Deep Convolutional Networks for Image Classification

by

Bing Xu

A thesis submitted in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

University of Alberta

Abstract

Image classification is an important problem in machine learning. Deep neural networks, particularly deep convolutional networks, have recently contributed great improvements to end-to-end learning quality for this problem. Such networks significantly reduce the need for human designed features in the image recognition process.

In this thesis I address two questions: first, how best to design the architecture of a convolutional neural network for image classification; and second, how to improve the activation functions used in convolutional neural networks. I review the history of convolutional network architectures, then propose an efficient network structure named TinyNet that reduces network size while preserving state of the art image classification performance.

For the second question I propose a new kind of activation function, called the Randomized Leaky Rectified Linear Unit, which improves the empirical generalization performance of the now widely used Rectified Linear Unit. Also, I make an explanation of the difficulty of training deep sigmoid network. The thesis culminates in a demonstration of the TinyNet architecture with Randomized Leaky Rectified Linear Units, which obtains state-of-art results on the CIFAR-10 image classification data set without any preprocessing.

To further demonstrate the generality of the results, I apply the general convolutional neural network structure to a different image classification problem, with completely different textures and shapes, and again achieve state-of-art results on a data set from the National Data Science Bowl competition.

Acknowledgements

I would like to thank Prof. Dale Schuurmans, Prof. Yoshua Bengio and Prof. Carlos Guestrin for supporting my research and providing truly freedom research environment.

I would like to thank Tianqi Chen, Ian Goodfellow, David Warde-Farley, Naiyan Wang, Mu Li, Ruitong Huang, Min Lin, Yifan Wu, Ying Xu and Fan Xie for your numerous help in both life and research difficulty. We make great projects together. Without you I cant image how the world would be like.

I would like to thank Jack Deng, Ben Hamner, Bangyong Liang, Yi Huang, Kyunghyun Cho, Cheng Wang, Robert Cowen, David Rousseau, Zhengdong Lu and Nicholas Leonard for your truly help & advice while I am facing H1-B catastrophe, which is the greatest challenge before I end school life.

At last I would like to thank my parents Yi Xu, Jinfeng Tong, my brother Lei Xu and my girlfriend Jingjing Xie. You give me continuous support and love while I am far away from you.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Machine learning is a subfield of artificial intelligence. A widely accepted definition of machine learning is: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E" [49].

For the image classification task (T), the experience E will consist of the raw pixels values of given images (in format of a matrix or multi-channel matrix). More accurately, in this work, E will be a set of single/multi-channel matrices in $\mathbb{R}^{c \times m \times n}$ where c is the channel, and m and n are the height and width correspondingly. The measurement P will be the classification error rate or multi-class log-loss.

For a long time, image classification has been considered to be a difficult task for machine learning. Researchers have had to design complex features in order to reduce the difficulty of training a classifier. Scale Invariant Feature Transform (SIFT) [45], Speeded Up Robust Features (SURF) [3], Histogram of Oriented Gradients (HoG) [15] and Fisher Vectors [54] have been quite popular feature extraction algorithms in the last decades. However none of these were able to push large scale image classification results to surpass human ability. Another drawback is that, for different datasets, researchers have had to design a different sets of features.

A natural question is whether can we design an end to end learning algorithm which is able to learn features, rather than classify based on human

expert designed features.

Three decades ago, the idea of "back-propagation" [58] suggested that a neural network could learn a hidden representation by running error propagation. A large number of end to end learning systems blossomed in the few years after the back-propagation algorithm appeared. Researchers designed Text-To-Speech [62], self-driving car controllers [56] and other neural network systems with little domain knowledge. Around that time, a special network structure—the convolutional network—was proposed and achieved state-of-art results in a handwritten number classification task [37].

In the 1990s and early 2000s, neural networks almost disappeared from researchers' attention. The reasons include:

1. hardware limits on computational power,
2. difficulty in training (gradient vanishing and exploding),
3. lack of theory for non-convex optimization,
4. learning from hand engineered features performed better in practice than end to end neural network training.

By the end of the 2000s, neural networks came back again. Hardware improvements, especially the development of heterogeneous computing, solved the first problem. Unsupervised pre-training and new activation functions made it easier to train a very deep networks, solving the second problem. Although there is still no breakthrough in non-convex optimization, in most cases, first order gradient optimization is good enough for training neural network; the third problem remains, but recent progress has not depended on it.

