Efficient Neural Architecture Search for Language Modeling

MASTER THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

EMBEDDED SYSTEMS

by

Mingxi Li

Thesis Committee:

Chair:	Dr. Frans A. Oliehoek,	TU Delft, supervisor
Committee Member:	Dr. Wei Pan,	TU Delft
Committee Member:	Dr. Jan van Gemert,	TU Delft
Committee Member:	Ph.D. student Hongpeng Zhou,	TU Delft

Faculty EEMCS, Delft University of Technology
Delft, the Netherlands

Preface

I would like to thank my supervisor Frans Oliehoek for his guidance and many helpful comments and discussions through the whole process of this master thesis. Moreover, I would like to thank Wei Pan and Jan van Gemert for agreeing to sit in my defense committee. And I would like to thank my daily supervisor Hongpeng who always made time when I was in need of advice.

Mingxi Li
Delft, the Netherlands
July 22, 2019

Abstract

Neural networks have achieved great success in many difficult learning tasks like image classification, speech recognition and natural language processing. However, neural architectures are hard to design, which requires lots of knowledge and time of human experts. Therefore, there has been a growing interest in automating the process of designing neural architectures. Though these searched architectures have achieved competitive performance on various tasks, the efficiency of NAS still needs to be improved. Moreover, current neural architecture search approach disregards the dependency between a node and its predecessors and successors.

This thesis builds upon BayesNAS which employs the classic Bayesian learning method to search for CNN architectures, and extends it to the problem of neural architecture search for recurrent architectures. Hierarchical sparse priors are used to model the architecture parameters to alleviate the dependency issue. Since the update of posterior variance is based on Laplace approximation, an efficient method to compute the Hessian of recurrent layer is proposed. We can find candidated architectures after training the over-parameterized network for only one epoch. Our experiments on Penn Treebank and WikiText-2 show that competitive architectures can be found in 0.3 GPU days using a single GPU for language modeling task. We find that our algorithm is more efficient than state-of-the-art.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Learning has achieved great success in various tasks, such as image classification, natural language processing and speech recognition over the last years. The powerful ability that deep neural networks can achieve state-of-the-art results on various competitive benchmarks derives from the featurea extraction stages. Thanks to the ImageNet [17] and GPU, the access to large amount of data and great improvements in the computation ability of the hardware have accelerated the research in deep learning. Since the first powerful model, AlexNet [33], was found, many innovative architectures has been proposed, like ResNet [24], DenseNet [27], and LSTM [26].

However, the ability of deep neural networks still depends on the design of neural architectures which requires substantial effort of human experts. There is a great demand for architecture engineering, where complex architectures of neural networks are designed manually. Therefore, developing algorithms to search for novel architectures automatically instead of the manual process of design has attracted more and more interests. Neural Architecture Search (NAS), which is the process of automating architecture engineering, has become a hot topic in the field of automated machine learning.

In this thesis, an efficient approach is proposed to automate the process of architecture engineering based on classic Bayesian learning. The approach is a one-shot based NAS which can be treated as a network pruning problem on the architecture parameters from an over-parameterized network. The search time of a one-shot neural architecture search can be significantly reduced because it does not need any seperate training. While some earlier work [66, 50, 38, 39, 13, 5] has proposed some promising approaches, there is one issue associated with most of them. The the dependency between a node and its predecessors and successors are often disregarded. The dependency logic is first considered and encoded in [64] which is based on Bayesian learning. However, this work is only able to design convolutional cells on CIFAR-10 for image classification. Therefore it makes sense to employ Bayesian learning approach to neural architecture search for a task which requires recurrent architectures.

The goal of this thesis is to apply Bayesian learning approach to automate the search process of novel architectures for word-level language modeling task, which predicts the probability that a given word appears next after a given sequence of words.

1.1 Research Questions and Contributions

In the search space for recurrent architectures of some previous work [66, 50, 38, 39], batch normalization is enabled during the architecture search stage to avoid gradient issues. However, it is not normal to use batch normalization in recurrent architectures as the statistics are computed per batch, which does not consider the recurrent part of the network. The statistics of all time steps are the same, which tend to be different. Therefore, the first research question:

Research Question 1. How can batch normalization [28] influence the architecture search? Moreover, how can the behaviour of the algorithm be influenced if we use other normalization methods, for example, layer normalization [30] which is widely used in RNN?

Previous work [64] showed that Bayesian learning approach could be used to design convolutional cells automatically. It assigns a probability distribution over neural networks and uses Laplace approximation [40, 48] to perform the inference. Since Laplace approximation requires computation of Hessian which costs an intensive computation burden in large networks, they propose an efficient method to compute Hessian. However, it is only applicable to convolutional layers. Inspired by [9] which proposed an efficient method to compute the Hessian of fully connected layer, the next research question is:

Research Question 2. Can this method be extended to recurrent layers to make classic Bayesian learning approach applicable to design recurrent architectures automatically for language modeling task?

The classic Bayesian learning approach [40, 48] prevents overfitting and promotes sparsity by specifying sparse priors. Additionally, as [15] has established that recurrent neural network models with more parameters can learn to store more information, the following research question rises:

Research Question 3. Can we generate novel recurrent architectures that can tradeoff between the performance and model size?

Thus, the contribution of this thesis is:

1. An efficient method to compute the Hessian of recurrent layers is proposed based on earlier work [9].

2. research on how can normalization in the search space influence the performance.

3. research on applying classic Bayesian learning approach to the problem of designing recurrent architectures by automation.

1.2 Outline

This section provides an outline of the rest of the thesis. Chapter 2 introduces the necessary background information for deep learning, the approaches used in neural architecture search and Bayesian learning. Chapter 3 presents methodology used for recurrent architectures. Chapter 4 presents an efficient computation of Hessian in recurrent neural network. Chapter 5 presents the details of experiments and results analysis. Chapter 6 concludes and suggests directions for future work.

Chapter 2

Background

Before starting to investigate the research questions proposed in Section 1, it is necessary to introduce background knowledge needed. Section 2.1 introduces some concepts in deep learning. Section 2.2 introduces an overview of the language modeling task. The next section presents existing methods of neural architecture search. Since our approach is based on BayesNAS which employs Bayesian learning, Section 2.4 introduces necessary background knowledge for understanding Bayesian learning. Finally section 2.5 presents Laplace approximation, an approximate inference methods that is widely used in Bayesian learning.

2.1 Deep Neural Network

Neural networks are composed of layers of connective units called artificial neurons (Figure 2.1). Usually a neural network is referred to as a "shallow network" if it has one input layer, one output layer and at most one hidden layer without recurrent connections. If we increase the number of layers or connect the layers recursively, the depth of the network is increased and this kind of neural network is referred to as "deep neural network" (DNN). Depends on how layers are connected in the network, there two kinds of deep neural network, Convolutional Neural Networks (CNN) and Recurrent Neural Networks (RNN).

2.1.1 Convolutional Neural Network

Convolutional neural network is a methodology which is often used to perform computer vision tasks, such as image classification [33, 24, 27] and object detection [21, 20, 54]. The design of CNN was inspired by how neurons of animals are connected. A convolutional neural network is composed of several building blocks, such as convolutional layer, pooling layer and fully connected layer. Basically each convolutional layer in the CNN performs an operation called convolution to the inputs with convolutional kernels. In order to learn hierarchical representations of features from the data, nonlinearity is introduced by passing features to non-linear activation function, such as tanh and Relu. To reduce the dimensions of features, the pooling layer combines multiple outputs of neurons of one layer into a single neuron of the next layer. In fully connected layer, each neuron is connected to all neurons in another layer. Figure 2.2 shows a example of convolutional neural network.

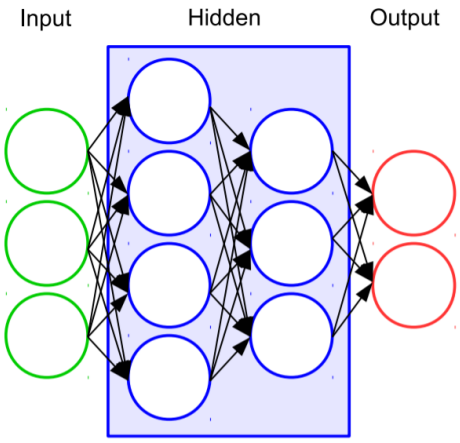


Figure 2.1: *A simple neural network, consisting of an input layer, an output layer and two hidden layers*

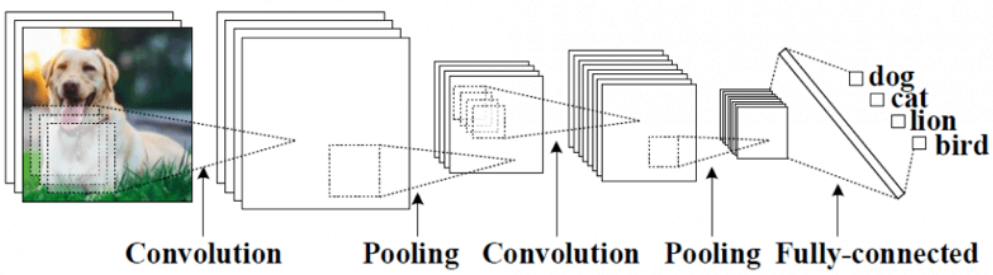


Figure 2.2: *Typical CNN architecture*

2.1.2 Recurrent Neural Network

Deep neural network with recurrent connections are usually called recurrent neural network. In RNN, neurons have at least one feedback loop, which are recurent cycles over sequence (time). RNN are used to process sequential data because the recurrent connection and hidden states work as the memory in the network, which makes them applicable to tasks such as handwriting recognition [56] and speech recognition [35].

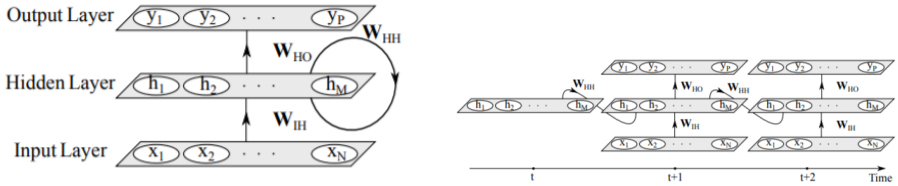
Model Architecture A simple RNN has three blocks, the input, hidden and output layers. The input layer is a N-dimension vector which has N input units. The inputs to the input layer are a sequence of vectors $(...,x_{t-1},x_t,x_{t+1},...)$ where t is the time step and $x_t = (x_1,...,x_N)$. The input units are connected to the M hidden units in the hidden layer $h_t = (h_1,...,h_M)$ with weight matrix W_{IH}. The hidden layer works as the memory of the network

$$h_t = f_H(W_{IH}x_t + W_{HH}h_{t-1} + b_H) \tag{2.1}$$

The output is computed as

$$y_t = f_O(W_{HO}h_t + b_O) \tag{2.2}$$

Equation 2.1 and Equation 2.2 show the non-linear state equations of RNN is iterable through time. In each time step, the hidden layer predicts the output based on the input vector and the information about the past hidden states of the network over many time steps. Therefore, we can always replace an unrolled RNN over time with a feedforward neural network as shown in Figure 2.3.



(a) *Folded RNN* **(b)** *Unfolded RNN through time*

Figure 2.3: *An example of a simple RNN structure and its unfolded architecture throught sequence t. Each arrow represents a fully conncetions of units between layers*

2.1.3 Batch Normalization

Training a deep neural network is complicated. The hyper-parameters of a model need to be tuned carefully, like the learning rate and the initial values of network parameters. During the training of deep neural networks, the distribution of the inputs of each layer can easily change because the inputs to each layer are affected by all its preceding layers. The change in the distributions of internal nodes of a deep network is referred as Internal Covariate Shift [28].

Internal covariate shift presents a problem that the layers need to continuously adapt to the new distribution. Consider a network computing the loss l as:

$$l = F_2(F_1(x, \theta_1), \theta_2) \tag{2.3}$$

where x is the input vector, F_1 and F_2 are transformations and θ_1 and θ_2 are parameters to be learned. This can be viewed as if the outputs of the first layer $u = F_1(x, \theta_1)$ are fed into sub-network. Therefore, the loss can be computed as:

$$l = F_2(u, \theta_2) \tag{2.4}$$

The input distribution properties that accelerate the training process apply to the training of sub-network. Therefore, if the distribution of the u is fixed, θ_2 does not have re-adjust to compensate for the change of the distribution of u.

Fixed-distribution of inputs has other positive consequences. Consider a layer with sigmoid activation function $z = sigmoid(Wu + b)$ where u is the input of this layer, W and b are network parameters to be learned. The gradient flowing down to u tends to be zero with $Wu + b$ increasing, which may cause gradient vanishing problem and the training process is slowed down. If the distribution of the inputs of layers could be stable, the optimizer would be less likely to get stuck in this situation.

Batch normalization [28] is proposed to eliminate internal covariate shift and speed up the training process. They first normalize each scalar feature independently, which

means that they have zero mean and variance of 1. For a layer with d-dimensional input $\mathrm{x} = (x^{(1)}, ..., x^{(d)})$, each dimension can be normalized by:

$$\hat{x^{(k)}} = \frac{x^{(k)} - \mathbf{E}[x^{(k)}]}{\sqrt{\mathbf{Var}[x^{(k)}]}} \tag{2.5}$$

where the expectation and variance are computed over the training data set as shown in [34]. However, such simple normalization can change what the layer can represent. For example, if we normalize the inputs to a sigmoid function, then the output would be bound to the linear region only. To make the transformation in the network can represent the original transform. Thus for each activation $x^{(k)}$, two learnable parameters are introduced, $\gamma^{(k)}$ and $\beta^{(k)}$. The output is the results of scaling and shifting the normalized value:

$$y^{(k)} = \gamma^{(k)} \hat{x}^{(k)} + \beta^{(k)} \tag{2.6}$$

In practice, when using mini-batch stochastic optimization, this process can be simplified. The mean and variance are estimated based on each mini-batch. Assume that the size of a mini-batch $\mathcal{B}$ is m. Therefore, for $x^{(k)}$, we have m values:

$$\mathcal{B} = \{x_1^{(k)}, ..., x_m^{(k)}\} \tag{2.7}$$

Since the normalization of each dimension is independent, we focus on this particular dimension and omit k for clarity. Therefore, $\mathcal{B} = \{x_1, ..., x_m\}$. The algorithm is presented in Algorithm 1.

