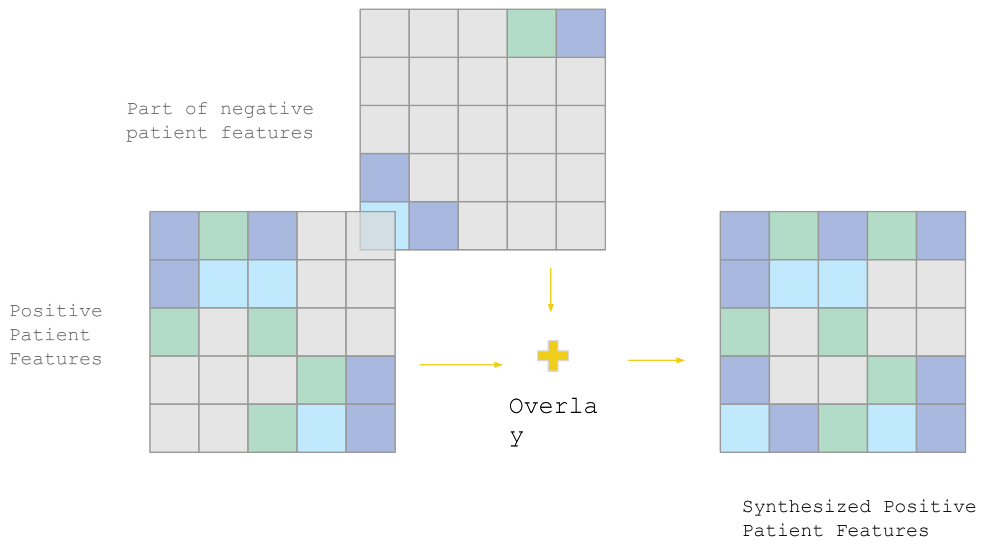

Part of negative patient features
Positive Patient Features
Overla y
Synthesized Positive Patient Features

Figure (5.6) To generate a synthesize positive patient image, a positive patient's feature is overlaid with part of a negative patient's features.

for feature extraction and classification. A CNN model of multiple convolution layers is created for a binary classification task. Algorithm 4 describes how the convolution filters learn the similarity information from neighboring pixels.

Algorithm 4: Convolution for feature similarity learning

```
begin
    obtain feature to image mapping rule $M$ from Algorithm 3;
    apply image mapping rule to convert diagnosis, procedure, drug features
      $\{D_x, P_x, R_x\}$ to images $X = M(D_x, P_x, R_x)$;
    obtain training sample scores $P$ using a base machine learning model;
    remove $n$ potentially confusing training examples according to $P$;
    data augmentation to produce balanced train data $X\_train$ according to
      Equation 5.2;
    tune CNN model architecture;
    train CNN model weights $W$ using $X\_train$;
    inference scores $P\_test$ on test datasets using trained model weights $W$.
end
```

5.2.4 Overall Model Architecture

Figure 5.7 shows the overall architecture of our model. Our input data contains both patient demographic information and patient journey information. All of this information is fed to the base machine learning model, and prediction scores are produced. The patient journey information - sequences of procedures, diagnosis, and treatments are mapped to images using optimized image mapping rules. We then use our data augmentation method to balance the positive and negative classes. The output scores from the base machine learning model here are also used to remove some training samples that could potentially be confusing to the deep learning model. The images are then fed to a CNN model for prediction. The output scores from the CNN model are finally combined with the base machine learning model for a final prediction score.

5.3 Experiments and Results

We perform our experiments on a CLL dataset. We compare our results with base machine learning models on the validation dataset. Top k patients with the highest scores are selected, and precisions are calculated for different k values;

5.3.1 Chronic Lymphocytic Leukemia Dataset

We perform our experiments on the CLL dataset. This dataset is obtained from the IQVIA longitudinal prescription claims database and electronic medical claims database. IQVIA receives 2 billion prescription claims per year with history from January 2004 with coverage up to 88% for the retail channel, 50-70% for traditional and specialty mail order, and 40% for long-term care. This information represents activities during the prescription transaction and contains information regarding the product, provider, payer, and geography. Prescription claims data is longitudinally linked back to an anonymous patient token and can be linked to events within the dataset and across other patient data assets. IQVIA receives just under 1 billion office-based electronic medical claims from office-based individual