In 2012, three researchers from the University of Toronto, Alex Krizhevsky, Ilya Sutskever and Geoffrey Hinton, achieved a dramatic breakthrough in large scale image classification by using a deep neural network [34]. The deep neural network they used, which is later referred to as "AlexNet", is an end to end learning model. Without any human designed features, their neural network was able to learn a hierarchy of features from raw pixel values. Since then, in

many learning scenarios "deep learning" has become a standard approach in both industry and computer vision research.

However, AlexNet is not the end of the story. Researchers have trained deeper and deeper networks to achieve better classification results, even surpassing human level. There are now a few famous empirical network structures, but there is still a lack of general guidelines for designing network structures.

1.1 Thesis Contribution

There are three core contributions in this thesis:

1. I am the main contributor of a popular deep convolutional network toolkit, called cxxnet (`https://github.com/dmlc/cxxnet`). The cxxnet toolkit has been starred nearly 1000 times on Github. The cxxnet toolkit is designed for large scale image classification tasks. Compared to other popular toolkits like Caffe [31], cxxnet uses few library dependencies. Also it is the first distributed deep convolutional toolkit. This toolkit greatly reduces the barrier of using deep convolutional networks for image classification.

2. I then co-developed an even faster and friendlier general neural network toolkit after cxxnet, called "MXNet"(`https://github.com/dmlc/mxnet`). MXNet has been starred more than 2500 times. MXNet provides a consistent user experience for developing deep learning algorithms of different flavors. A user can construct a neural network from a symbolic interface, or build it from basic tensor operations, or use a mixed approach. More details will be provided in Chapter 3.

3. I performed a full investigation of deep convolutional network structure design. I have surveyed famous network structures and some unpublished network design theories, then designed a new structure called TinyNet. The TinyNet model greatly reduces computational complexity while still achieving state-of-art performance. I have successfully deployed this network on an Android smartphone.

4. I propose a new randomized activation function that empirically performs well on many datasets. In all my experiments, the new randomized activation function works better than the widely used rectified linear unit.

5. I figured out the reason why sigmoid networks fail in training deep neural networks, and solved the sigmoid network training problem without pre-training.

The main hypotheses that I investigated in this thesis are:

1. Can a practical deep learning toolkit be developed that supports efficient distributed training, simple tensor and symbolic interface, and efficient execution on mobile devices? Can an alternative activation function be devised that surpasses the state of the art rectified linear unit (ReLU) in both convergence speed and classification accuracy?

The main claims this thesis supports are:

1. TinyNet is the first network architecture to achieve state-of-art image classification result with a model size under 10MB, making it suitable for deployment on a mobile device.

2. The difficulty of training a deep sigmoid network is due primarily to improper initialization and poor learning rate choices, rather than the saturation properties of the sigmoid function.

The outline of this thesis is:

- Chapter 2 introduces some background on machine learning, neural networks, heterogeneous computing and the design of cxxnet.

- Chapter 3 compares the designs of cxxnet and MXNet, then compares the performance of MXNet and cxxnet, showing that MXNet achieves significantly better results than cxxnet.

- Chapter 4 reviews the topic of structure design, describing what I have learned about these structures. By using the lessons learned, I designed a network called TinyNet, which is tiny enough to run on a mobile device with state-of-art result.

- Chapter 5 empirically evaluates the most popular activations on TinyNet. I have found that the most widely used activation, the Rectified Linear unit, is not the best choice. I propose a new kind of activation function that exploits randomness to reduce over-fitting.

- Chapter 6 applies deep convolutional neural networks to the problem of large scale image classification. I achieve state-of-art results on the plankton classification problem.

1.2 Publications and Public Competitions

This thesis is based on the following peer-reviewed papers:

- Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Zheng Zhang. MXNet: A Distributed Deep Learning Framework for Efficiency and Flexibility, Learning System Workshop in Advances in Neural Information Processing Systems (NIPS), 2015. (Chapter 3)

- Bing Xu, Naiyan Wang, Tianqi Chen and Mu Li. Empirical Evaluation of Rectified Activations in Convolutional Network, Deep Learning Workshop, International Conference of Machine Learning (ICML), 2015. (Chapter 5)

The software and network structure design method were used in public competitions:

- National Data Science Bowl, 2015. 2nd place (out of 1049).
  https://www.kaggle.com/c/datasciencebowl
  Total Prize: $175,000

Chapter 2

Background

2.1 Machine Learning and Neural Networks

Machine learning is a subfield of artificial intelligence. Machine learning aims to design algorithms that are able to learn from data automatically, and reduce the workload of human beings.