Algorithm 1 Batch normalization

Input: Values of x over a mini-batch: $\mathcal{B} = x_{1...m}$; Parameters to be learned: γ, β
Output: $\{y_i = \mathbf{BN}_{\gamma,\beta}(x_i)\}$

$\mu_{\mathcal{B}} \leftarrow \frac{1}{m} \sum_{i=1}^{m} x_i$
$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m} \sum_{i=1}^{m} (x_i - \mu_{\mathcal{B}})^2$
$\hat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 - \varepsilon}}$
$y_i \leftarrow \gamma \hat{x}_i + \beta \equiv \mathbf{BN}_{\gamma,\beta}(x_i)$

During inference, we do not want to use the statistics of each mini-batch. Instead we use the normalization

$$\hat{x} = \frac{x - \mathbf{E}[x]}{\sqrt{\mathbf{Var}[x]}} \tag{2.8}$$

over the entire dataset. The expectation and variance are over all training mini-batches of size m:

$$\mathbf{E}[x] = \mathbf{E}_{\mathcal{B}}[\mu_{\mathcal{B}}] \tag{2.9}$$

and

$$\mathbf{Var}[x] = \frac{m}{m-1} \mathbf{E}_{\mathcal{B}}[\gamma_{\mathcal{B}}^2] \tag{2.10}$$

where $\mu_{\mathcal{B}}$ and $\gamma_{\mathcal{B}}^2$ is the means and variances produced by each mini-batch respectively.

Batch normalization has several advantages by normalizing activations. It makes the model less sensitive to the learning rate and scale of network parameters which

may result in gradient explosion and vanishing. Also, since a training example is seen in conjunction with other examples in the mini-batch, this effect promotes the generalization of the model [28].

2.1.4 Layer Normalization

Though batch normalization reduces the training time by normalizing the inputs of each layer, it has several limitations. The first is that batch normalization estimates the mean and variance using mini-batch statistics. Since smaller mini-batch sizes increase the variance of these estimates, we have to be careful about the batch size when batch normalization is performed using SGD. Additionally, the recurrent activations of each hidden layer will have different statistics, which makes it difficult to apply batch normalization in recurrent neural network. This can be solved by storing the statistics separately for each hidden layer. However, batch normalization requires running averages of the summed input statistics [28]. It can be problem if a test sequence is longer than any of the training sequences because the statistics of some time steps are not estimated during training. Layer normalization [30] is a simple method to improve the training speed of recurrent neural network models. It estimates the normalization statistics from the summed inputs to a layer at the current time-step directly.

In a feedforward neural network, the changes in the output of one layer will cause changes in the summed inputs of this layer. Thus the covariate shift can be reduced by fixing the mean and variance of the summed inputs of a layer. In this case, batch normalization is transposed into layer normalization [30]. Layer normalization computes the mean and variance used for normalization from all of the summed inputs to the neurons in a layer as follows:

$$\mu^l = \frac{1}{H}\sum_{i=1}^{H} a_i^l \tag{2.11}$$

$$\sigma^l = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^l - \mu^l)^2} \tag{2.12}$$

where H represents the number of hidden units in a layer.

In case of a standard RNN, the summed inputs to a recurrent layer can be computed as:

$$a^t = W_{HH}h^{t-1} + W_{IH}x^t \tag{2.13}$$

where x^t is the current input and h^{t-1} is the previous hidden state. W_{HH} is the the recurrent hidden to hidden weights and W_{IH} is the input to hidden weights. Then we can normalize the recurrent layer using normalization terms similar to Equation 2.11 and Equation 2.12:

$$\mu^t = \frac{1}{H}\sum_{i=1}^{H} a_i^t \tag{2.14}$$

$$\sigma^t = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^t - \mu^t)^2} \tag{2.15}$$

In a standard RNN, the summed inputs of the recurrent layers tend to grow or shrink at each time step. It leads to exploding and vanishing gradients which makes

it hard to train recurrent neural network models. In a layer normalized RNN, the normalization terms makes it possible to re-scale all of the summed inputs to a layer. Therefore, the recurrent neural network models can be more stable.

2.2 Language Modeling

In this section, we first presents an overview of the task of statistical language modeling. Then we discuss the application of recurrent neural network to language modeling and some drawbacks of current RNN models.

2.2.1 Statistical language modeling

Statistical language modeling aims to estimate the probability distribution of various linguistic units, such as words, sentences and whole documents [55]. The applications of language modling include speech recognition [29], text generation [16] and machine translation [11].

A traditional language modeling task is word-level, which is also our target task. The goal is to model the probability that a given word appears next after a given sequence of words. The most common measure to evaluate language models is perplexity (PPL). It is the inverse of the geometric average probability assigned by the model to each word in the test dataset. The training data of language modeling is often taken from large and structured sets of texts, which are also know as corpora. The Wikicorpus and the Brown corpus are two frequently used corpora.

N-gram model [29] is one of the earliest techniques to model the probability of observing a word after several previous words. The N represents the number of previous words which are considered plus the next word in the sequence. Thus N - 1 is the number of words that the model uses the predict the probability of the next word. The probability is estimated as the number of times that the given sequence augmented with the given word appears in the training divided by the number of times that the given sequence is present in the training data. In practice, the non-zero probabilities are assigned to the words that do not appear in the training data to smooth the probability distribution.

Some more models were developed later which estimated the probability based on the incorporation of various features. Then neural network models found their way into the language modeling domain with the work [6]. It used a feedforward neural network to learn a vector which represents every word in a predefined vocabulary and predict the next word in a sentence. The ability of neural networks project the vocabulary into hidden layers can explain why neural networks can provide better estimates for *N*-grams. And that is why neural networks has driven much of the recent interest in the domain of language modeling.

The text documents in corpora need some preprocessing before they can be used by neural network. A vocabulary V needs to be constructed, which contains all distinct words in a corpus. It requires the splitting of the corpus into words. The word **UNK** is often used to represent all words that are not in the vocabulary but in the test dataset. Assume the label of **UNK** is w_1 and the labels of all the other words $w_2, ..., w_N$. Thus we have $V = (w_1, w_2, ..., w_N)$. Each word in the vocabulary is represented by a vector of N components via function τ. For example, $\tau(w_1) = (1, 0, ..., 0)$. Assume there are

n words in the corpus. Then the corpus can be represented by a sequence of vectors, $D = (w_1, ..., w_n)$, by replacing each word by its corresponding coding vector.

2.2.2 Application of RNN to language modeling

Though FNN has show that neural networks can be used for language modeling, it has one drawback that only a fixed number of words can be considered to predict the next word. In other words, the structure of FNN lacks of memory. Only the words that are presented by a fixed number of neurons are taken into account. All words that are presented earlier are forgotten, even if they may be essential to determine the next word. In this case, recurrent neural network was used to introduce memory to overcome the limited fixed context length. The inputs of RNN are a sequence of coding vectors of words from the vocabulary. The softmax can be used as the output function to ensure that the outputs can be interpreted as the probabilities of apperance of each word in the vocabulary V given inputs.

However, the standart RNN has several drawbacks:

1. Training time. Training RNN is known to be slow. And the size of vocabulary V is very large for many language applications, which increases the real complexity of training.

2. Short context length in practice. Although the context length can be unlimited in theory, the range of context that is accessed in practice is limited due to the vanishing and exploding gradient problem [26].

There are several methods to reduce the training time. One obvious way is to limit the size of vocabulary V. It can be achieved by replacing the least frequent words with unkown word **UNK**. However, a compromise has to be made between the performance and training time. To increase the context size of RNN, we can use the long short-term memory (LSTM) [26]. It extends the standart RNN by replacing each hidden unit by a memory block. Each block contains self-connected memory cells and three gates, the input, output and forget gates. They provide continuous analogues of write, read and reset for the cells. Then the memory cells can store and access information over many training examples.

2.3 Neural Architecture Search

Neural Architecture Search (NAS) a kind of algorithms which automate the manual design of neural architectures [66, 50, 38], which is a subfield of Auto Machine Learning. We can denote NAS in three dimensions: search space, search strategy and performance estimation strategy. Basically search space defines which kind of architectures can be generated theoretically. Search strategy defines how to search for an architecture in the search space. For a generated architecture, we need to estimate its performance because the target is to find architectures with high performance on test data. Figure 2.4 illustrates how NAS methods work [59].

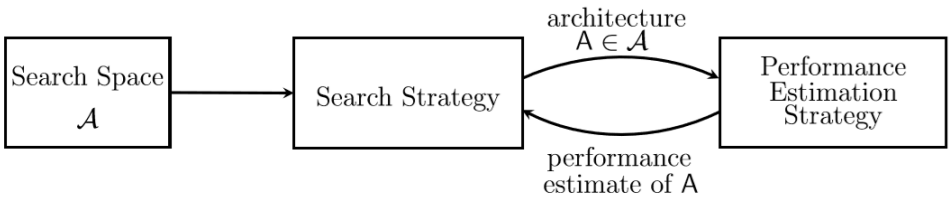


Figure 2.4: *Abstract illustration of Neural Architecture Search methods*

2.3.1 Search Space Design

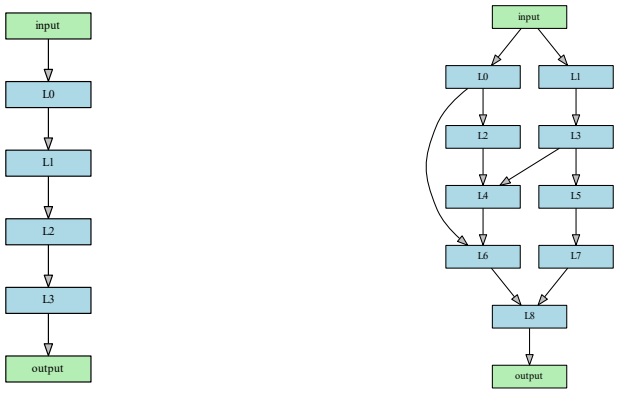
Search space determines which architectures can be searched in principle. The settings of search space have a strong influence on the search cost and the performance of the learned architectures.

Chain-Structured Search Space The space of chain-structured neural networks is a relatively basic search space as illustrated in Figure 2.5(a). We can consider the structure of the chain structure of the neural network as a neural network with n sequential layers, where the output of each layer serves as the input of its successors. Therefore, the architectures could be represented by: i) the maximum number of layers n; ii) types of operations that can be performed at each layer; and iii) hyperparameters associated with the operation.

Some previous works include [2] explore this search space. Their operations set inclueds convolutions, pooling, linear transformations (dense layers) with activation, and global average pooling. They introduce different hyperparameter settings such as number of filters, kernel size, stride and pooling size. They also add constraints that exclude certain architectures. For example, they do not consider architectures with pooling as the first operation. Additionally, architectures with feature transformations by fully connected layer before convolutions are also excluded.

Incorporating previous knowledge about hand-built architectures for some specific tasks is widely used in recent NAS work ([67]; [12]; [66]). It allows to build more complex networks with hand-crafted elements such as skip connections. Such settings allow for higher degrees of freedom. Figure 2.5(b) shows an example of a more complex search space with multiple branches and skip connections. [66] permits skip connections between the ordered nodes of a chain-structured architecture. However, only convolutions with different hyperparameters are included in the set of operations. If nodes have multiple inputs, the combination operation is fixed as concatenation.

[60] uses summation instead of concatenation in their search space. Furthermore, architectures are seperated into sequentially connected segments in their search space. Each segment is a set of nodes with convolutions as their operation. Note that each segment begins with a convolution and concludes with a max pooling. The maximum number of convolution operations is also fixed for each segment. [57] consider a similar search space. Architectures are also structured by sequential segments. Each segment has repeating patterns of operations. We can select operations and the number of repetitions. Note that they allow segments to be different.



(a) **(b)**

Figure 2.5: *An illustration of chain-structured search space. Each node in the graphs corresponds to a layer in a neural network. Figure 2.5(a) is an element of a basic chain-structured space. Each layer has single input and single output. Figure 2.5(b) shows an example of a more complex search space with multiple branches and skip connections.*

Cell Based Search Space Inspired by the fact that many hand-crafted architectures consists of repeated motifs ([24]), [67] and [12] propose to search for such cells instead of searching for the whole architectures. Such architectures often consist of smaller-sized graphs that are stacked to form a larger architecture.

NASNet [67] is one of the first that uses cell based search space as shown in Figure 2.6. It learns two kinds of cells as shown in Figure 2.7(a) and Figure 2.7(b): a normal cell which keeps the dimensionality of the input and a reduction which reduces the spatial dimension. Then the final learned architecture is based on stacking these cells in a predefined manner. They include concatenation as a possible merge operation. However, in their experiments they find that architectures with summation operation outperform those with concatenation operation in terms of accuracy. Therefore, [36] fix the merge operation to summation. A similar search space is proposed by [63]. Graphs comprising of arbitrary connections between different nodes are defined as cells. Also, they use fixed max-pooling layers instead of reduction cells for decreasing feature dimensions. The instances of cells differ in the type of operations and their input, which allow higher degrees of freedom than normal cells.

Cell based search space outperforms chain-structured search space in two main aspects. The first is the search cost is significantly reduced. Compared with searching for the whole architecture, the size of cell based search space can be comparably small. Also, the transferability of architectures learned based on chain-structured search space is weak. However we can easily transfer cells to other datasets by adapting the number of cells used within a model.

Summary The key design choice of search space is how much prior knowledge from human experts we should introduce. Introducing properties well-suited for specific tasks can reduce the size of the search space and simplify the search process. For

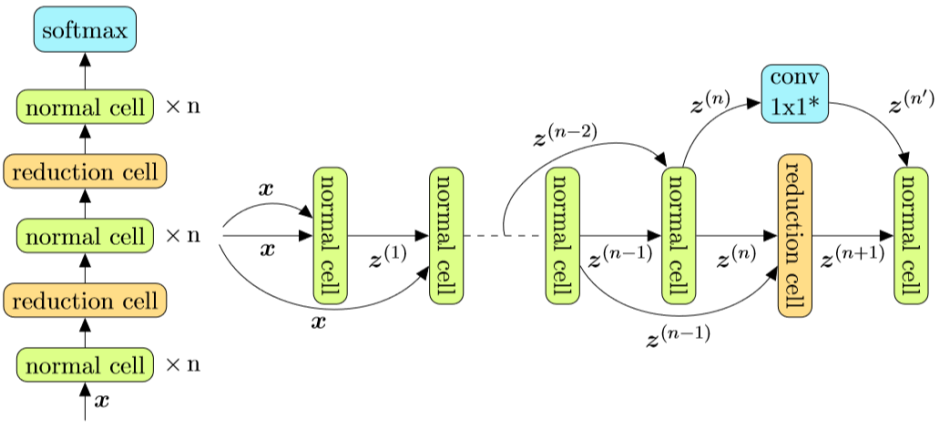
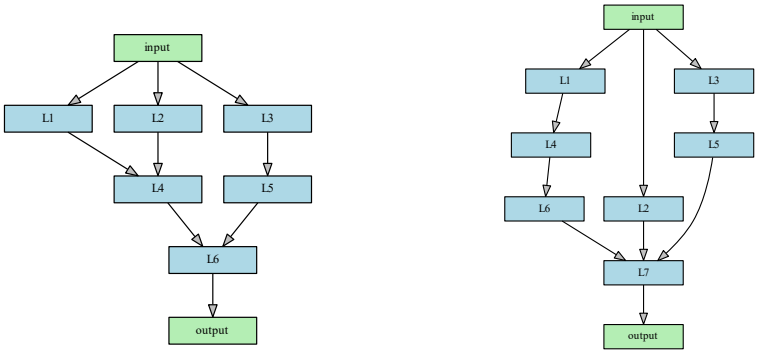


Figure 2.6: *LEFT: structure of the NASNet search space where n normal cells followed by a reduction cell. RIGHT: detailed view with the skip inputs. The 1*1 convolution is a special operation which converts ‡$^{(n)}$ to match the shape of ‡$^{(n+1)}$.*



(a) **(b)**

Figure 2.7: *An illustration of cell-based search space. Figure 2.7(a) and Figure 2.7(b) show an example of a normal cell and a reduction cell respectively. The final architecture is built by stacking the cells sequentially*

example, [12] and [67] incorporate modern design elements known from hand-crafted architectures such as skip connections to build more complex neural networks. [12] employs the high-level structure of well-known manually designed architectures, such as DenseNet ([27]), and use their cells within these models. By utilizing such prior knowledge, the search cost is reduced while achieving better performance.