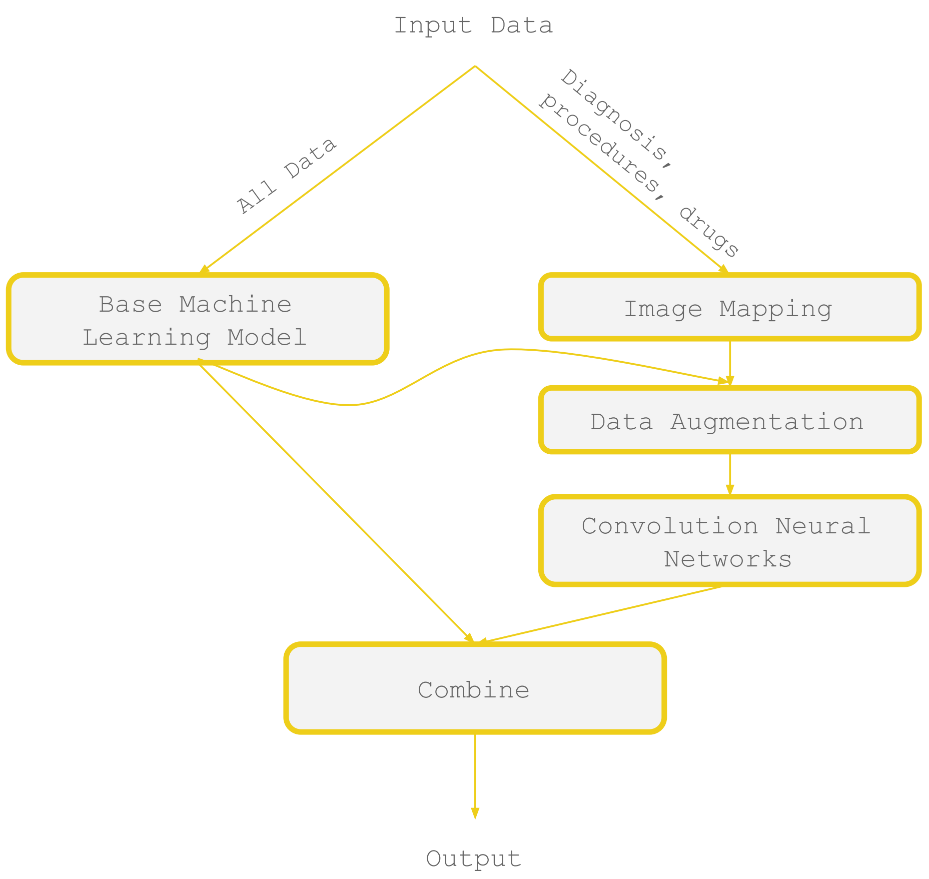

Input Data
All Data
Diagnosis, procedures, drugs
Base Machine Learning Model
Image Mapping
Data Augmentation
Convolution Neural Networks
Combine
Output

Figure (5.7) The overall architecture of the proposed model.

professionals, ambulatory, and general healthcare sites per year, including patient-level diagnosis and procedure information. The information represents nearly 40% of all electronically filed medical claims in the US and is now the largest diagnosis database of its kind. The medical claims data includes information that is tracked to a specific prescriber enabling the user to obtain a complete picture of a prescriber's activity, which can, in turn, be linked to prescription data, affiliations, and other attributes of the practitioner. Information for patients who were diagnosed with CLL is selected from the databases. We select the training cohort from April 2018 through May 2019. We divide this time frame into three equal time intervals. Positive patients are those who were diagnosed for CLL before the interval and initiated treatments within the time interval, whereas negative ones were diagnosed before the interval but did not receive any treatments inside of the interval. The validation dataset

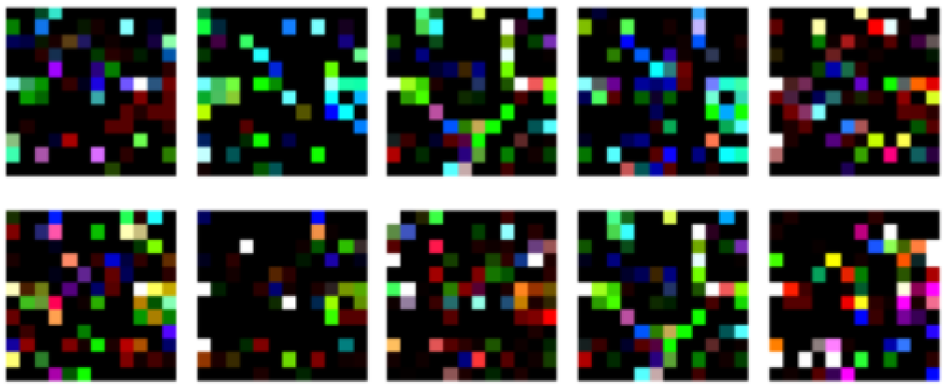


Figure (5.8) Actual examples of mapping CLL diagnosis and treatments to images.

is cohort from 201909 to 201912. Due to potential patient information overlap from neighboring cohorts, we ensured that validation cohorts are at least three months later than the most recent training cohort to avoid information leakage. Overall, the amount of positive patients is around 3% among all CLL patients. Figure 5.8 shows some real examples of how the EHR diagnosis-treatment data for CLL patients look like when converted to images.

5.3.2 Disease progression prediction on CLL dataset

Since our dataset is highly unbalanced between positives and negatives, we use number of relevant items present in the top-k recommendation of our algorithm (Precision at K) to evaluate model performances. Scores are calculated for all patients, and k patients with the highest scores are selected. We then calculate among the selected patients to see what is the percentage of patients who are actually positive.

We apply our method on three baseline models for comparison: Random Forest, XGBoost (XGB) [14], and CatBoost. We evaluate the performance of these models with and without CNN learned feature similarities.

In practice, we care about model performance for the top 3,000 patients with the highest scores; thus we compare for values 3,000 and less.