There are three kinds of machine learning: supervised learning, unsupervised learning and reinforcement learning. In this thesis, we focus on supervised learning.

Various approaches to supervised learning problems can be treated as selecting an appropriate loss function and regularizer, then selecting a suitable optimization method to minimize their combination.

Successful application of machine learning involves many factors, from theoretical correctness to engineering. Engineering includes two perspectives: feature engineering and software engineering. Feature engineering requires domain experts to design special features based on given data to make machine learning algorithms work better, while software engineering makes training possible on large scale data.

Although machine learning combined with human feature engineering has been a successful approach in the past, an important goal is to learn features from data directly rather than from human designers every time. We refer to learning features from raw data as representation learning. Both supervised learning and unsupervised learning can involve representation learning.

Although there are many learning algorithms that are able to learn repre-

sentations, currently neural networks are the most successful and widely used approach.

2.1.1 Loss Function

In classification, cross-entropy and negative log likelihood (NLL) are often used as the loss. Also, we can use Kullback-Leibler divergence (KL-Divergence) as the loss function. Formally, cross-entropy is defined as:

$$\varepsilon(y_i, \hat{y_i}) = \begin{cases} -\log(\hat{y_i}) & \text{if } y_i = 1 \\ -\log(1 - \hat{y_i}) & \text{if } y_i = 0 \end{cases} \tag{2.1}$$

We can treat the NLL loss as an extension of the cross-entropy loss from binary to multi-class classification. If we assume y is a one-hot vector that indicates which class an example belongs to, where i is label, $y_i = 1$ and otherwise is 0, The NLL loss is

$$\varepsilon(y, \hat{y}) = -\sum_{i}^{N} y_i log(\hat{y_i}) \tag{2.2}$$

where N is the total number of classes.
The KL-divergence $D_{KL}(P||Q)$ is a measure of the information lost when Q is used to approximate P. Formally, it is described as:

$$\begin{aligned} \varepsilon(y, \hat{y}) &= D_{KL}(P||Q) \\ &= \sum_{i} log\left(\frac{Q(y|x)}{P(y|x)}\right) Q(y|x) \\ &= \sum_{i} \log\left(\frac{y_i}{\hat{y_i}}\right) y_i \end{aligned} \tag{2.3}$$

The gradient of the KL-divergence loss is:

$$\begin{aligned} \frac{\partial \varepsilon_{KL}(y_i, \hat{y_i})}{\partial \hat{y_i}} &= \frac{\partial}{\partial \hat{y_i}} y_i(\log(y_i) - \log(\hat{y_i})) \\ &= \frac{\partial}{\partial \hat{y_i}}(-y_i \log(\hat{y_i})) \\ &= \frac{\partial \varepsilon_{NLL}(y_i, \hat{y_i})}{\partial \hat{y_i}} \end{aligned} \tag{2.4}$$

From Equation 2.4 we know that minimizing negative log likelihood loss is exactly the same as minimizing KL-divergence.

2.1.2 Logistic Regression and Softmax Regression

Logistic regression has been proposed as a method for estimating the probability of an event as a function of several independent variables [70]. It has been widely used in real world applications to learn binary classifiers for tasks such as spam filtering [8], social network analysis [40] and click prediction [57].

Logistic Regression uses a logistic function to transform the output of a linear function into the range $[0, 1]$. Formally we have

$$\hat{y} = \sigma(h(x)) \tag{2.5}$$

where $\hat{y}$ is the output of logistic regression, σ is the logistic function:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.6}$$

and $h(x)$ is linear transform function:

$$h(x) = W^T x + b, \text{ where } W \in \mathbb{R}^N, b \in \mathbb{R} \tag{2.7}$$

Softmax regression is an extension of logistic regression for multi-class classification problems. The softmax function is defined as:

$$\hat{y}_i = \text{softmax}(x) = \frac{e^{W_i^T x + b_i}}{\sum_j e^{W_j^T x + b_j}} \tag{2.8}$$

where $\hat{y}_i$ is the i^{th} outputs probability prediction.