However, this also introduces a human bias, which may prevent finding novel architectural building blocks that go beyond the current human knowledge. Consider the case when using cell-based search space. If how to connect the learned cells is decided by human experts, the search for the cell could become overly simple if most of the

complexity is already accounted for by the high-level architecture.

2.3.2 Search Strategy

Given a fixed search space, the search strategy defines how to explore it. There are many different search strategies which can be used, including reinforcement learning (RL), evolutionary methods and one-shot methods.

Reinforcement Learning Reinforcement Learning [31] is a family of algorithms to solve the problem that an agent has to learn its behaviour based on rewards of its interactions with the environment as shown in Figure 2.8. Formally, a reinforcement learning consists of a set of environment states S, a set of agent's behaviour B, a set of reinforcement signals R, an input function I and a state transition function T. In each step, the agent receives the indication i of the current state s. Then the agent learns to choose an action a based on some algorithms. The action changes the state of the environment. This transition is passed to the agent through a reinforcement signal r.

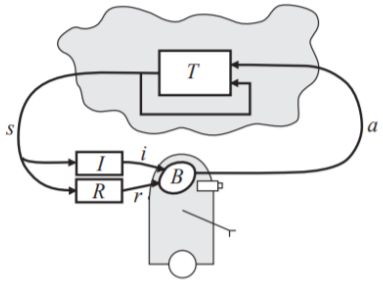


Figure 2.8: *The standard reinforcement-learning model.*

To frame neural architecture search as a reinforcement learning problem ([2]; [66]; [67]), the action space is considered to be identical to the search space. Therefore, when the agent takes actions, a neural architecture is generated. The performance of the learned architectures on unseen data can be used to estimate the reward of the agent.

Different RL methods have different strategies on how to represent agents and how to optimize them. [2] is one of the first to use RL-based algorithms for neural architecture search. Their algorithm is a combination of Q-learning, ε-greedy, and experience replay. The actions in their approach are the choice of different layers to add to an architecture. Also we have the option to terminate building the architecture. After initializing the action-value function, the agent samples a trajectory which comprises of multiple decision steps, eventually leading to a terminal state. The algorithm then trains the model corresponding to the trajectory and updates the action-value function as defined in the Q-learning algorithm.

[66] is the first to employ policy gradient methods. It generates the architecture description of the neural network by using a recurrent neural network (RNN) policy and trains the RNN with reinforcement learning based on the expected accuracy of the generated architecture on the validation dataset. Every action is sampled from the probability distribution implied by a softmax operation and then fed into the next time

step as input as shown in Figure 2.9. The RNN-controller in their approach samples layers which are sequentially appended to construct the final network. Finally they trained the network with the REINFORCE policy gradient algorithm and achieved very competitive performance on the CIFAR-10 and Penn Treebank benchmarks. After that NAS became a mainstream research topic in the machine learning community. In their follow-up work of scalable image recognition, Proximal Policy Optimization (PPO) is used. [2] generates architectures by sequentially sampling a layer's type and its corresponding hyperparameters with Q-learning.

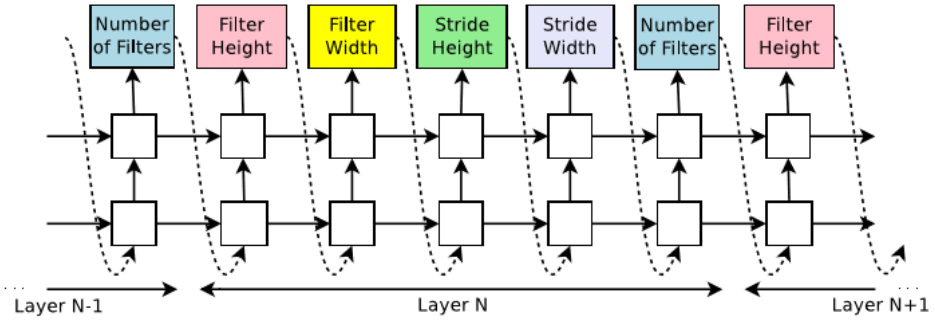


Figure 2.9: *The controller used by [66] to predict configuration of one layer*

Despite different strategies of these approaches, they can be considered as sequential decisions which samples actions to generate a architecture sequentially. All the previous decisions define an environment's state. However, during this decission process, there is no interaction with the environment and the agent can only obtain the reward after the final decision action. One alternative view is framing NAS as the sequential generation of a single action ([12]). The architecture is encoded into a fixed-length representation by a bi-directional LSTM. And the actor controller make a decision on which action to sample based on this representation.

Neuro-Evolutionary A neuro-evolutionary use evolutionary algorithms for generating the neural architecture. Evolutionary algorithms have a candidated pool of models, a set of neural netowrk architectures. Each model is assigned a fitness based on its performance. In every evolution step, some models with good fitness from the pooling are sampled. By applying mutations to these parents, these models generate descendants. Mutations are local operations including adding or removing a layer and generating new settings of the hyperparameters of a layer. Then the descendants are trained and evaluated based on their performance. The fitness is assigned. Models with poor fitness will be removed from the pool. The framework is shown in Figure 2.10.

In the context of neural architecture search, the population consists of candidated architectures. Several architectures are selected for mutation, which can generate new architectures in the search space. These new architectures are evaluated then for fitness. The process is repeated till convergency.

Tournament selection [22] is the most widely used parent selection method in neural architecture search. It first selects k individuals randomly. Then it iterates over

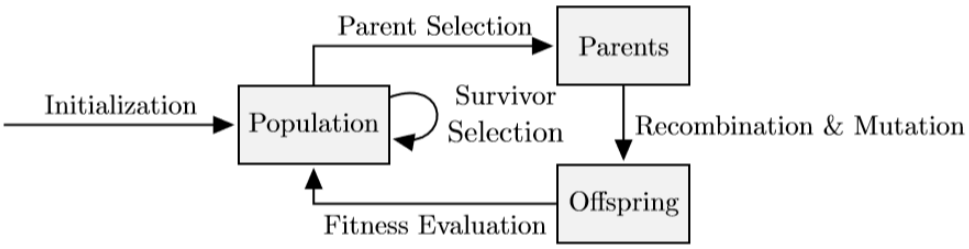


Figure 2.10: *A general framework for evolutionary algorithms*

them in the descending order of their fitness while selecting individuals for further steps with some high probability p.

[46] is the first approach which optimizes the neural architecture using evolutionary algorithms. They use evolutionary algorithms to generate neural architectures and optimize weights with backpropagation. In recent work ([52]; [53]; [37]; [18]), evolutionary algorithms are not used to optimize weights because the large size of parameters related to weights of neural networks. These neuro-evolutionary approaches use evolutionary algorithms to generate neural architectures and use gradient-based methods like SGD for optimizing weights.

[52] is one of the first to find CNN architectures for image classification based on evolutionary algorithm. In the parent selection step, they sample a pair of architectures. The architecture with better fitness is mutated, evaluated and added to the population. The other is removed. The set of mutations includes adding convolutions, altering kernel size, number of channels, stride and learning rate, adding or removing skip connections, removing convolutions, resetting weights, and a mutation that corresponds to no change. Since no further domain constraints are enforced, architectures with redundant components can be potentially sampled in this approach as shown in Figure 2.11.

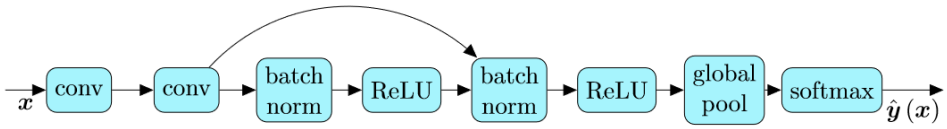


Figure 2.11: *A possible architecture discovered by the algorithm described by [52]*

[37] uses the hierarchically organized search space along with evolutionary algorithm. Mutations modify the representations at every step. The population is initialized with 200 trivial genotypes which are diversified by applying 1000 random mutations. In their setting, mutations can add, alter and remove operations from the genotype.

[53] is one of the most significant works in the direction of using evolutionary algorithms for architecture search. They select parents with tournament selection based

on the NASNet search space in each iteration. Mutation includes a random change in an operation or a connection in either a normal or a reduction cell as shown in Figure 2.12. Then the new architectures are trained for 25 epochs. They add a regularization term to the objective function to make sure that generated architectures are not only capable of reaching high validation accuracy once but every time.

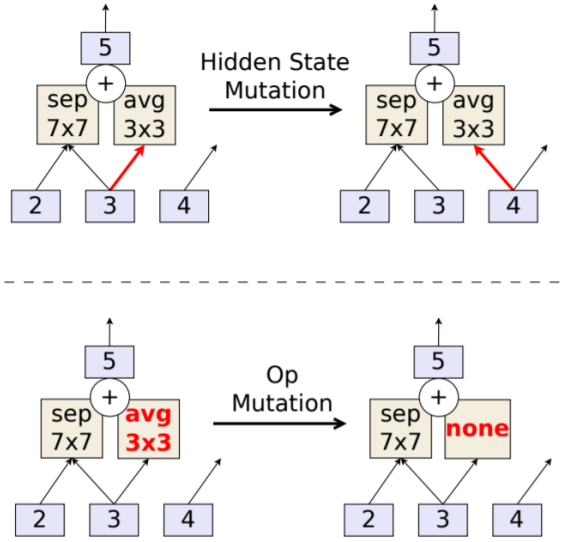


Figure 2.12: *Illustration of the two mutation types*

One-Shot One-shot architecture search is another mainstream approach for NAS. The essential idea of the one-shot algorithm is to treat all candidated architectures as subgraphs of a supergraph ([10]; [5]; [50]; [38]). The weights of edges are shared among different subgraphs if they have edges in common in the supergraph. Therefore, only the supergraph needs to be trained because the weights of edges are shared among candidated architectures. The generated architectures can be evaluated without any separate training which could speed up performance estimation.

ENAS ([50]) uses a RNN controller to sample candidated architectures from the search space and train the one-shot model based on approximate gradients obtained through REINFORCE which is similar to [66]. DARTS ([38]) places a mixture of candidate operations on each edge of the supergraph. It proposes a method for continuous relaxation of the search space. Then the weigths of all edges could be jointly trained based on it.

Darts [38] propose a method based on differentiable loss function which makes all parameters differentiable. Each node $x^{(i)}$ represents a feature map and each directed edge (i, j) is associated with some operation $o^{(i,j)}$. Let O be a set of candidated operations (e.g., convolution, max pooling, zero) that can be applied to x^i. To make the search space continuous, they relax the categorical choice of a particular operation to

a softmax over all possible operations:

$$\overline{o}^{(i,j)}(x) = \sum_{o \in O} \frac{exp(\alpha_o^{(i,j)})}{\sum_{o' \in O} exp(\alpha_{o'}^{(i,j)})} o(x) \tag{2.16}$$

where the operation mixing weights for a pair of nodes (i, j) are parameterized by a vector α of dimension $|O|$. The task of architecture search then reduces to learning the set of continuous variables. At the end of search, a discrete architecture can be obtained by replacing each mixed operation $\overline{o}^{i,j}$ with the most likely operation (Figure 2.13):

$$o^{(i,j)} = \text{argmax}_{o \in O} \alpha_o^{(i,j)} \tag{2.17}$$

The network parameters and architecture parameters can be learned by minimizing the loss on the training set and loss on the validation by gradient-based optimization methods.

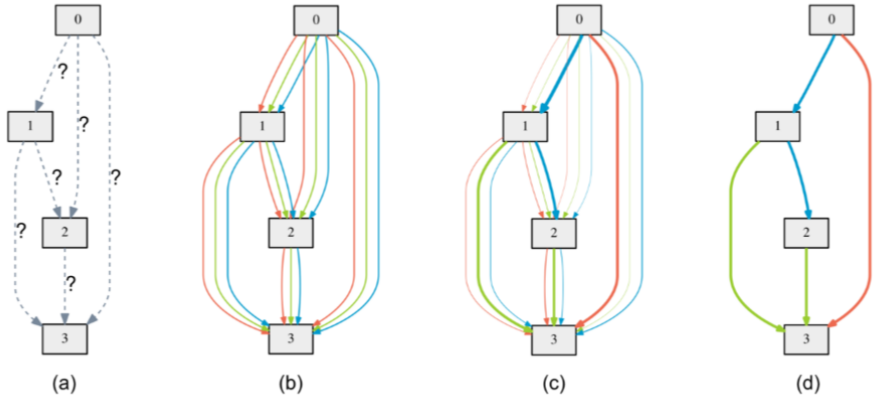


Figure 2.13: *An overview of DARTS: (a) Operations on the edges are initially unknown. (b) Continuous relaxation of the search space by placing a mixture of candidate operations on each edge. (c) Joint optimization of the mixing probabilities and the network weights. (d) Inducing the final architecture from the learned mixing probabilities.*

However, one disadvantage is that all parameters must be kept in memory all the time. Since the super-graph is large, this is a significant drawback. To overcome this problem, [13] introduce update rules that keep only a part of the network in memory. They use binary gate a part of the path based on a learned probability. The memory load is significantly reduced because we do not need to cash the entire over-parameterized model.