As shown in Figure 5.9, we compare the base machine models with our CNN boosted models. We also include ensembles of baseline models as a reference. It can be seen that our

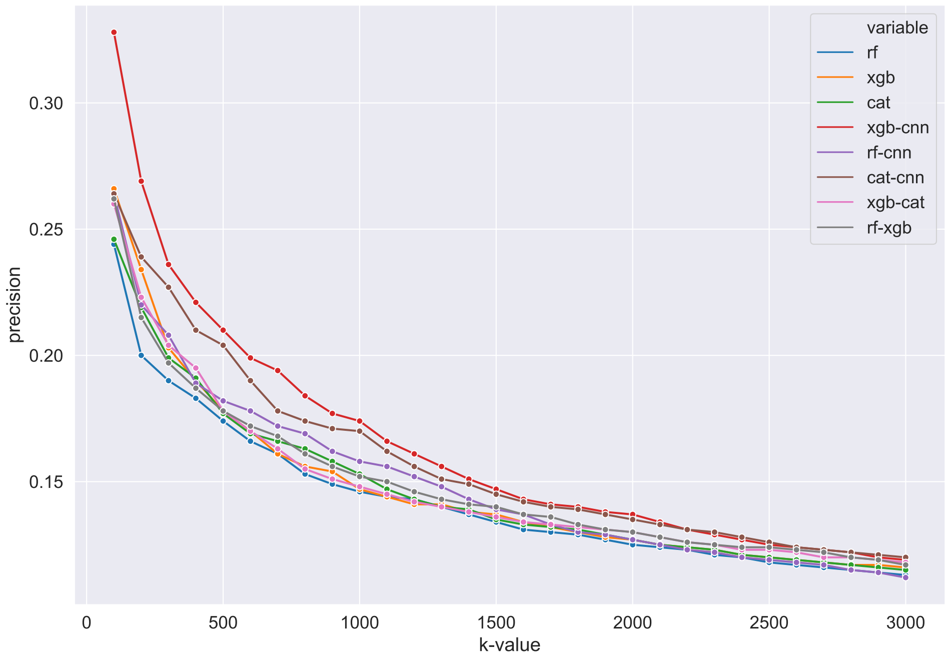


Figure (5.9) Overall model performance.

model outperforms the baselines overall.

In Table 5.1, we show the Precision at K results for different values of k: 100, 300, 500, 1,000, 2,000, and 3,000. We compare our method with the base machine learning models: Random Forest, XGB, and CatBoost. It can be observed that our method has better precision values compared with the baseline models. When k is smaller, the improvements tend to be larger.

5.4 Summary

In this work, we propose to map procedure, prescription, and diagnosis features in an EHR to images and use a CNN to learn the features and local patterns of the images. The relationship between these medical features is taken into consideration this way. We apply our method to real-world patient medical data, and experimental results show that our method can boost the performance of a few base machine learning models. In the future, we

Table (5.1) Precision comparison on the CLL dataset with different K values. The CNN boosted Model outperforms the baseline models.

K value	**100**	**300**	**500**	**1,000**	**2,000**	**3,000**
Random Forest	24.4%	19.0%	17.4%	14.6%	12.5%	11.3%
CNN-Boosted-RF	26.4% (+2.0%)	20.8% (+1.8%)	18.2% (+0.8%)	15.8% (+1.2%)	12.7% (+0.2%)	11.3% x
XGB	26.6%	20.3%	17.8%	14.7%	12.7%	11.5%
CNN-Boosted-XGB	32.8% (+6.2%)	23.6% (+3.3%)	21.0% (+3.2%)	17.4% (+2.7)%	13.7% (+1.0%)	11.9% (+0.4%)
CatBoost	24.6%	19.9%	17.7%	15.3%	12.7%	11.5%
CNN-Boosted-Cat	26.4% (+1.8%)	22.7% (+ 2.8%)	20.4% (+2.7%)	17.0% (+1.7%)	13.5% (+0.8%)	12.0% (+0.5%)
RF + XGB	26.2%	19.7%	17.8%	15.2%	13.0%	11.7%
XGB + CAT	26.0%	20.4%	17.8%	14.8%	13.0%	11.8%

plan to apply our method to more medical datasets. Also, medical features can be mapped to 3D images where more detailed time sequence information can be encoded. The 3D images can then be fed into 3D-CNN models for feature learning.

PART 6

CONCLUSIONS AND FUTURE WORKS

6.1 Conclusions

In this research, we present various methods to improve the training time and accuracy of deep learning models. Specifically, we propose a method that efficiently tunes hyperparameters of CNNs automatically. We use a variable length GA to tune hyperparameters of a CNN model. The results are compared with a fix-length GA and a mutation only EA. Experimental results show that our method can discover models with better performance when under similar time constraints. We also designed an intelligent freezing method that accelerates the training of deep learning models. A formula is designed to indicate which layers can be frozen during model training to save time. This formula is designed by analyzing the gradient changes of deep learning layers. Experimental results on standard computer vision datasets and various deep learning models show that the intelligent freezing method can save training time while maintaining similar model prediction performance. We also explore ways to incorporate prior knowledge into deep learning training. We extract class similarity knowledge from the validation dataset and use a graph convolution layer to embed this knowledge in a CNN model. We performed analysis on the effect of adding prior knowledge when the size of the model, and amount of training data varies. And results show that adding knowledge helps, especially when the amount of training data is small. We also propose a method that maps non-image data into images while taking feature similarity information into consideration. Specifically, we turn EHRs into images and use a CNN to learn the local feature patterns. We add this component as a boosting module to build machine learning models. Results show that this method has improvements over the baselines.

6.2 Future Works

6.2.1 Extend Layer Freezing to Freeze Arbitrary Layers

We plan to extend the work mentioned in Chapter 3: freezing layers to save training time. Right now, in this work, there are requirements on which layers can be frozen. Because of the nature of backpropagation, freezing must start at the first layer and be consecutive. The question is whether we can employ some techniques so that we can freeze arbitrary layers in the model.