For numerical stability, usually we use

$$\hat{y}_i = \text{softmax}(x) = \frac{e^{W_i^T x + b_i - m}}{\sum_j e^{W_j^T x + b_j - m}} \tag{2.9}$$

where $m = max_j(W_j^T x + b_j)$. The output $\hat{y}_i$ is a number in the range $[0, 1]$ which indicates the probability that the input example belongs to class i.

After we select a classification model, we can select a suitable loss. For logistic regression we can use the cross-entropy loss. For softmax regression, we can use the NLL loss. The final step is choosing an optimization method to minimize the loss.

2.1.3 Gradient Descent

Gradient descent is a first-order optimization algorithm used to find a local minimum of an objective function. The gradient descent algorithm is given in Algorithm 1.

input : loss function ε, learning rate η, dataset with N examples X, y, model $F(\theta, x)$
output: An optimal θ which minimize ε
while *not converged* **do**
  $\hat{y} \leftarrow F(\theta, x)$;
  $\theta \leftarrow \theta - \eta \cdot \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \varepsilon(y, \hat{y})}{\partial \theta}$;
end

Algorithm 1: Gradient Descent

If training time is a bottleneck, we will use the stochastic gradient descent method [6] given in Algorithm 2.

input : loss function ε, learning rate η, dataset X, y, model $F(\theta, x)$
output: An optimal θ which minimize ε
while *not converge* **do**
  Shuffle X, y;
  foreach x_i, y_i *in* X, y **do**
  $\hat{y}_i \leftarrow F(\theta, x_i)$;
  $\theta \leftarrow \theta - \eta \cdot \frac{\partial \varepsilon(y_i, \hat{y}_i)}{\partial \theta}$;
  ;
end

Algorithm 2: Stochastic Gradient Descent

There is also a popular form of gradient descent called "mini-batch" gradient descent that divides the training data into many small batches of size 64, 128 or more, then runs stochastic gradient descent over each mini-batch (computing the sum of gradients within each mini-batch) to optimize the model.

Choosing the correct mini-batch size is also a factor in successfully optimizing by using gradient descent.

2.1.4 Multi-Layer Perception

Logistic regression and softmax are linear classification algorithms that will fail when the data is not linearly separable. To solve such problems, we can

use manually crafted feature, kernels, boosting or multi-layer perceptions.

In general, a multilayer perception model stacks multiple non-linear transformations. Each layer is in the form of Equation 2.10

$$h_n = \sigma(W_n^T h_{n-1} + b_n) \tag{2.10}$$

where h_n is the output of the n^{th} layer, $\sigma(x)$ is a nonlinear activation function, and W_n, b_n are the parameters for this layer. For h_0, h_{n-1} is the input data X.

On top of a multi-layer perceptron sits a logistic transform layer or softmax transform layer to transform output into the range $[0, 1]$. The gradient of the loss function can then be propagated down the model to update the parameters.

If we set all of the transformations σ to the sigmoid function (Equation 2.6), we can treat the multi-layer perceptron as multiple layers of logistic regression models.

By simply using a two layer network, we can visualize how the network transforms the input [4] [12].