Though one-shot algorithm greatly speeds up performance estimation of architectures, one general limitation of one-shot NAS is that the architecture of the supergraph actually restricts the search space of its subgraph. Therefore, the performance of one-shot NAS relies on a good design of the search space. Moreover, the size of the super-graph will be relatively small because the entire supergraph needs to be cashed in the GPU memory during the architecture search process.

Bayesian Approach Since this thesis is based on BayesNAS [64], which employs Bayesian learning, a brief introduction of BayesNAS is presented. It treats neural architecture search problem as a netowrk compression problem on the architecture parameters form an over-parameterized network. The main idea is to use Hierarchical sparse priors to model the architecture parameters. BayesNAS ([64]) uses the priors to model the dependency between a node and its predecessors and successors, which is often disregarded by current NAS approaches. This is mainly the consequence of improper zero operations. Consider the case that all operations from a node's predecessors are zero, which means that this node is disconnected with all the previous nodes. However, there is the case that the child network still has other non-zero edges to keep it connected as shown in Figure 2.14. For a complex search space, it would take extra cost to safely check and remove such isolated nodes from the graph. For each w_{ij}^{o}, they introduce a variable, switch s_{ij}^{o} which has two states {ON, OFF}. Then they assign a probability distribution to encode this logic, for example Gaussian distribution, over $w_{ij}^{o'}$. If $s_{ii'}$ is OFF, $s_{ii'} = 0$, γ_{jk}^{o} will always be a small value close to zero. which means that we have great confidence that the corresponding weight w_{jk} is zero. Therefore, the super-graph can be pruned based on γ. Bayesian approach shares the advantage of Bayesian learning. It prevents overfitting and promotes sparsity by specifying sparse priors.

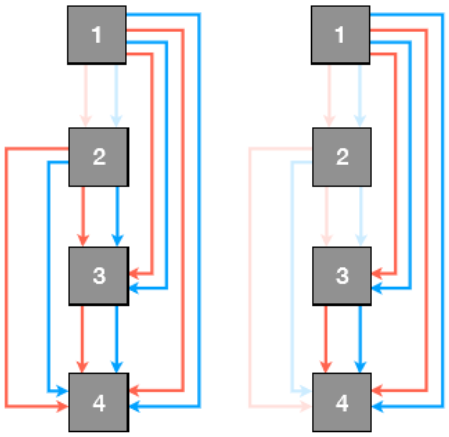


Figure 2.14: *An example of the problem caused by disregarding dependency. Left: The isolated node 2 should be removed from the graph but there are still edges to keep it connected. Right: Expected connected graph with no connection from node 2 to 3 and from node 2 to 4*

2.3.3 Performance Estimation Strategy

The search strategy of NAS defines how to generate an architecture that maximizes the evaluated performance. To guide these approaches, the performance of architectures on unseen dataset needs to be estimated. We can always train architectures on training dataset from scratch and report their performance on validataion dataset, which is the simplest option. However, the computation cost can be expensive and the number of architectures that can be explored will be limited. For example, the computation cost

of NAS ([66]) is in the order of thousands of GPU days. Therefore, some recent work focuses on methods that could estimate the performance without much cost.

Estimation Based on Lower Fidelities The computation cost of performance estimation can be reduced by validating architectures based on lower fidelities of the actual performance after normal training. Low-fidelity approximations include shorter training time ([67]), training on a subset of data ([32]), and on lower-resolution images ([14]). However, estimation based on lower fidelities can introduce bias because the performance is typically underestimated. Even search strategies which ranks all candidated architectures may not solve this problem. [62] indicates that the relative ranking is not stable if the difference between the low-fidelity approximations and the real evaluation is too big.

Estimation Based on Extrapolation Some recent works propose another possible way of estimating an architecture's performance based on extrapolation. [3] trains a prediction network with initial learning curves. Then it will predict the converged performance based on the learning curve for a predefined number of epochs. The architectures which are predicted to perform poorly will be terminated to speed up the estimation process. [36] trains the prediction network with architectural/cell properties instead of learning curve and extrapolate to architectures/cells with larger size than architectures during training.

Estimation Based on Weight Sharing Weight sharing is another promising approach to speed up performance estimation. The main idea is to reuse weights between different architectures as shown in Figure 2.15. [50] formulates the NAS problem as searching for subgraphs in a super directed acyclic graph. The weights corresponding to each operation are shared between different subgraphs. This significantly speed up the estimation process as we only need to train the weights of the supergraph. The generated architectures do not need to be trained because they can inherit trained weights from the supergraph. However, weight sharing may also introduce biases into the architecture search. Consider the case that there is an initail bias in exploring some parts of the search space more than others. The weights of the supergraph will be better adapted for these architectures. And in return the bias of the search to these parts of the search space would be reinforced. This may lead to premature convergence of NAS to a region of suboptimal architectures.

2.4 Bayesian Learning

Bayesian learning [40, 7, 48] is a particular set of approaches to probabilistic machine learning. Bayesian learning treats model parameters as random variables. To define a model, a generative process is provided describing how the data was created, which also includes the known model parameters. The initial beliefs about parameters are the distributions over possible values that the parameters might take [48, 7]. Data is viewed as observations from the generative process. And the beliefs about parameters will be updated based on observations, which means that a new distribution over the parameters will be assigned. Bayesian learning uses Bayes' theorem to determine the

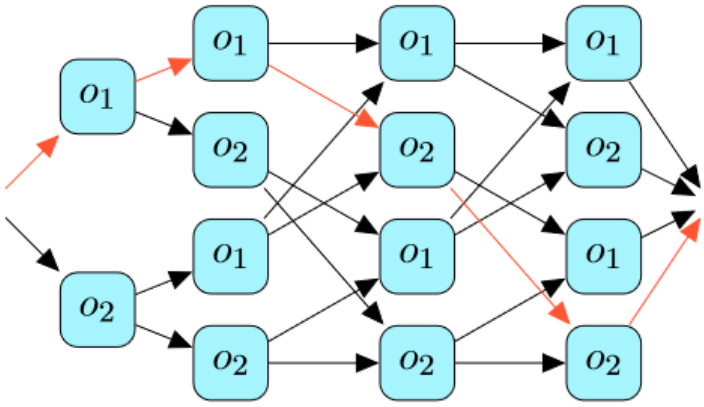


Figure 2.15: *Example for weight sharing with only two operations in the chain-structured search space. Each box has its own weights, every path is one architecture in the search space (e.g. the red path). Thus, weights are shared across different architectures.*

conditional probability of a hypotheses given some evidence or observations. In this section, several key concepts related to Bayesian learning are explained.

2.4.1 Bayes' Theorem

The main idea of Bayes' theorem is that we can calculate the probability of an event given prior knowledge of some conditions related to the event [7, 48]. Equation 2.18 states Bayes' theorem mathematically.

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)} \tag{2.18}$$

where A and B are events and $P(B) \neq 0$. $P(A|B)$ and $P(B|A)$ are the likelihood of a event occuring when another event is true. $P(A)$ and $P(B)$ are often called marginal likelihood, which represents the probabilities of observing A and B independently of each other.

2.4.2 Bayesian Inference

Given Bayes' theorem, we can computes the posterior probability based on a prior probability and a likelihood function derived from observed data [40]. Assume that:

- x: a data point
- θ: the parameter of data distribution
- α: the hyperparameter of parameters' distribution
- $\mathbf{X}$: the set of observed data points, $x_1, ..., x_n$
- $\tilde{x}$: a new data point with distribution to be predicted

Therefore, the posterior distribution can be seen as the distribution of parameter θ after observing $\mathbf{X}$, which is determined by Bayes' theorem:

$$p(\theta|\mathbf{X},\alpha) = \frac{p(\theta,\mathbf{X},\alpha)}{p(\mathbf{X},\alpha)} = \frac{p(\mathbf{X}|\theta,\alpha)p(\theta,\alpha)}{p(\mathbf{X}|\alpha)p(\alpha)} = \frac{p(\mathbf{X}|\theta,\alpha)p(\theta|\alpha)}{p(\mathbf{X}|\alpha)} \tag{2.19}$$

where $p(\mathbf{X}|\alpha) = \int p(\mathbf{X}|\theta)p(\theta|\alpha)d\theta$.

Then we can predict the distribution of a new data point $\tilde{x}$, which is also known as the posterior predictive distribution, by:

$$p(\tilde{x}|\mathbf{X},\alpha) = \int p(\tilde{x}|\theta)p(\theta|\mathbf{X},\alpha)d\theta \tag{2.20}$$

2.4.3 Evidence Framework

In most cases, the hyperparameter α in Equation 2.19 is unknown. However, they can be set accoding to evidence framework [40]. Given the noise precision β, the evidence framework works as follows. First, the parameters are estimated by:

$$p(\mathbf{X}|\alpha,\beta) = \int p(\mathbf{X}|\theta,\beta)p(\theta|\alpha)d\theta \tag{2.21}$$

Then the hyperparameters can be updated by:

$$\beta_t = \frac{N-\gamma}{(\mathbf{Y}-\mathbf{X}\theta)^\top(\mathbf{Y}-\mathbf{X}\theta)} \tag{2.22}$$

and

$$\alpha_t = \frac{p(\mathbf{X}|\alpha_{t-1},\beta_{t-1})}{2\theta^\top\theta + \text{Trace}(\sum)} \tag{2.23}$$

where N is the number of inputs, $\sum$ is the covariance of the input data, and γ is the number of the 'well-determined' coefficients, given by

$$\gamma = p(\mathbf{X}|\alpha_{t-1},\beta_{t-1}) - \alpha_{t-1}\text{Trace}(\sum) \tag{2.24}$$

The derivation of Equation 2.22 to 2.24 is available in [40].

2.5 Approximate Inference Methods

In case of neural networks, Bayesian inference can be used to calculate the posterior distribution $P(\theta|\mathcal{D})$ of the network parameters given the training data. This distributions can then answer predictive quires about unseen data by taking expections. Each possible configuration of the parameters, weighted according to the posterior distribution, makes a prediction about the unknown label given the test data. Thus taking an expectation under the posterior distribution on parameters is equivalent to using an ensemble of an uncountably infinite number of neural networks [8]. However, this is not tractable in practice because of the large size of neural networks.

Several approaches have been proposed to find approximation to the posterior distribution on the parameters, e.g., the Laplace approximation [40], Hamiltonian Monte Carlo [48] and variational inference [25, 23]. Among these methods, the Laplace approximation is widely used because of its easy implementation. The Laplace approximation [40] is based on a second-order Taylor approximation of the log posterior around the MAP estimate, which results in a Gaussian approximation to the posterior.

2.5.1 Taylor Series

In mathematics a function can be represented by a Taylor series, which is an infinite sum of terms that are calculated from the values of the function's derivatives at a single point [58]. The Taylor series of function $f(x)$ that is infinitely differentiable at point a is the series:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + ... \tag{2.25}$$

Then the function can be approximated by using a finite number of terms of Taylor series [1]. In case of a second-order Taylor approximation, which is also know as a quadratic approximation, we can get:

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 \tag{2.26}$$

2.5.2 MAP

In Bayesian statistics, a maximum a posterior probability (MAP) estimate is an estimate of an unknown quantity based on observed data [47]. Assume that $\mathbf{X}$ is the set of observed data points, $x_1, ..., x_n$, and θ is the parameter of data distribution. According to Bayes' Theorem, the posterior distribution can be calculated by

$$p(\theta|\mathbf{X}) = \frac{p(\mathbf{X}|\theta)p(\theta)}{p(\mathbf{X})} \tag{2.27}$$

The method of maximum a posterior estimation then estimates θ [4]:

$$\hat{\theta}_{MAP}(x) = \arg\max_{\theta} f(\theta|x) = \arg\max_{\theta} \frac{p(\mathbf{X}|\theta)p(\theta)}{p(\mathbf{X})} \tag{2.28}$$

Since the marginal likelihood $p(\mathbf{X})$ is always positive and does not depend on θ, it plays no role in the optimization. Therefore, we can rewrite Equation 2.27 as:

$$\hat{\theta}_{MAP}(x) = \arg\max_{\theta} p(\mathbf{X}|\theta)p(\theta) \tag{2.29}$$

2.5.3 Laplace approximation

The Laplace approximation is a method of approximating Bayesian parameter estimation [40]. It will find a Gaussian approximation to the conditional distribution of a set of continuous variables.

Assume that an unnormalized probability density $P^*(x)$ which achieves its maximum at a point x_0. Its normalizing constant is:

$$Z_P \equiv \int P^*(x)dx \tag{2.30}$$

If we Taylor-expand the logarithm of $P^*(x)$ around its maximum point x_0 where $\frac{\partial}{\partial x}\ln P^*(x)|_{x=x_0} = 0$ and approximate $P^*(x)$ using 2nd derivative, we can get:

$$\ln P^*(x) \simeq \ln P^*(x_0) - \frac{c_1}{2}(x-x_0) - \frac{c_2}{2}(x-x_0)^2 \tag{2.31}$$

where

$$c_1 = -\frac{\partial}{\partial x}\ln P^*(x)|_{x=x_0} \tag{2.32}$$

and

$$c_2 = -\frac{\partial^2}{\partial x^2}\ln P^*(x)|_{x=x_0} \tag{2.33}$$

Since x_0 is a local maximum and $\frac{\partial}{\partial x}P^*(x)|_{x=x_0} = 0$, we can rewrite Equation 2.31 as:

$$\ln P^*(x) \simeq \ln P^*(x_0) - \frac{c}{2}(x-x_0)^2 \tag{2.34}$$

where

$$c = -\frac{\partial^2}{\partial x^2}\ln P^*(x)|_{x=x_0} \tag{2.35}$$

We then approximate $P^*(x)$ by an unnormalized Gaussian:

$$Q^*(x) \equiv P^*(x_0)\exp\{-\frac{c}{2}(x-x_0)^2\} \tag{2.36}$$

and approximate the normalizing constant Z_P by the normalizing constant of this Gaussian,

$$Z_Q = P^*(x)\sqrt{\frac{2\pi}{c}} \tag{2.37}$$

Equation 2.30 to 2.37 can be generalized to approximate Z_P for a density $P^*(\mathbf{x})$ over a K-dimensional space $\mathbf{x}$. Assume $\mathbf{A}$ is the Hessian matrix, the matrix of the second derivatives, of $-\ln P^*(\mathbf{x})$ at the maximum $\mathbf{x}_0$. And we can get:

$$A_{ij} = -\frac{\partial^2}{\partial x_i \partial x_j}\ln P^*(\mathbf{x})|_{\mathbf{x}=\mathbf{x}_0} \tag{2.38}$$

Then Equation 2.31 can be generalized to:

$$\ln P^*(\mathbf{x}) \simeq \ln P^*(\mathbf{x}_0) - \frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^\top \mathbf{A}(\mathbf{x}-\mathbf{x}_0) \tag{2.39}$$

The normalizing constant can be approximated by:

$$\begin{aligned} Z_p \simeq Z_Q &= P^*(\mathbf{x})\frac{1}{\sqrt{\det\frac{1}{2\pi}\mathbf{A}}} \\ &= P^*(\mathbf{x})\sqrt{\frac{(2\pi)^K}{\det\mathbf{A}}} \end{aligned} \tag{2.40}$$

Predictions can be made using the approximation Q^*

$$Q^*(\mathbf{x}) \equiv P^*(\mathbf{x_0})\exp\{\frac{1}{2}(\mathbf{x}-\mathbf{x}_0)^\top \mathbf{A}(\mathbf{x}-\mathbf{x}_0\} \tag{2.41}$$

Chapter 3

BayesNAS for RNN

BayesNAS [64] is a promising method to significantly reduce search time. It employs the classic Bayesian learning approach using hierarchical automatic relevance determination (HARD) priors. It can find candidated architectures by training the over-parameterized network for only one epoch. However, this work is only able to design convolutional cells for image classification. Since our goal is to extend BayesNAS to the design of recurrent architectures, this section first describes how the search space for RNN is designed. Then the details of the search strategy of BayesNAS are explained.