If a layer is frozen, we do not calculate its gradients. However, if we want to keep the layer before the frozen one active, we need to know how to calculate its gradients so we can update its weights. We cannot use back-propagation because the next layer is frozen. One way is to use the old gradients from the previous epoch. There are also ways to predict a layer's gradients based on its gradients in previous epochs.

6.2.2 Use More Metrics for Model Evaluations

Additional metrics can be used for a more comprehensive analysis on our models. For example, when comparing our intelligent freezing method with the baseline, we used time and test accuracy as evaluation metrics. We wanted to make sure that similar test accuracy can be maintained while reducing training time. In the future, we can apply our freezing methods to class imbalanced datasets and observe the effect on precision and recall. Also, when experimenting on datasets with imbalanced classes, receiver operating characteristic (ROC) curve and area under the ROC (AUC) curve can be used as additional metrics for a more thorough understanding of the model performance.

6.2.3 Combining Improvements from Various Methods

We proposed various methods to improve time and accuracy of deep learning models. It would be interesting to combine multiple methods together to see if that gives us further more improvements. For example, can combining freezing and prior knowledge together contribute to both faster training and higher model accuracy? Hessel et al. [30] combined algorithms in deep reinforcement learning and studied their combination. We can do similar experiments and analysis on our methods.

6.2.4 Use Prior Knowledge to Generate Soft Targets

Typically, the label for the data is one-hot encoded in a classification problem: the target class is 1 and the rest of the classes are 0's. The targets are hard targets since they cannot take other values besides 0 and 1. Soft targets on the other hand, is a real value between 0 and 1. Unlike hard targets, soft targets can carry more information: the similarity between classes. For example, it can tell us which 2's look like 7's and which 4's look like 9's. Soft targets have been used to distill knowledge from a cumbersome ensemble of models into a small model. We plan to calculate the soft targets using prior knowledge. Since we have already defined class similarity using misclassification graph in Chapter 4, we can start from there. We plan to use the misclassification graph to generate soft targets, combine that which the hard targets, and see how that can help with the deep learning model.

6.2.5 Use Graph Attention Networks to Learn Class Similarities

Instead of using a graph convolutional layer at the end of a CNN mode with predefined edge information, we can use a graph attention layer. Graph attention networks do not require edge information as it learns the edges through the training process. It would be interesting to add a graph attention layer at the end of a CNN, train the model, and observe the learned edges. Analysis can be performed to see if the learned edges correlate to class similarities.

BIBLIOGRAPHY

[1] Saleh Albelwi and Ausif Mahmood. “A Framework for Designing the Architectures of Deep Convolutional Neural Networks”. In: *Entropy* 19.6 (2017), p. 242 (cit. on p. 9).

[2] Kazuma Arino and Yohei Kikuta. “ClassSim: Similarity between Classes Defined by Misclassification Ratios of Trained Classifiers”. In: *arXiv preprint arXiv:1802.01267v1* (2018) (cit. on pp. 50, 58).

[3] Barrie M. Baker and M.A. Ayechew. “A Genetic Algorithm for the Vehicle Routing Problem”. In: *Computers & Operations Research* 30.5 (Apr. 2003), pp. 787–800 (cit. on p. 8).

[4] Bowen Baker et al. “Designing Neural Network Architectures using Reinforcement Learning”. In: *arXiv preprint arXiv:1611.02167* (Nov. 2016) (cit. on pp. 9, 48).

[5] Alejandro Baldominos, Yago Saez, and Pedro Isasi. “Evolutionary convolutional neural networks: An application to handwriting recognition”. In: *Neurocomputing* 283 (Mar. 2018), pp. 38–52 (cit. on pp. 9, 48).

[6] TK Bamunu Mudiyanselage et al. “Deep Fuzzy Neural Networks for Biomarker Selection for Accurate Cancer Detection”. In: *IEEE Transactions on Fuzzy Systems* (2019) (cit. on pp. 25, 64).

[7] Sunitha Basodi et al. “Gradient Amplification: An efficient way to train deep neural networks”. In: *arXiv preprint arXiv:2006.10560* (June 2020) (cit. on p. 48).

[8] Inci M Baytas et al. “Patient Subtyping via Time-Aware LSTM Networks”. In: *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 2017 (cit. on p. 73).

[9] James Bergstra and Yoshua Bengio. “Random Search for Hyper-Parameter Optimization”. In: *J. Mach. Learn. Res.* 13 (2012), pp. 281–305 (cit. on pp. 8, 24, 48).

[10] Andrew Brock et al. "FreezeOut: Accelerate Training by Progressively Freezing Layers". In: *arXiv preprint arXiv:1706.04983* (June 2017) (cit. on pp. 28, 33, 34).

[11] Nitesh V Chawla et al. "SMOTE: Synthetic Minority Over-sampling Technique". In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357 (cit. on p. 81).

[12] Chao Che et al. "An RNN Architecture with Dynamic Temporal Matching for Personalized Predictions of Parkinson's Disease". In: *Proceedings of the 2017 SIAM International Conference on Data Mining*. Philadelphia, PA: Society for Industrial and Applied Mathematics, June 2017 (cit. on p. 73).

[13] Meihao Chen, Zhuoru Lin, and Kyunghyun Cho. "Graph Convolutional Networks for Classification with a Structured Label Space". In: *arXiv preprint arXiv:1710.04908v2* (2016) (cit. on p. 3).

[14] Tianqi Chen and Carlos Guestrin. "XGBoost: A Scalable Tree Boosting System". In: *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 2016, pp. 785–794 (cit. on p. 85).

[15] Edward Choi et al. "Learning the Graphical Structure of Electronic Health Records with Graph Convolutional Transformer". In: *arXiv preprint arXiv:1906.04716* (2019) (cit. on p. 73).

[16] Edward Choi et al. "MiME: Multilevel Medical Embedding of Electronic Health Records for Predictive Healthcare". In: *arXiv preprint arXiv:1810.09593* (2018) (cit. on p. 73).

[17] Edward Choi et al. "RETAIN: An Interpretable Predictive Model for Healthcare using Reverse Time Attention Mechanism". In: *Advances in Neural Information Processing Systems*. 2016 (cit. on p. 73).

[18] Edward Choi et al. "Using recurrent neural network models for early detection of heart failure onset." In: *Journal of the American Medical Informatics Association : JAMIA* 24.2 (2017), pp. 361–370 (cit. on p. 73).

[19] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. “Fast and accurate deep network learning by exponential linear units (ELUs)”. In: *arXiv preprint arXiv:1511.07289* (2015) (cit. on p. 7).

[20] Jia Deng et al. “Large-Scale Object Classification using Label Relation Graphs”. In: *Computer Vision – ECCV*. 2014, pp. 48–64 (cit. on p. 50).

[21] Michelangelo Diligenti, Soumali Roychowdhury, and Marco Gori. “Integrating prior knowledge into deep learning”. In: *Proceedings - 16th IEEE International Conference on Machine Learning and Applications, ICMLA 2017* 2018-Janua (2018), pp. 920–923 (cit. on p. 49).

[22] Xintao Ding et al. “Prior knowledge-based deep learning method for indoor object recognition and application”. In: *Systems Science & Control Engineering* 6.1 (Jan. 2018), pp. 249–257 (cit. on p. 49).

[23] John Duchi, Elad Hazan, and Yoram Singer. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2121–2159 (cit. on p. 33).

[24] Jie Fu et al. “DrMAD: Distilling Reverse-Mode Automatic Differentiation for Optimizing Hyperparameters of Deep Neural Networks”. In: *Proceedings of the Twenty-Fifth International Joint Conference on Artificial Intelligence*. Jan. 2016, pp. 1469–1475 (cit. on p. 9).

[25] X Fu et al. “Using a genetic algorithm approach to solve the dynamic channel-assignment problem”. In: *International Journal of Mobile Communications* 4.3 (2006), pp. 333–353 (cit. on p. 8).

[26] Jatinder N.D Gupta and Randall S Sexton. “Comparing backpropagation with a genetic algorithm for neural network training”. In: *Omega* 27.6 (Dec. 1999), pp. 679–684 (cit. on p. 8).

[27] Aaron Harlap et al. “PipeDream: Fast and Efficient Pipeline Parallel DNN Training”. In: *arXiv preprint arXiv:1806.03377* (June 2018) (cit. on p. 30).

[28] Kaiming He et al. “Deep Residual Learning for Image Recognition”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016 (cit. on pp. 1, 29, 43).

[29] Yihui He, Xiangyu Zhang, and Jian Sun. “Channel pruning for accelerating very deep neural networks”. In: *arXiv preprint arXiv:1707.06168* (2017) (cit. on p. 31).

[30] Matteo Hessel et al. “Rainbow: Combining Improvements in Deep Reinforcement Learning”. In: *arXiv preprint arXiv:1710.02298* (2017) (cit. on p. 90).

[31] Chris Hettinger et al. “Forward Thinking: Building and Training Neural Networks One Layer at a Time”. In: *arXiv preprint arXiv:1706.02480* (2017) (cit. on p. 33).

[32] Geoffrey Hinton et al. “Deep Neural Networks for Acoustic Modeling in Speech Recognition”. In: *Signal Processing Magazine, IEEE* 29.6 (2012), pp. 82–97 (cit. on p. 1).

[33] Sepp Hochreiter and Jurgen Schmidhuber. “Long short-term memory”. In: *MEMORY Neural Computation* 9.8 (1997), pp. 1735–1780 (cit. on p. 1).

[34] Jie Hu et al. “TW-Co-MFC: Two-level weighted collaborative fuzzy clustering based on maximum entropy for multi-view data”. In: *Tsinghua Science and Technology* 26.2 (Apr. 2020), pp. 185–198 (cit. on p. 64).

[35] Zhiting Hu et al. “Harnessing Deep Neural Networks with Logic Rules”. In: *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, 2016, pp. 2410–2420 (cit. on p. 49).

[36] Gao Huang et al. “Deep Networks with Stochastic Depth”. In: *arXiv preprint arXiv:1603.09382* (2016) (cit. on pp. 33, 34).

[37] Gao Huang et al. “Densely Connected Convolutional Networks”. In: *CVPR*. 2017 (cit. on pp. 29, 43).

[38] Sung Ju Hwang, Kristen Grauman, and Sha Fei. “Semantic Kernel Forests from Multiple Taxonomies”. In: *Advances in neural information processing systems*. 2012, pp. 1718–1726 (cit. on p. 50).