3.1 Search Space

The search space of neural architecture search defines what kind of neural architectures could be learned based on a NAS approach. Specifically for recurrent neural network, we search for a basic recurrent cell. The inputs of the learned cell are the input of the current time step and output of the previous time step and the hidden state of the current time step is generated based on the inputs. Therefore the learned cell could be recursively connected to form a recurrent neural network.

3.1.1 Design Search Space as a DAG

[66] iterates over all of the possible graphs and search for a cell based on the reward signal. We can consider this as searching for sub-graphs in a super-graph. Therefore, the search space of NAS could be viewed as a directed acyclic graph (DAG) ([50]; [38]). The candidate architecture can be generated by sampling a sub-graph of the DAG. The edges in the DAG represent the operations and the nodes represent the local feature map.

We design our search space as followings. A recurrent cell is viewed as a directed acyclic graph, which consists of an ordered sequence of N nodes. Each node x_i is associated with a feature map in the recurrent network and each edge $e_{i,j}$ represents a kind of operations $o_{i,j} \in O$ which transforms the feature map x_i. To control the information flow in the graph, operations of all edges are associated with scaling scalars $w^o_{i,j}$. The input nodes are defined as the input of the current time step and the hidden state of the previous time step. For the output of the recurrent cell, all the intermediate nodes are averaged. The cell uses the average as the output. Since all the nodes are ordered, each

intermediate node is connected with all of its predecessors. The intermediate node is computed by

$$x_j = \sum_{i<j} \sum_{o \in O} w^o_{i,j} o_{i,j}(W^o_{i,j} x_i) \tag{3.1}$$

where $\mathbf{W} = \{W^o_{i,j}\}$ are network parameters and $\mathbf{w} = \{w^o_{i,j}\}$ are architecture parameters. Figure 3.1 illustrates an example of how to design recurrent cell. The example recurrent cell has $N = 4$ nodes and two alternative operations. The two input nodes x_t and h_t represents the input of the current time step and the hidden state of the previous time step respectively. The first node, node 0, is obtained by linearly transforming x_t and h_t, summing the results then passing through a tanh activation function, which means that $x_0 = tanh(x_t \cdot W_x + h_t \cdot W_h)$. Note that how to compute x_0 is fixed. Then for each of the intermediate nodes, node 1 to 3, the search strategy decides the index of its predecessor to connect with and the kind of operation to apply. For example, Figure 3.2 shows a learned cell with $x_1 = relu(x_0 \cdot W^{relu}_{0,1})$, $x_2 = tanh(x_0 \cdot W^{tanh}_{0,2})$, and $x_3 = relu(x_1 \cdot W^{relu}_{1,3})$.

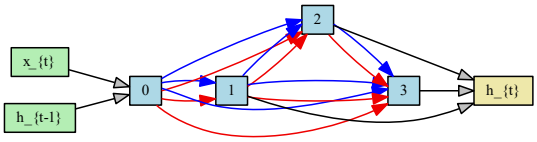


Figure 3.1: *The DAG defines the search space of NAS. The very first two nodes x_t and h_t are the input of the current time step and the output of the previous time step. The red and blue arrows represent candidate operations.*

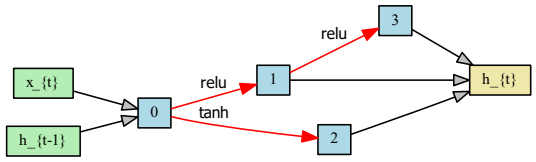


Figure 3.2: *An example of the learned recurrent cell.*

3.1.2 Recurrent Highway Network (RHN)

In a standard recurrent layer, assume x_t is the inputs of the current time step and h_{t-1} is the hidden state of the previous time step. Then the current hidden state can be computed by:

$$h_t = f_H(W_{IH} x_t + W_{HH} h_{t-1} + b_H) \tag{3.2}$$

Therefore, there is only one nonlinear transition function f_H from one time step to the next in a standard recurrent layer. However, in our search space as discussed in

Section 3.1.1, each recurrent cell has more complex nonlinear transition functions. In each time step, the depth of the step-to-step recurrent state transition is increased, which is also known as the recurrence depth [65].

Compared to stacking recurrent layers, increasing the recurrence depth can add significantly higher modeling power to an RNN as shown in Figure 3.3. Assume that there are T time steps. Stacking d recurrent layers allows a maximum path length of $d+T-1$ between hidden states, while a recurrence depth of d enables a maximum path length of $d \times T$. While this allows greater power and efficiency using larger depths, it also explains why such architectures are much more difficult to train compared to stacked RNNs. Deep networks suffer from what are commonly referred to as the vanishing and exploding gradient problems [26], because the magnitude of the gradients may shrink or explode exponentially during backpropagation.

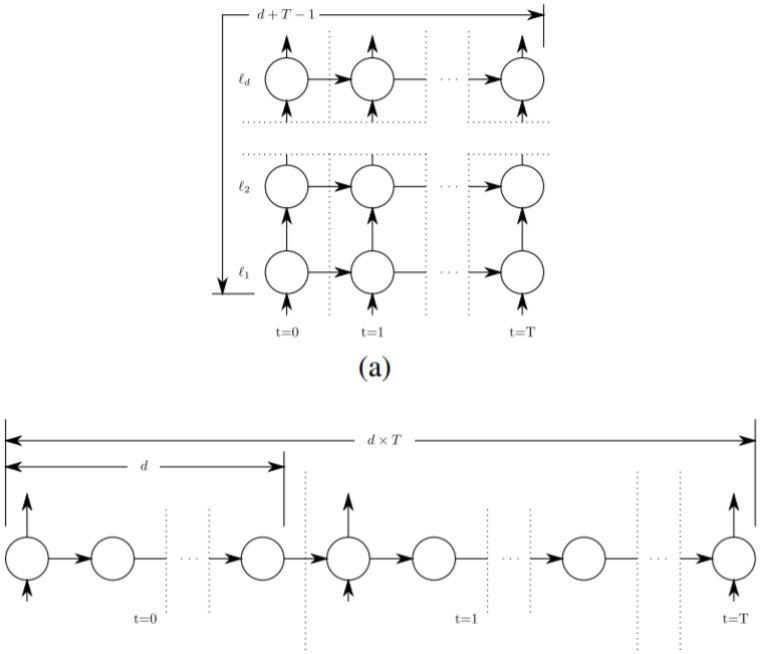


Figure 3.3: *Comparison of (a) stacked RNN with depth d and (b) Deep Transition RNN of recurrence depth d, both operating on a sequence of T time steps. The longest path between hidden states T time steps is $d \times T$ for Deep Transition RNNs.*

In order to solve the gradient issues and augment the simple transformations between nodes, our setting is similar to the settings of ENAS ([50]) and DARTS ([38]). Each operation in the DAG is enhanced with a highway bypass ([65]) as shown in Figure 3.4. Highway connections enable easy training of very deep feedforward networks through the use of adaptive computation. Assume the output of node i is s_i. Let $h = \sigma(W_h s_{i-1})$, $c = sigmoid(W_c s_{i-1})$ be outputs of nonlinear transforms with associated weight matrices and σ is the activation function of the corresponding operation. Therefore, the output of node i could be computed by:

$$s_i = c \otimes h + (1-c) \otimes s_{i-1} \tag{3.3}$$

For instance, instead of having $x_2 = tanh(x_0 \cdot W_{0,2}^{tanh})$ as shown in the example from Section 3.1.1, we have

$$x_2 = c \otimes h + (1-c) \otimes x_0 \tag{3.4}$$

where $c = \text{sigmoid}(x \cdot W^{(c)})_{0,2})$ and $h = \tanh(x \cdot W^{(h)})_{0,2})$

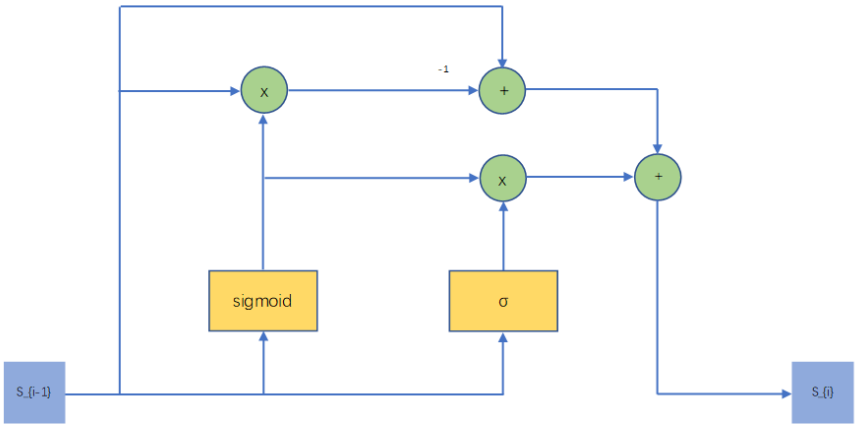


Figure 3.4: *An illustration of enhancement of operation. s_{i-1} and s_i are two nodes in the DAG, which represents feature maps. σ is the activation function. Instead of applying operation σ directly, each operation in the DAG is enhanced with a highway bypass.*

3.2 Dependency

In this section, how to model the dependency between a node and its predecessors and successors is stated. As shown in Figure 2.14, both the blue and red edges from node 2 to 3 and from node 2 to 4 should be removed as a consequence. Therefore, the following proposition is presented:

Proposition 1 There is no information flow from node j to node k under operation o' if and only if all the operations of all the predecessors of node j are zeros or the corresponding scaling scalar $w_{j,k}^{o'}$ in Equation 3.1 is zero.

One possible expression to encode this proposition is $w_{j,k}^{o'} \sum_{o \in O} \sum_{i<j} w_{j,k}^{o}$. In order to implement this expression, for each w_{ij}^{o}, we introduce a variable, switch s_{ij}^{o} which has two states {ON, OFF}. Assume that $w_{jk}^{o'}$ is redundant if $s_{jk}^{o'}$ is OFF or all s_{ij}^{o} are OFF, $\forall i < j, o \in O$. One possible solution is

$$\bigcup_{i<j} \bigcup_{o \in O} \{s_{ij}^{o} \cap s_{jk}^{o'}\} \tag{3.5}$$

To encode set union and intersection, assume that s is a continous variable. If $s = \infty$, the switch is ON and $s = 0$ for OFF. Then we can use addition and multiplication to

represent set union and intersection respectively. For example, if $s1$ is ON and $s2$ is OFF, then $s1 \cup s2$ is ON and $s1 \cap s2$ is OFF, which can be represented by $s1 + s2 = \infty$ and $s1 \times s2 = 0$ respectively.

An approach to encode this logic is to assign a probability distribution, for example Gaussian distribution, over $w_{ij}^{o'}$

$$p(w_{jk}^{o'}) = \mathcal{N}(w_{jk}^{o'}|0, s_{jk}^{o'}), \sum_{i<j} p(w_{ij}^{o}) = \sum_{i<j} \mathcal{N}(w_{ij}^{o}|0, s_{ij}^{o}) \tag{3.6}$$

Since $w_{ij}^{o}, \forall i, j, o$ are independent with each other, the distribution over $w_{jk}^{o'} \sum_{i<j} w_{ij}^{o}$ is

$$\begin{aligned} p(w_{jk}^{o'} \sum_{i<j} w_{ij}^{o}) &= \mathcal{N}(w_{jk}^{o'}|0, s_{jk}^{o'}) \sum_{i<j} \mathcal{N}(w_{ij}^{o}|0, s_{ij}^{o}) \\ &= \mathcal{N}(w_{jk}^{o'}|0, s_{jk}^{o'}) \mathcal{N}(\sum_{i<j} \frac{s_{ij}^{o} w_{ij}^{o}}{\sum_{i<j} s_{ij}^{o}}|0, s_{ij}^{o}) \\ &= \mathcal{N}(w_{jk}^{o'} \sum_{i<j} \frac{s_{ij}^{o} w_{ij}^{o}}{\sum_{i<j} s_{ij}^{o}}|0, \gamma_{jk}^{o'}) \end{aligned} \tag{3.7}$$

where

$$\gamma_{jk}^{o'} = (\frac{1}{\sum_{i<j} \sum_{o \in O} s_{ij}^{o}} + \frac{1}{s_{jk}^{o'}})^{-1} \tag{3.8}$$

Since Equation 3.5 and Equation 3.7 are equivalent, we may find the redundant operations in a probabilistic manner by applying Bayesian methods.

In our search space, we exclude zero operations as candidate operations. This is because zero operations should have higher priority than other possible operations. Once zero operations are selected, all the non-zero operations should not be selected after all. Instead a dummy node i' is added between node i and node j. Only a single identity operation is allowed. The associated weight $w_{ii'}$ is trainable and initialized to 1 as well as its switch $s_{ii'}$. If s'_{ii} if OFF, all the operations from i' to j will be disabled. Therefore Equation 3.8 can be rewritten as:

$$\gamma_{jk}^{o'} = (\frac{1}{\sum_{i<j} \sum_{o \in O} (s_{ii'} + s_{i'j}^{o})^{-1}} + \frac{1}{s_{jk}^{o'}})^{-1} \tag{3.9}$$

Equation 3.9 encode the logic of Equation 3.5. If $s_{ii'}$ is OFF, which means that $s_{ii'} = 0$, γ_{jk}^{o} will always be a small value close to zero. Therefore, we have great confidence that the corresponding weight w_{jk} is zero.