[39] Forrest N Iandola et al. *FireCaffe: near-linear acceleration of deep neural network training on compute clusters*. Tech. rep. (cit. on p. 31).

[40] Hiroshi Inoue. “Data Augmentation by Pairing Samples for Images Classification”. In: *arXiv preprint arXiv:1801.02929v2* (2018) (cit. on pp. 48, 81).

[41] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: *arXiv preprint arXiv:1502.03167* (2015) (cit. on pp. 29, 31).

[42] Chunyan Ji et al. “Infant Sound Classification on Multi-stage CNNs with Hybrid Features and Prior Knowledge”. In: *Artificial Intelligence and Mobile Services – AIMS 2020”*. Sept. 2020, pp. 3–16 (cit. on p. 49).

[43] Chenhan Jiang et al. “Hybrid Knowledge Routed Modules for Large-scale Object Detection”. In: *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. Montreal, Canada: Curran Associates Inc., 2018, pp. 1559–1570 (cit. on p. 49).

[44] Aman Chandra Kaushik et al. “CoronaPep: An Anti-coronavirus Peptide Generation Tool”. In: *IEEE/ACM Transactions on Computational Biology and Bioinformatics* (2021) (cit. on p. 1).

[45] Yong-Deok Kim et al. “Compression of deep convolutional networks for fast and low power mobile applications”. In: *arXiv preprint arXiv:1511.06530* (2016) (cit. on p. 31).

[46] Thomas N Kipf and Max Welling. “Semi-supervised Classification With Graph Convolutional Networks”. In: *arXiv preprint arXiv:1609.02907v4* (2017) (cit. on p. 51).

[47] Günter Klambauer et al. “Self-Normalizing Neural Networks”. In: *arXiv preprint arXiv:1706.02515* (2017) (cit. on p. 7).

[48] Alex Krizhevsky. *Learning Multiple Layers of Features from Tiny Images.* Tech. rep. Department of Computer Science, University of Toronto, 2009 (cit. on pp. 2, 10, 16, 32, 43, 63).

[49] Andrew Lavin and Scott Gray. “Fast Algorithms for Convolutional Neural Networks”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016 (cit. on p. 31).

[50] Vadim Lebedev and Victor Lempitsky. “Fast ConvNets using group-wise brain damage”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* 2016 (cit. on p. 31).

[51] Hae Beom Lee et al. “DropMax: Adaptive variational softmax”. In: *Proceedings of the 32Nd International Conference on Neural Information Processing Systems.* Montreal, Canada: Curran Associates Inc., 2018, pp. 927–937 (cit. on pp. 3, 49, 58).

[52] Zachary C Lipton et al. “Learning to diagnose with lstm recurrent neural networks”. In: *arXiv preprint arXiv:1511.03677* (2015) (cit. on p. 73).

[53] Hui Liu et al. “An adaptive genetic fuzzy multi-path routing protocol for wireless ad-hoc networks”. In: *Sixth International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing and First ACIS International Workshop on Self-Assembling Wireless Network.* 2005, pp. 468–475 (cit. on pp. 25, 64).

[54] Jelena Luketina et al. “Scalable Gradient-Based Tuning of Continuous Regularization Hyperparameters”. In: *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48.* 2016, pp. 2952–2960 (cit. on p. 9).

[55] Marijnvk. *Large Scale Evolution.* 2018 (cit. on p. 23).

[56] Michael Mathieu, Mikael Henaff, and Yann Lecun. “Fast training of convolutional networks through FFTs”. In: *arXiv preprint arXiv:1312.5851* (2014) (cit. on p. 31).

[57] Conrado S Miranda and Fernando J Von Zuben. “Reducing the training time of neural networks by partitioning”. In: *arXiv preprint arXiv:1511.02954* (2016) (cit. on p. 30).

[58] David J Montana and Lawrence Davis. “Training Feedforward Neural Networks Using Genetic Algorithms”. In: *Proceedings of the 11th international joint conference on Artificial Intelligence.* San Francisco, CA, USA, 1989, pp. 762–767 (cit. on p. 8).

[59] Yu Nesterov. “A method of solving a convex programming problem with convergence rate o (1/k2)”. In: *Soviet Mathematics Doklady* 27.2 (1983), pp. 372–376 (cit. on p. 33).

[60] Sunil Nilkanth Pawar and Rajankumar Sadashivrao Bichkar. “Genetic Algorithm with Variable Length Chromosomes for Network Intrusion Detection”. In: *International Journal of Automation and Computing* 12.3 (June 2015), pp. 337–342 (cit. on p. 8).

[61] Esteban Real et al. “Large-Scale Evolution of Image Classifiers”. In: *Proceedings of the 34th International Conference on Machine Learning - Volume 70.* Sydney, NSW, Australia, 2017, pp. 2902–2911 (cit. on pp. 6, 8–10, 22–24, 48).

[62] Herbert Robbins and Sutton Monro. “A stochastic approximation method”. In: *Annals of Mathematical Statistics* 22 (1951), pp. 400–407 (cit. on p. 2).

[63] Omer Sagi and Lior Rokach. “Ensemble learning: A survey”. In: *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8.4 (July 2018) (cit. on p. 48).

[64] Tim Salimans and Diederik P Kingma. “Weight Normalization: A Simple Reparameterization to Accelerate Training of Deep Neural Networks”. In: *arXiv preprint arXiv:1802.09750* (2016) (cit. on p. 31).

[65] Bobak Shahriari et al. “Taking the Human Out of the Loop: A Review of Bayesian Optimization”. In: *Proceedings of the IEEE* 104.1 (2016), pp. 148–175 (cit. on p. 8).

[66] Alok Sharma et al. “DeepInsight: A methodology to transform a non-image data to an image for convolution neural network architecture”. In: *Scientific Reports* 9.1 (2019), pp. 1–7 (cit. on p. 74).

[67] Connor Shorten and Taghi M. Khoshgoftaar. “A survey on Image Data Augmentation for Deep Learning”. In: *Journal of Big Data* 6.1 (Dec. 2019), p. 60 (cit. on pp. 48, 81).

[68] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2015) (cit. on pp. 29, 43).

[69] Bharat Singh et al. “Layer-Specific Adaptive Learning Rates for Deep Networks”. In: *2015 IEEE 14th International Conference on Machine Learning and Applications (ICMLA)*. Miami, FL, Dec. 2015, pp. 364–368 (cit. on pp. 32, 34).

[70] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. “Practical Bayesian Optimization of Machine Learning Algorithms”. In: *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 2*. Lake Tahoe, Nevada, USA, 2012, pp. 2951–2959 (cit. on p. 8).

[71] R. Srikanth et al. “A Variable-Length Genetic Algorithm for Clustering and Classification”. In: *Pattern Recognition Letters* 16.8 (Aug. 1995), pp. 789–800 (cit. on p. 8).

[72] Nitish Srivastava et al. “Dropout: A Simple Way to Prevent Neural Networks from Overfitting”. In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958 (cit. on pp. 33, 34, 48).

[73] Russell Stewart and Stefano Ermon. “Label-Free Supervision of Neural Networks with Physics and Domain Knowledge”. In: *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*. San Francisco, California, USA: AAAI Press, 2017, pp. 2576–2582 (cit. on p. 49).

[74] Masanori Suganuma, Shinichi Shirakawa, and Tomoharu Nagao. "A genetic programming approach to designing convolutional neural network architectures". In: *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '17*. Berlin, Germany, Apr. 2017, pp. 497–504 (cit. on pp. 9, 48).

[75] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *Proceedings of the 30th International Conference on Machine Learning*. Ed. by Sanjoy Dasgupta and David McAllester. Atlanta, Georgia, USA, 2013, pp. 1139–1147 (cit. on p. 31).

[76] Kevin Swersky, Jasper Snoek, and Ryan P Adams. "Multi-Task Bayesian Optimization". In: *Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2*. Lake Tahoe, Nevada, USA, 2013, pp. 2004–2012 (cit. on p. 8).

[77] Ryo Takahashi, Takashi Matsubara, and Kuniaki Uehara. "Data Augmentation using Random Image Cropping and Patching for Deep CNNs". In: *arXiv preprint arXiv:1811.09030v2* (2018) (cit. on p. 48).

[78] Eric J. Topol. "High-performance medicine: the convergence of human and artificial intelligence". In: *Nature Medicine* 25.1 (Jan. 2019), pp. 44–56 (cit. on p. 73).

[79] Geoffrey G Towell and Jude W Shavlik. "Knowledge-Based Artificial Neural Networks". In: *Artif. Intell.* 70.1-2 (1994), pp. 119–165 (cit. on p. 49).

[80] Mark J.J.P. Van Grinsven et al. "Fast Convolutional Neural Network Training Using Selective Data Sampling: Application to Hemorrhage Detection in Color Fundus Images". In: *IEEE Transactions on Medical Imaging* 35.5 (2016), pp. 1273–1284 (cit. on p. 32).

[81] Petar Veličkovic et al. "Graph Attention Networks". In: *arXiv preprint arXiv:1710.10903* (2017) (cit. on p. 64).

[82] Linnan Wang et al. "Accelerating Deep Neural Network Training with Inconsistent Stochastic Gradient Descent". In: *arXiv preprint arXiv:1603.05544* (2017) (cit. on p. 31).

[83] X Wang et al. "Ant Colony Optimization-Based Location-Aware Routing for Wireless Sensor Networks". In: *Wireless Algorithms, Systems, and Applications* (2008), pp. 109–120 (cit. on p. 25).

[84] Z Wang et al. "A modified ant colony optimization algorithm for network coding resource minimization". In: *IEEE Transactions on evolutionary computation* 20.3 (2015), pp. 325–342 (cit. on p. 25).

[85] Karl Weiss, Taghi M. Khoshgoftaar, and DingDing Wang. "A survey of transfer learning". In: *Journal of Big Data* 3.1 (Dec. 2016), p. 9 (cit. on p. 48).

[86] Tong Wu et al. "Representation Learning of EHR Data via Graph-Based Medical Entity Embedding". In: *arXiv preprint arXiv:1910.02574* (2019) (cit. on p. 73).

[87] Cao Xiao, Edward Choi, and Jimeng Sun. "Opportunities and challenges in developing deep learning models using electronic health records data: a systematic review." In: *Journal of the American Medical Informatics Association : JAMIA* 25.10 (2018), pp. 1419–1428 (cit. on p. 73).

[88] Xiuchun Xiao et al. "An Optimized Method for Bayesian Connectivity Change Point Model". In: *Journal of Computational Biology* 25.3 (Mar. 2018), pp. 337–347 (cit. on p. 8).