3.3 Search Strategy

The search strategy details how to explore the search space. Since our search strategy is one-shot based, the objective becomes removing redunt edges. This section explains how BayesNAS employ classic Bayesian learning to determine the uncertainty of parameter distribution.

3.3.1 Bayesian Neural Network

We can easily apply Bayesian learning to the NAS problem ([64]). The likelihood for the network weights and the noise precision σ^{-2} is

$$p(\mathbf{Y}|\mathbf{W},\mathbf{w},\mathbf{X},\sigma^2) = \prod_{n=1}^{N} \mathcal{N}(y_n|\mathbf{Net}(x_n;\mathbf{W},\mathbf{w});\sigma^2) \tag{3.10}$$

We have assigned a Gaussian prior distribution. In particular,

$$p(\mathbf{W}|\lambda) = \prod_{i<j}\prod_{o\in O} \mathcal{N}(W_{ij}^{o}|0,\lambda^{-1}) \tag{3.11}$$

$$p(\mathbf{w}|\mathbf{s}) = \prod_{j<k}\prod_{o\in O}\prod_{o'\in O} \mathcal{N}(w_{jk}^{o'}\sum_{i<j} w_{ij}^{o}|0,\gamma_{jk}^{o'}) \tag{3.12}$$

where $\gamma_{jk}^{o'}$ is the variance of $w_{jk}^{o'}$ as defined in Equation 3.9 and σ^{-2}, λ and $\mathbf{s}$ are hyperparameters. Hierarchical priors are employed on these variables using Gamma priors on the inverse variances following Mackay's evidence framework ([40]). Following [7], the hyper-prior for these hyperparameters can be chosen to be a gamma distribution, i.e., $p(\lambda) = \mathrm{Gam}(\lambda|a^{\lambda},b^{\lambda})$, $p(\sigma^{-2}) = \mathrm{Gam}(\sigma^{-2}|a^{\sigma^{-2}},b^{\sigma^{-2}})$ and $p(s_{ij}^{o}) = \mathrm{Gam}(s_{ij}^{o}|a^{s_{ij}^{o}},b^{s_{ij}^{o}})$. To make these priors flat, we fix a and b to zero, which means that the priors are uniform for analysis and implementation. This formulation of prior distribution is a type of hierarchically constructed automatic relevance determination (HARD).

Then we can get the posterior distribution based on Bayes' rule:

$$p(\mathbf{W},\mathbf{w},\lambda,\mathbf{s},\sigma^2|\mathcal{D}) = \frac{p(\mathbf{Y}|\mathbf{X},\mathbf{W},\mathbf{w},\lambda,\mathbf{s},\sigma^2)p(\mathbf{W}|\lambda)p(\mathbf{w}|\mathbf{s})p(\lambda)p(\gamma)p(\sigma^2)}{p(\mathbf{Y}|\mathbf{X})} \tag{3.13}$$

In a NAS problem, we only focuse on the architecture parameters $\mathbf{w}$, which means that we can fix hyperparameters related to network parameters $\mathbf{W}$. Particularly, λ is set to the weight decay coefficient in SGD and σ^2 is set to the regularization coefficient for network parameters. In case of uniform hyperpriors, we only need to maximize the term $p(\mathbf{Y}|\lambda,\mathbf{s},\sigma^2)$ ([40], [7]):

$$\int\int p(\mathbf{Y}|\mathbf{W},\mathbf{w},\mathbf{X},\lambda,\mathbf{s},\sigma^2)p(\mathbf{W}|\lambda)p(\mathbf{w}|\mathbf{s})d\mathbf{W}d\mathbf{w} \tag{3.14}$$

3.3.2 Laplace approximation

Since the quantity in Equation 3.14 is the marginal likelihood [48], we can use the Laplace approximation to approximate the integral. However, we need to compute the inverse Hessian of the log-likelihood in Equation 2.39 if we adopt Laplace approximation, which can cause intensive computation cost. For a k dimensional parameter vector, computing the Hessian needs $O(k^2)$ evaluations of the log likelihood function. Due to the large size of the practical neural networks, k can be very large. It presents a significant computation cost, which makes it infeasible. We propose an efficient method to approximate Hessian in RNN. The detailed approximation method is explained in Chapter 4.

3.3.3 Optimization

The search strategy is summarized in Algorithm 2. The candidate architecture is derived by:

1. Jointly train architecture parameters **w** and network weights **W** by minimizing the loss function. The loss function has three components: the performance prediction loss l_{pred}, the regularization of network weights **W** and the regularization of architecture parameters **w**:

$$L = l_{pred} + \lambda \|\mathbf{W}\|_2^2 + \lambda_w \sum_{i<j} \sum_{o \in O} \left\| \omega_{ij}^o(t) w_{ij}^o \right\|_1 \tag{3.15}$$

   where λ and λ_w are the regularization coefficients of network parameters and architecture parameters respectively.

2. Freeze the architecture parameters **w** and compute corresponding hessian.

3. Update the variables associated with w. Equation 3.16 computes the convariance of the posterior of the energy function over data. Equation 3.18 updates the value of hyperparameter s, which represents the uncertainty of the corresponding architecture parameter. However, the update rules for s does not consider the dependency has been explained in Section 3.2. Therefore, based on the Gaussian prior defined in Equation 3.8 and 3.9, the uncertainty could be computed in Equation 3.19.

4. Prune the architecture. Since $p(w_{jk}^{o'})$ in Equation 3.7 is Gaussian with zero mean $\gamma_{jk}^{o'}$ variance, the maximum information entropy of the distribution is $\frac{1}{2} ln(2\pi e \gamma_{jk}^{o'})$. Therefore if $\gamma_{jk}^{o'} \leq 0.0585$, the related link will be removed.

To make our generated architectures comparable with those in the existing work, for each node, we retain the edge with the highest γ in the pruned graph, which means that we select the operation of its prodecessor with the highest confidence that the operation should not be removed.

Algorithm 2 BayesNAS Algorithm

Initialization:

$\gamma(0), \omega(0), w(0) = 1$, sparsity intensity $\lambda_\omega^o \in \mathbb{R}^+$

Iteration:

for t = 1 to T_{max} **do**

1. Update w and W by minimizing loss function
2. Compute Hessian for **w**
3. Update variables associated with **w**

while $i < j < k, o, o' \in O$ **do**

$$C_{jk}^{o'}(t) = (\frac{1}{\gamma_{jk}^{o'}(t-1)} + H_{jk}^{o'}(t))^{-1} \tag{3.16}$$

$$\omega_{jk}^{o'}(t) = \frac{\sqrt{\gamma_{jk}^{o'}(t-1) - C_{jk}^{o'}(t)}}{\gamma_{jk}^{o'}(t-1)} \tag{3.17}$$

$$s_{jk}^{o'}(t) = \left| \frac{w_{jk}^{o'}(t)}{\omega_{jk}^{o'}(t)} \right| \tag{3.18}$$

$$\gamma_{jk}^{o'}(t) = (\frac{1}{\sum_{i<j} \sum_{o \in O} s_{ij}^{o}(t)} + \frac{1}{s_{jk}^{o'}(t)})^{-1} \tag{3.19}$$

end while

end for

Chapter 4

Efficient Hessian Computation

As shown in Equation 3.16, to make the algorithm work, we need Hessian of architecture parameters. Although Hessian of weight matrix has been widely used in second-order optimization technology, it is still not feasible to calculate explicit Hessian directly because of the great computation cost. Inspired by the Hessian calculation methods for Fully Connected (FC) layers as shown in [9], we propose a recursive and efficient method to compute the Hessian of recurrent layer.

4.1 Hessian computation for fully connected layer

Previous work [9] proposed a recursive method to compute $\mathbf{H}$ in a fully connected layer. The behaviour of fully connected layer can be formulated as:

$$h_j^o = W_{i,j}^o a_i, \quad a_i = \sigma(h_i) \tag{4.1}$$

where h_i is the pre-activation value of node i and a_i is the activation value of node i. σ represents the element-wise activation function. $W_{i,j}^o$ is the weight matrix associated with operation o between node i and node j. Therefore, the hessian H of $W_{i,j}^o$ can be computed by:

$$\mathbf{H}_{i,j}^o = a_i \cdot (a_i)^\top \otimes H_j^o \tag{4.2}$$

where $\otimes$ stands for Kronecker product.

The pre-activation Hessian H_j^o is known and could be used to compute the pre-activation Hessian recursively for the previous layer:

$$H_i = B_i (W_{i,j}^o)^\top H_j^o W_{i,j}^o B_i + D_i \tag{4.3}$$

where

$$B_i = diag(\sigma'(h_i)), \quad D_i = diag(\sigma''(h_i) \circ \frac{\partial L}{\partial a_i}) \tag{4.4}$$

It should be noted that $diag()$ does not represent extracting the diagonal values of input variable. Instead it means that B_i and D_i are diagonal matrices, whose diagonal values are the first and second derivatives of σ respectively. If we replace the original pre-activation Hessian H_j^o and Hessian $\mathbf{H}$ with their diagonal values for recursive computation, the matrix multiplication could be reduced to vector multiplication, which means

that the computation complexity could be significantly reduced. We could rewrite the Hessian computation process as:

$$\mathbf{H}^o_{i,j} = a_i^2 \otimes H^o_j \tag{4.5}$$

$$H_i = B_i^2 \circ (((W^o_{i,j})^\top)^2 H^o_j) + D_i \tag{4.6}$$

where

$$B_i = \sigma'(h_i), \quad D_i = \sigma''(h_i) \circ \frac{\partial L}{\partial a_i} \tag{4.7}$$

4.2 Hessian computation for recurrent layer

A RNN cell could be unrolled over the time sequence. We denote u_t, x_t, and o_t as the input of the current time step, the hidden state of the current time step and the output of the current time step respectively. The mathematical operation in a recurrent layer can be formulated as:

$$x_t = tanh(W_i u_t + W_x x_{t-1}) \tag{4.8}$$

$$o_t = \sigma(W_o x_t) \tag{4.9}$$

where W_i, W_x and W_o represent the weight matrix of the input, hidden state and output. σ is the output activation function. Therefore, the Hessian computation at a certain time step can refer to the approach for fully connected layer which has already been explained.

The left main problem is how to conduct sequential Hessian calculations when considering the influence of time sequence and the backward propagation through time (BPTT) process. BPTT is a generalization of back-propagation for feed-forward networks. The standard BPTT method for learning RNN unfolds the recurrent neural network in time and propagates the gradient backwards through time as shown in Figure 4.1. Assume θ is the set of network parameters and h_t represents the hidden state at time t. The gradient of h_t is

$$\frac{\partial \mathcal{L}}{\partial \theta} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial \theta} \tag{4.10}$$

where the expansion of loss function gradients at time t is

$$\frac{\partial \mathcal{L}_t}{\partial \theta} = \sum_{k=1}^{t} (\frac{\partial \mathcal{L}_t}{\partial h_t} \frac{\partial h_t}{\partial h_k} \frac{\partial h_k}{\partial \theta}) \tag{4.11}$$

$\frac{\partial h_k}{\partial \theta}$ describes how the parameters affect the loss at time step k. Also, in order to transport the error from time step t to time step k, we have

$$\frac{\partial h_t}{\partial h_k} = \prod_{i=k+1}^{t} \frac{\partial h_i}{\partial h_{i-1}} \tag{4.12}$$

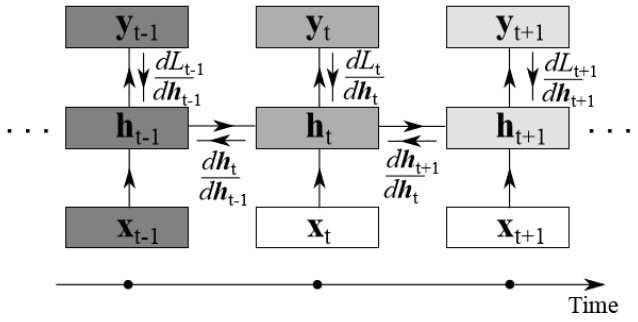


Figure 4.1: *An abstract illustration of BPTT process of RNN*

How we compute Hessian is similar to the computation of gradient. Assume that the input sequence length of a RNN cell is T and τ is the backward propagation time horizon. Thus, the Hessian for W_o can be calculated as:

$$\mathbf{H}_o = \frac{1}{T}\sum_{t=1}^{T}\mathbf{H}_o^t \tag{4.13}$$

where Hessian $\mathbf{H}_o^t = (x_t)^2 \otimes H_o^t$ and $\mathbf{H}_o^t$ is the pre-activation Hessian of W_o at time step t.

Next we compute the pre-activation Hessian for W_x. Then the Hessian $\mathbf{H}_x$ is computed as:

$$\mathbf{H}_x = \sum_{t=1}^{T}\sum_{k=1}^{t}\mathbf{H}_x^{t,k} \tag{4.14}$$

$$\mathbf{H}_x^{t,k} = (x_{k-1})^2 \otimes H_x^{t,k} \tag{4.15}$$

where $\mathbf{H}_x^{t,k}$ and $H_x^{t,k}$ represent the Hessian and the pre-activation Hessian transported from time step t to time step k respectively. Therefore, we can compute $H_x^{t,bptt}$ as:

$$H_x^{t,k} = \prod_{bptt=k+1}^{t} B^2 \circ (((W_x)^\top)^2 H_x^{bptt,bptt-1}) \tag{4.16}$$

where

$$B = \sigma'(h_{bptt}), \quad D = \sigma''(h_{bptt}) \circ \frac{\partial L}{\partial x_{bptt}} \tag{4.17}$$

To avoid the gradient issues in recurrent neural network, at each time step t, we compute the pre-activation Hessian H_x and Hessian $\mathbf{H}_x$ over a proper backward time horizon τ. Therefore, we can rewrite Equation 4.14 as:

$$\mathbf{H}_x = \frac{1}{T \times \tau}\sum_{t=1}^{T}\sum_{k=t-\tau+1}^{t}\mathbf{H}_x^{t,k} \tag{4.18}$$

Similarly the Hessian of the input could be computed as:

$$\mathbf{H}_i = \frac{1}{T \times \tau}\sum_{t=1}^{T}\sum_{k=t-\tau+1}^{t}\mathbf{H}_i^{t,k} \tag{4.19}$$

where $\mathbf{H}_i^{t,k} = (u_{k-1})^2 \otimes H_i^{t,k}$. $H_i^{t,k}$ can be computed as:

$$H_i^{t,k} = \prod_{bptt=k+1}^{t} B^2 \circ (((W_i)^\top)^2 H_i^{bptt,bptt-1}) \tag{4.20}$$

$$B = \sigma'(h_{bptt}), \quad D = \sigma''(h_{bptt}) \circ \frac{\partial L}{\partial h_{bptt}} \tag{4.21}$$

4.3 Hessian computation for architecture parameters

Now we need to consider the computation of Hessian of architecture parameters. The output from node i to node j under operation o can be computed as:

$$\mathcal{B}_j^o = w_{ij}^o \mathcal{B}_i \tag{4.22}$$

where w_{ij}^o is the architecture parameter and $B_i = \sum_{o \in O} B_i^o$ is the output from node i. Since w_{ij}^o is a scalar, inspired by [9], the Hessian of w_{ij}^o can be computed as:

$$\mathbf{H}_{ij}^o = \mathbb{E}(B_i^2 H_j) \tag{4.23}$$

where $\mathbb{E}$ return the mean. The pre-activation Hessian of w_{ij}^o can be computed as:

$$H_i = \sum_{o \in O} (w_{ij}^o)^2 H_j \tag{4.24}$$

Chapter 5

Experiments

In this section, the designed experiments which search for competitive novel RNN architectures on datasets of language modelling are reported. Our settings are similar to settings of [38]; [50]; [39].