[89] Xiuchun Xiao et al. "Detecting Change Points in fMRI Data via Bayesian Inference and Genetic Algorithm Model". In: *ISBRA 2017: Bioinformatics Research and Applications*. 2017, pp. 314–324 (cit. on p. 8).

[90] Xueli Xiao et al. "Efficient Hyperparameter Optimization in Deep Learning Using a Variable Length Genetic Algorithm". In: *arXiv preprint arXiv:2006.12703* (2020) (cit. on p. 48).

[91] Jingyi Xu et al. "A Semantic Loss Function for Deep Learning with Symbolic Knowledge". In: *arXiv preprint arXiv:1711.11157* (2018) (cit. on p. 49).

[92] Ming Yan et al. "Deep Neural Networks with Short Circuits for Improved Gradient Learning". In: *arXiv preprint arXiv:2009.11719* (2020) (cit. on p. 48).

[93] Chengxi Ye et al. "On the Importance of Consistency in Training Deep Neural Networks". In: *arXiv preprint arXiv:1708.00631* (Aug. 2017) (cit. on pp. 28, 32, 34).

[94] Jason Yosinski et al. "How transferable are features in deep neural networks?" In: *Proceedings of the 27th International Conference on Neural Information Processing Systems*. Cambridge, MA, USA, 2014, pp. 3320–3328 (cit. on p. 32).

[95] Yang You, Igor Gitman, and Boris Ginsburg. "Large Batch Training of Convolutional Networks". In: *arXiv preprint arXiv:1708.03888* (Aug. 2017) (cit. on p. 32).

[96] Zhao You and Bo Xu. "Improving training time of deep neural networkwith asynchronous averaged stochastic gradient descent". In: *The 9th International Symposium on Chinese Spoken Language Processing* 1 (2014), pp. 446–449 (cit. on p. 31).

[97] Steven R. Young et al. "Optimizing deep learning hyper-parameters through an evolutionary algorithm". In: *Proceedings of the Workshop on Machine Learning in High-Performance Computing Environments - MLHPC '15*. Austin, Texas, USA, 2015, pp. 1–5 (cit. on pp. 6, 8–10, 22, 24, 48).

[98] Kezi Yu et al. "Rare Disease Detection by Sequence Modeling with Generative Adversarial Networks". In: *arXiv preprint arXiv:1907.01022* (2019) (cit. on p. 73).

[99] Zeng Yu et al. "Convolutional networks with cross-layer neurons for image recognition". In: *Information Sciences* 433-434 (Apr. 2018), pp. 241–254 (cit. on pp. 1, 48).

[100] Zeng Yu et al. "Reconstruction of hidden representation for robust feature extraction". In: *ACM Transactions on Intelligent Systems and Technology (TIST)* 10.2 (2019), pp. 1–24 (cit. on p. 48).

[101] Matthew D Zeiler. “ADADELTA: An Adaptive Learning Rate Method”. In: *arXiv preprint arXiv:1212.5701* (Dec. 2012) (cit. on p. 33).

[102] Xian Zeng et al. “Measure clinical drug–drug similarity using Electronic Medical Records”. In: *International Journal of Medical Informatics* 124.May 2018 (2019), pp. 97–103 (cit. on p. 73).

[103] Fan Zhang et al. “Predicting Treatment Initiation from Clinical Time Series Data via Graph-Augmented Time-Sensitive Model”. In: *arXiv preprint arXiv:1907.01099* (2019) (cit. on p. 73).

[104] Haiping Zhang et al. “Deep Learning Based Drug Screening for Novel Coronavirus 2019-nCov”. In: *Interdisciplinary sciences, computational life sciences* 12.3 (2020), pp. 368–376 (cit. on p. 1).

[105] Huishuai Zhang, Wei Chen, and Tie-Yan Liu. “Train Feedfoward Neural Network with Layer-wise Adaptive Rate via Approximating Back-matching Propagation”. In: *arXiv preprint arXiv:1802.09750* (Feb. 2018) (cit. on pp. 28, 32, 34).

[106] Junbo Zhang et al. “A comparison of parallel large-scale knowledge acquisition using rough set theory on different MapReduce runtime systems”. In: *International Journal of Approximate Reasoning* 55.3 (Mar. 2014), pp. 896–907 (cit. on p. 64).

[107] Junbo Zhang et al. “A Parallel Matrix-Based Method for Computing Approximations in Incomplete Information Systems”. In: *IEEE Transactions on Knowledge and Data Engineering* 27.2 (Feb. 2015), pp. 326–339 (cit. on p. 64).

[108] Junbo Zhang et al. “Efficient parallel boolean matrix based algorithms for computing composite rough set approximations”. In: *Information Sciences* 329 (Feb. 2016), pp. 287–302 (cit. on p. 64).

[109] Xin Zhang et al. “Diagnosis of COVID-19 pneumonia via a novel deep neural architecture”. In: *Journal of Computer Science and Technology* (2021) (cit. on p. 1).

[110] Aleksandar Zlateski, Kisuk Lee, and H. Sebastian Seung. “ZNN - A Fast and Scalable Algorithm for Training 3D Convolutional Networks on Multi-core and Many-Core Shared Memory Machines”. In: *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. Chicago, IL, 2016, pp. 801–811 (cit. on p. 31).

[111] Barret Zoph and Quoc V. Le. “Neural Architecture Search with Reinforcement Learning”. In: *arXiv preprint arXiv:1611.01578* (Nov. 2017) (cit. on p. 9).