5.1 Datasets

In the following, the datasets which are relevant for this work will be discussed. These datasets were pre-splitted when we downloaded them.

5.1.1 Penn Treebank

The Penn Treebank ([41]) has long been a central data set for experimenting with language modeling. It is a corpus consisting of over 4.5 million words of American English, which consists of 929k training words, 73k validation words, and 82k test words. The data set was heavily preprocessed by [45]. Words were lower-cased, numbers were replaced with N, newlines were replaced with $< eos >$, and all other punctuation was removed. The vocabulary has 10k words. The rest of the tokens was replaced by an $< unk >$ token.

5.1.2 WikiText-2

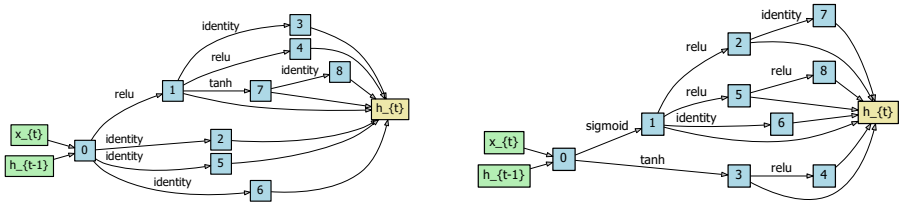
WikiText-2 ([43]) is another benchmark dataset for language modelling. The PTB was heavily preprocessed which means that it has many limitations and is unrealistic for real language use. To overcome these limitations, WikiText-2 was constructed by using articles extracted from Wikipedia which have been reviewed by humans. It is broad in coverage and stable. The size of WikiText-2 is two times the size of the Penn Treebank dataset. It has 2088k training words, 217k validation words, and 245k test words. The vocabulary has more than 33k words.

5.2 Architecture Search

The set of operations includes four activation functions, tanh, relu, sigmoid and identity mapping, which follows [50], [66], and [38].

The cell has 12 intermediate nodes. The first node, node 0, is obtained by adding the results of linearly transforming the two input nodes and passing through a tanh activation function. The rest of the cell is learned based on the search strategy. Each operation is enhanced with highway connections([65]). The output of each node is the sum of results of all operations of all its predecessors. And the final output of the cell is the average of all intermediate nodes. Considering the gradient explosion problem, batch normalization layer is always added after each node during architecture search stage. Also in order to avoid scaling the output of each operation, the affine parameters of batch normaliztion is not learnable, which means that $\gamma = 1$, $\beta = 0$. The final recurrent neural network is a single cell without stacking.

Both the embedding and the hidden sizes are set to 300. Dropout is applied to several layers([19]), 0.2 to word embeddings, 0.75 to the cell input, and 0.25 to all the hidden nodes. Other training settings are identical to those in [44].



(a) $\lambda = 0.15$ **(b)** $\lambda = 0.17$

Figure 5.1: *The best 2 cells after training for 800 epochs*

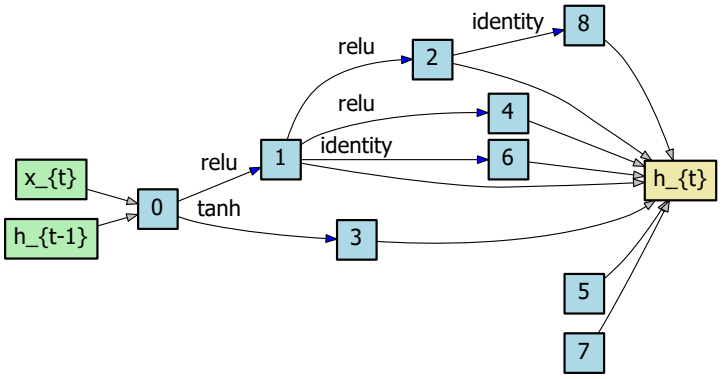


Figure 5.2: *The learned cell with fewer parameters ($\lambda = 0.20$)*

5.3 Architecture Evaluation

To determine the architecture for final evolutation, the set of regularization coefficients of architecture parameters λ_w (Equation 3.15) is set to $\{0.1, 0.13, 0.15, 0.17, 0.2\}$. For

each λ_w value, we run the experiments four times with different random seeds because the architecture learned can be initialization-sensitive (Figure 5.3). Therefore, there are 20 candidated architectures. For evaluation, we first train all candidates five times for 800 epochs. The best 2 cells are picked based on their average validation performance obtained. The best 2 cells after training for 800 epochs are shown in Figure 5.1. To evaluate the selected architectures, we train them for 2500 more epochs and report their performance on the test dataset.

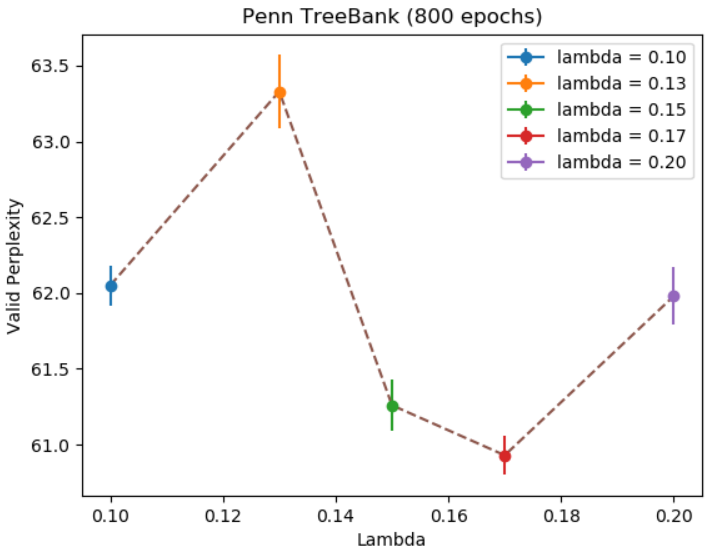


Figure 5.3: *Search progress for recurrent cell on Penn TreeBank. For each λ, we run the experiments four times with different initial seeds and report the mean and variance of the performance.*

Experimental details A single layer recurrent neural network is trained 3300 epochs in total with batch size 64 using averaged SGD([51])(ASGD), with learning rate $\eta_w =$ 10 and weight decay 8×10^{-7}. To speedup, we start with SGD and trigger ASGD using the same protocol as in [44]. To ensure the model size is comparable with other baselines, the sizes of both the embedding and the hidden layer are set to 850. We apply variational dropout ([19]) of 0.1 to word embeddings, 0.75 to the cell input, 0.25 to the hidden nodes, and 0.75 to the output layer. Other training settings are identical to those in [44]. For fair comparison, we do not finetune our model at the end of the training, nor do we use any additional enhancements.

As we treat the architecture search problem as model compression and remove the edge of operation o' between node i and node j if $\gamma_{ij}^{o'} \leq 0.0585$, our search strategy is able to learn a light cell with acceptable performance decrease but with fewer parameters as shown in Figure 5.2. Two intermediate nodes are not connected with any of their predecessors. Therefore, they are removed from the graph. One thing to be noted is that grid search is not used for architecture search due to the limited time. Therefore, the learned cells may not be optimal. It is likely for our search strategy to learn cells with better performance or fewer parameters.

5.4 Normalization

While recurrent architectures are simple and powerful models, it is actually hard to train them properly. There are two widely known issues with properly training recurrent architectures like LSTM and GRU, the vanishing and the exploding gradient problems [49]. These gradient problems can be alleviated by normalization. In the previous work [66, 50, 38, 39], batch normalization is enabled in each node to prevent gradient explosion. However, from the point of view of language modeling, batch normalization is not applicable to recurrent architectures directly. On one hand, batch normalization estimates the variance and mean of the inputs of each batch during training process and uses these estimation values when evaluating the model on validation dataset during evaluation. Therefore, if the distribution of training data and validation data is different, batch normalization can result in poor performance on unseen data. On the other hand, since there are recurrent connections in the architecture, actually we reuse the same batch normalization layer in all time steps, which means that we use the same estimated variance and mean at every time step. However, they are more likely to be different.

In order to get how different normalization methods influence the performance of the learned cell, two normalization methods, batch normalization and layer normalization, are selected. For each method, we can set parameters of the normalization layer to learnable or non-learnable. If the parameters are not learnable, γ and β are fixed to 1 and 0 respectively. Otherwise they are learned by optimization algorithms. Therefore, there are four kinds of normaliztion layer to test. Note that for each normalization method, the same set of initialization seeds is used during the architecture search stage. The normalization layer is removed during performance estimation stage.

Experimential details A single layer recurrent neural network is trained 50 epochs in total with batch size 64 using SGD, with learning rate $\eta_w = 10$ and weight decay 8×10^{-7}. The sizes of both the embedding and the hidden layer are set to 850. We apply variational dropout ([19]) of 0.1 to word embeddings, 0.75 to the cell input, 0.25 to the hidden nodes, and 0.75 to the output layer. Other training settings are identical to those in [44]. The initialization seed during the search stage is reused here to initialize network weights.

5.5 Transferability

In order to investigate the transferability of the cells learned by the search strategy on Penn TreeBank, we train the recurrent cell on another dataset, Wiki-Text2 ([43]), which has a larger data size compared with Penn TreeBank.

Experimential details The experiment hyperparameters are almost the same as in Penn TreeBank. A single layer recurrent neural network is trained 3300 epochs in total with batch size 64 using averaged SGD([51])(ASGD), with learning rate $\eta_w = 10$ and weight decay 5×10^{-7}. To speedup, we start with SGD and trigger ASGD using the same protocol as in [44]. To ensure the model size is comparable with other baselines, the sizes of both the embedding and the hidden layer are set to 700. we

apply variational dropout ([19]) of 0.1 to word embeddings, 0.75 to the cell input, 0.25 to the hidden nodes, and 0.75 to the output layer. Other training settings are identical to those in [44]. For fair comparison, we do not finetune our model at the end of the training, nor do we use any additional enhancements.

5.6 Results Analysis

Table 5.1 presents the performance of learned cells generated by different normalization methods. There is not much difference between the performance of batch normalization and layer normalization. The normalization layer with non-learnable parameters outperforms that with learnable parameters regardless of how the inputs are normalized. This can be explained by the fact that the outputs of operations are rescaled if the parameters are set to learnable.

Table 5.1: *Performance of learned cells generated by different normalization methods (lower perplexity(PPL) is better)*

Architecture	Valid PPL	Parameters(M)
Batch Normalization (non-learnable)	80.4 ± 0.5	23
Batch Normalization (learnable)	82.5 ± 0.4	23
Layer Normalization (non-learnable)	80.7 ± 0.3	23
Layer Normalization (learnable)	82.3 ± 0.4	23

Table 5.2 presents the results of our learned cell and state-of-the-art. The best model learned by our search strategy can achieve a validation perplexity around 59.62 and test perplexity around 57.20, which is very comparable with state-of-the-art. Our automatically learned cell is superior to most of the architectures designed manually except LSTM + SE [61].

Table 5.2: *Comparison with state-of-the-art language models on PTB (lower perplexity(PPL) is better)*

Architecture	Valid PPL	Test PPL	Parameters(M)	Search Cost(GPU days)
Variational RHN [65]	67.9	65.4	23	-
LSTM [44]	60.7	58.8	24	-
LSTM + skip connections [42]	60.9	58.3	24	-
LSTM + 15 softmax experts [61]	58.1	56.0	22	-
NAS [66]	-	64.0	25	10^4
ENAS(v2) [50]	-	55.8	24	0.5
DARTS(1st) [38]	60.2	57.6	23	0.5
DARTS(2nd) [38]	58.1	55.7	23	1
BayesNAS (lambda = 0.17)	59.62 ± 0.53	57.20 ± 0.52	23	0.3
BayesNAS (lambda = 0.20)	60.56 ± 0.38	58.34 ± 0.33	20	0.3

Results in Table 5.3 show the competitive results of the learned cell on WT2 among automatically architectures, although the transferability between Penn TreeBank and WikiText-2 seems a bit weak. The performance of our learned cell is worse than most manually designed architectures on WT2, which is not the case on PTB. This may be because the learned cell is searched on a small dataset (PTB) while evaluate on a large

dataset (WT2). If we search directly on WT2, we could get a better performance and circumvent the issue of transferability.

Table 5.3: *Comparison with state-of-the-art language models on WT2 (lower perplexity(PPL) is better)*

Architecture	Valid PPL	Test PPL	Parameters(M)	Search Cost(GPU days)
LSTM [44]	69.1	66.0	33	-
LSTM + skip connections [42]	69.1	65.9	24	-
LSTM + 15 softmax experts [61]	66.0	63.3	33	-
ENAS [50] (searched on PTB)	72.4	70.4	33	0.5
DARTS [38] (searched on PTB)	71.2	69.6	33	1
BayesNAS (lambda = 0.17)	72.5 ± 0.3	70.1 ± 0.2	33	0.3
BayesNAS (lambda = 0.20)	74.0 ± 0.1	71.4 ± 0.0	31	0.3

Currently most neural architecture search apporaches search on relatively small datasets, such as CIFAR and PTB, then transfer the learned architectures to large-scale datasets, like ImageNet and WT2, although the transferable ability may be a bit weak and the overall results on large-sclae datasets are less strong. The obstacle of directly searching on large-scale datasets is the huge search cost. To the best of our known, we can achieve the lowest search cost compared with other neural architecture search approaches (20 runs in 0.3 GPU days). Note that the code is not heavily optimized and it is likey to achieve a lower search cost than reported. Our search strategy is very efficient which makes searching directly on large-scale datasets possible. We will explore this opportunity in our future work.

Chapter 6

Conclusion

In this work, we propose an efficient gradient-based neural architecture search approach which employs the classic Bayesian learning approach. Our approach reduces the search cost significantly. We get the candidate architecture which could achieve competitive performance compared with state-of-the-art architectures by running only one epoch.

Research Question 1. asks '*How can batch normalization influence the architecture search? Moreover, how can the behaviour of the algorithm be influenced if we use other normalization method, for example, layer normalization which is widely used in RNN?*' - We investigate this in Section 5.4 and find that Batch normalization alleviates the gradient issues and help to stablize the architecture search process. To achieve better performance, the parameters of batch normalization layer should not be learnable to avoid rescaling the outputs of the candidate operations. However, we do not see much difference between the performance of batch normalization and that of layer normalization.

Research Question 2. - '*Can this method be extended to recurrent layers to make classic Bayesian learning approach applicable to design recurrent architectures automatically for language modeling task?*' - To employ Bayesian learning, we propose an efficient method to compute the Hessian of recurrent layer. Then the posterior variance can be updated based on Laplace approximation which requires computation of the inverse Hessian of log likelihood. Our approach is able to find architectures, which can achieve competitive performance, with much lower search cost compared with state-of-the-art.

Research Question 3. - ' *Can we generate novel recurrent architectures that can trade-off between the performance and model size?*' - In our experiments, we find that we can get more sparse architectures if we increase the regularization coefficient of architecture parameters. Finally we discover an architecture with less model parameters and competitive performance.

In conclusion, our algorithm is a promising method for neural architecture search for recurrent architectures. The main contributions are presented below.

6.1 Future Work

There are many directions to improve our NAS approach further. Future research can focus on improving the efficiency or performance of NAS.

Perform Hessian Computation during Backpropagation The hessian computation in our implementation is not efficient. Currently to compute the hessian, all the feature maps are cashed in the memory. However, it is possible to perform hessian computation during backpropagation, which can further reduce the search cost and memory used. If so, it may be possible to search directly on large datasets like WT2. Then the transferability of the learned cell can be improved.

Design novel Search Space Our search space follows Darts and NAO [38, 39], which could be viewed as a directed acyclic graph. Since the search space defines what kind of neural architectures could be learned in principle, a different setup of search space may improve the performance. For example, in our setup, the input of each node is the sum of all outputs from all its predecessors. We can involve more combination methods, like element-wise multiplication. Inspired by the construction of the LSTM cell, we can also introduce memory state as another input to the cell.

Bibliography

[1] Tom. Apostol, editor. *Calculus*, Wiley, 1996. ISBN 0-471-00005-1.

[2] Bowen Baker, Otkrist Gupta, Nikhil Naik, and Ramesh Raskar. Designing neural network architectures using reinforcement learning. *CoRR*, abs/1611.02167, 2016.

[3] Bowen Baker, Otkrist Gupta, Ramesh Raskar, and Nikhil Naik. Practical neural network performance prediction for early stopping. *CoRR*, abs/1705.10823, 2017. URL http://arxiv.org/abs/1705.10823.

[4] Robert Bassett and Julio Deride. Maximum a posteriori estimators as a limit of bayes estimators. *Mathematical Programming*, 11 2016. doi: 10.1007/s10107-018-1241-0.

[5] Gabriel Bender, Pieter-Jan Kindermans, Barret Zoph, Vijay Vasudevan, and Quoc Le. Understanding and simplifying one-shot architecture search. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 550–559, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[6] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, March 2003. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=944919.944966.

[7] James O Berger. Statistical decision theory and bayesian analysis. *Springer Science & Business Media*, 2013.

[8] Charles Blundell, Julien Cornebise, Koray Kavukcuoglu, and Daan Wierstra. Weight uncertainty in neural networks. In *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, ICML'15, pages 1613–1622. JMLR.org, 2015. URL http://dl.acm.org/citation.cfm?id=3045118.3045290.

[9] Aleksandar Botev, Hippolyt Ritter, and David Barber. Practical gauss-newton optimisation for deep learning. In *Proceedings of the 34th International Confer-*

ence on Machine Learning - Volume 70, ICML'17, pages 557–565. JMLR.org, 2017.

[10] Andrew Brock, Theodore Lim, James M. Ritchie, and Nick Weston. SMASH: one-shot model architecture search through hypernetworks. *CoRR*, abs/1708.05344, 2017.

[11] Peter F. Brown, Vincent J. Della Pietra, Stephen A. Della Pietra, and Robert L. Mercer. The mathematics of statistical machine translation: Parameter estimation. *Comput. Linguist.*, 19(2):263–311, June 1993. ISSN 0891-2017. URL http://dl.acm.org/citation.cfm?id=972470.972474.

[12] Han Cai, Jiacheng Yang, Weinan Zhang, Song Han, and Yong Yu. Path-level network transformation for efficient architecture search. *CoRR*, abs/1806.02639, 2018. URL http://arxiv.org/abs/1806.02639.

[13] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *CoRR*, abs/1812.00332, 2018.

[14] Patryk Chrabaszcz, Ilya Loshchilov, and Frank Hutter. A downsampled variant of imagenet as an alternative to the CIFAR datasets. *CoRR*, abs/1707.08819, 2017. URL http://arxiv.org/abs/1707.08819.

[15] Jasmine Collins, Jascha Sohl-Dickstein, and David Sussillo. Capacity and trainability in recurrent neural networks. *ArXiv*, abs/1611.09913, 2016.

[16] Eder Miranda De Novais, Thiago Dias Tadeu, and Ivandré Paraboni. Improved text generation using n-gram statistics. In *Proceedings of the 12th Ibero-American Conference on Advances in Artificial Intelligence*, IBERAMIA'10, pages 316–325, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16951-1, 978-3-642-16951-9. URL http://dl.acm.org/citation.cfm?id=1948131.1948173.

[17] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei. ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*, 2009.

[18] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Efficient multi-objective neural architecture search via lamarckian evolution. In *ICLR 2019*, 2019.

[19] Yarin Gal and Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, NIPS'16, pages 1027–1035, USA, 2016. Curran Associates Inc. ISBN 978-1-5108-3881-9.

[20] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015. URL http://arxiv.org/abs/1504.08083.

[21] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013. URL http://arxiv.org/abs/1311.2524.

[22] David E. Goldberg and Kalyanmoy Deb. A comparative analysis of selection schemes used in genetic algorithms. In *FOGA*, 1990.

[23] Alex Graves. Practical variational inference for neural networks. In *Proceedings of the 24th International Conference on Neural Information Processing Systems*, NIPS'11, pages 2348–2356, USA, 2011. Curran Associates Inc. ISBN 978-1-61839-599-3. URL http://dl.acm.org/citation.cfm?id=2986459.2986721.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

[25] Geoffrey E. Hinton and Drew van Camp. Keeping the neural networks simple by minimizing the description length of the weights. In *Proceedings of the Sixth Annual Conference on Computational Learning Theory*, COLT '93, pages 5–13, New York, NY, USA, 1993. ACM. ISBN 0-89791-611-5. doi: 10.1145/168304.168306. URL http://doi.acm.org/10.1145/168304.168306.

[26] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Comput.*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[27] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016. URL http://arxiv.org/abs/1608.06993.

[28] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

[29] Frederick Jelinek. *Statistical Methods for Speech Recognition*. MIT Press, Cambridge, MA, USA, 1997. ISBN 0-262-10066-5.

[30] Geoffrey E. Hinton Jimmy Lei Ba, Jamie Ryan Kiros. Layer normalization. *arXiv e-prints*, abs/1607.06450, 2016. URL https://arxiv.org/abs/1607.06450.

[31] Leslie Pack Kaelbling, Michael L. Littman, and Andrew P. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996. URL http://people.csail.mit.edu/lpk/papers/rl-survey.ps.

[32] Aaron Klein, Stefan Falkner, Simon Bartels, Philipp Hennig, and Frank Hutter. Fast Bayesian Optimization of Machine Learning Hyperparameters on Large Datasets. In Aarti Singh and Jerry Zhu, editors, *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics*, volume 54 of *Proceedings of Machine Learning Research*, pages 528–536, Fort Lauderdale, FL, USA, 20–22 Apr 2017. PMLR. URL http://proceedings.mlr.press/v54/klein17a.html.

[33] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'12, pages 1097–1105, USA, 2012. Curran Associates Inc. URL http://dl.acm.org/citation.cfm?id=2999134.2999257.

[34] Orr G.B. Müller KR. LeCun Y.A., Bottou L., editor. *Neural Networks: Tricks of the Trade*, London, UK, UK, 1998. Springer-Verlag. ISBN 3-540-65311-2.

[35] Xiangang Li and Xihong Wu. Constructing long short-term memory based deep recurrent neural networks for large vocabulary speech recognition. *CoRR*, abs/1410.4281, 2014. URL http://arxiv.org/abs/1410.4281.

[36] Chenxi Liu, Barret Zoph, Jonathon Shlens, Wei Hua, Li-Jia Li, Li Fei-Fei, Alan L. Yuille, Jonathan Huang, and Kevin Murphy. Progressive neural architecture search. *CoRR*, abs/1712.00559, 2017.

[37] Hanxiao Liu, Karen Simonyan, Oriol Vinyals, Chrisantha Fernando, and Koray Kavukcuoglu. Hierarchical representations for efficient architecture search. *CoRR*, abs/1711.00436, 2017.

[38] Hanxiao Liu, Karen Simonyan, and Yiming Yang. DARTS: differentiable architecture search. *CoRR*, abs/1806.09055, 2018.

[39] Renqian Luo, Fei Tian, Tao Qin, and Tie-Yan Liu. Neural architecture optimization. *CoRR*, abs/1808.07233, 2018.

[40] David J. C. MacKay. Bayesian interpolation. *Neural Computation*, 4(3):415–447, 1992. doi: 10.1162/neco.1992.4.3.415. URL https://doi.org/10.1162/neco.1992.4.3.415.

[41] Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. Building a large annotated corpus of english: The penn treebank. *Comput. Linguist.*, 19 (2):313–330, June 1993. ISSN 0891-2017.

[42] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *CoRR*, abs/1707.05589, 2017.

[43] Stephen Merity, Caiming Xiong, James Bradbury, and Richard Socher. Pointer sentinel mixture models. *CoRR*, abs/1609.07843, 2016.

[44] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. Regularizing and optimizing LSTM language models. *CoRR*, abs/1708.02182, 2017.

[45] Tomas Mikolov, Martin Karafiát, Lukas Burget, Jan Cernocký, and Sanjeev Khudanpur. Recurrent neural network based language model. *Proceedings of the 11th Annual Conference of the International Speech Communication Association, INTERSPEECH 2010*, 2:1045–1048, 01 2010.

[46] Geoffrey Miller, Peter M. Todd, and Shailesh U. Hegde. Designing neural networks using genetic algorithms. pages 379–384, 01 1989.

[47] Kevin P Murphy. *Machine learning: a probabilistic perspective*. Cambridge, MA, 2012.

[48] Radford M. Neal. *Bayesian Learning for Neural Networks*. PhD thesis, Toronto, Ont., Canada, Canada, 1995. AAINN02676.

[49] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. In *Proceedings of the 30th International Conference on International Conference on Machine Learning - Volume 28*, ICML'13, pages III–1310–III–1318. JMLR.org, 2013. URL http://dl.acm.org/citation.cfm?id=3042817.3043083.

[50] Hieu Pham, Melody Y. Guan, Barret Zoph, Quoc V. Le, and Jeff Dean. Efficient neural architecture search via parameter sharing. *CoRR*, abs/1802.03268, 2018.

[51] B. Polyak and A. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, 1992. doi: 10.1137/0330046.

[52] Esteban Real, Sherry Moore, Andrew Selle, Saurabh Saxena, Yutaka Leon Suematsu, Quoc V. Le, and Alex Kurakin. Large-scale evolution of image classifiers. *CoRR*, abs/1703.01041, 2017.

[53] Esteban Real, Alok Aggarwal, Yanping Huang, and Quoc V. Le. Regularized evolution for image classifier architecture search. *CoRR*, abs/1802.01548, 2018. URL http://arxiv.org/abs/1802.01548.

[54] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015. URL http://arxiv.org/abs/1506.01497.

[55] R. Rosenfeld. Two decades of statistical language modeling: where do we go from here? *Proceedings of the IEEE*, 88(8):1270–1278, Aug 2000. ISSN 0018-9219. doi: 10.1109/5.880083.

[56] Hasim Sak, Andrew W. Senior, and Françoise Beaufays. Long short-term memory recurrent neural network architectures for large scale acoustic modeling. In *INTERSPEECH*, 2014.

[57] Mingxing Tan, Bo Chen, Ruoming Pang, Vijay Vasudevan, and Quoc V. Le. Mnasnet: Platform-aware neural architecture search for mobile. *CoRR*, abs/1807.11626, 2018. URL http://arxiv.org/abs/1807.11626.

[58] Jr.; Finney Ross L. Thomas, George B., editor. *Calculus and Analytic Geometry (9th ed.)*, Addison Wesley, 1996. ISBN 0-201-53174-7.

[59] Martin Wistuba, Ambrish Rawat, and Tejaswini Pedapati. A survey on neural architecture search. *CoRR*, abs/1905.01392, 2019. URL http://arxiv.org/abs/1905.01392.

[60] Lingxi Xie and Alan L. Yuille. Genetic CNN. *CoRR*, abs/1703.01513, 2017. URL http://arxiv.org/abs/1703.01513.

[61] Zhilin Yang, Zihang Dai, Ruslan Salakhutdinov, and William W. Cohen. Breaking the softmax bottleneck: A high-rank RNN language model. *CoRR*, abs/1711.03953, 2017. URL http://arxiv.org/abs/1711.03953.

[62] Arber Zela, Aaron Klein, Stefan Falkner, and Frank Hutter. Towards automated deep learning: Efficient joint neural architecture and hyperparameter search. *CoRR*, abs/1807.06906, 2018. URL http://arxiv.org/abs/1807.06906.

[63] Zhao Zhong, Junjie Yan, and Cheng-Lin Liu. Practical network blocks design with q-learning. *CoRR*, abs/1708.05552, 2017. URL http://arxiv.org/abs/1708.05552.

[64] Hongpeng Zhou, Minghao Yang, Jun Wang, and Wei Pan. Bayesnas: A bayesian approach for neural architecture search. *CoRR*, abs/1905.04919, 2019. URL http://arxiv.org/abs/1905.04919.

[65] Julian G. Zilly, Rupesh Kumar Srivastava, Jan Koutník, and Jürgen Schmidhuber. Recurrent highway networks. *CoRR*, abs/1607.03474, 2016.

[66] Barret Zoph and Quoc V. Le. Neural architecture search with reinforcement learning. *CoRR*, abs/1611.01578, 2016.

[67] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. *CoRR*, abs/1707.07012, 2017. URL http://arxiv.org/abs/1707.07012.