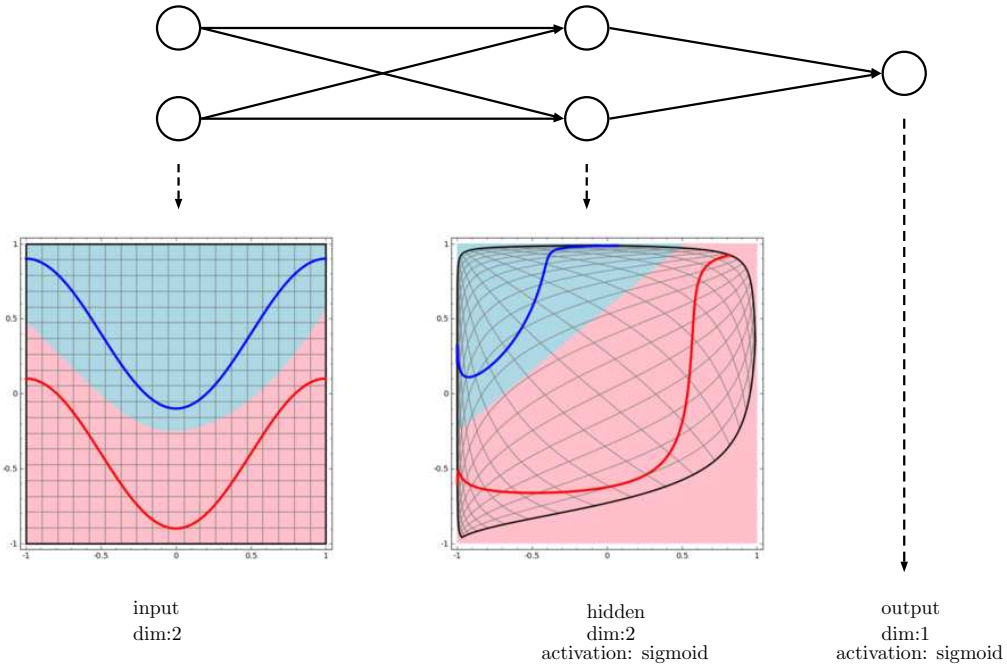

Figure 2.1: A two hidden unit network transform

From Figure 2.1 we find that non-linearly separable data is transformed

into linearly separable data in the hidden layer without any human designed features.

The data representation in hidden layer is also called a learned "representation". Successful examples of learning by using a learned representation in hidden layers includes image tracking [72], image captioning [69] and word embedding [48].

2.1.5 Convolutional Network

While a fully connected multilayer perceptron is a universal function approximator that can approximate any continuous function on compact subsets of $\mathbb{R}^n$ [29], there are three problems with using a shallow architecture:

First, for real world high resolution images, using a fully connected layer will make the number of parameters in the model explode. For example, for a color image with 224x224 pixels and 3 color channels, if we want to map it to 256 dimension by using a single layer fully connected network, the weight matrix will have 38 million parameters. It will take too much memory and computation power to compute such a large matrix multiplication.

Second, a fully connected network is prone to over-fitting [65]. Although it is able to approximate any function, it is also able to learn a function that exactly fits any noise in the training data, which will lead to poorly fitting the test data.

Third, a fully connected network is only a general function approximator that does not exploit any domain knowledge. It would be more efficient if we could design a network that incorporated some domain knowledge.

A convolutional network is able to address all three of these problems. Yann LeCun and other researchers originally proposed convolutional networks as part of a famous network structure, LeNet-5, that was used to successfully solve the handwritten digital classification problem [38]. Formally, for a two dimensional image I and a two dimensional convolution kernel $K^{m,n}$, the

output S is:

$$s[i,j] = (I * K)[i,j] = \sum_m \sum_n I[m,n]K[i-m, j-n]$$
$$= \sum_m \sum_n I[i-m, j-n]K[m,n] \tag{2.11}$$

Additionally, a stride parameter can be used to specify how far the kernel is moved between successive measurements of the input image. To illustrate, consider Figures 2.2 and 2.3, which show how a 1x3 kernel with different stride lengths is applied to a 1x6 input.

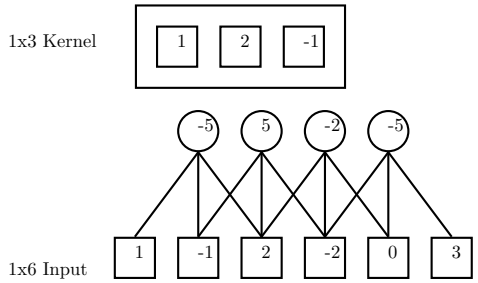

Figure 2.2: Stride 1 convolution example

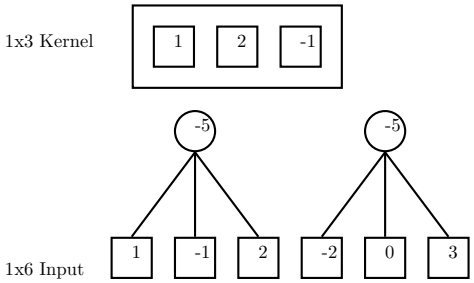

Figure 2.3: Stride 3 convolution example

The use of convolution leverages three important ideas to aid learning: sparse interaction, parameter sharing and equivariance of representation [4]. Sparse interaction arises from using a kernel that is smaller than the input, in which case each output is only connected to a small region of the input instead of the entire image. Parameter sharing arises from the use of the same kernel everywhere. In this way, a convolutional network greatly reduces the number of parameters, which reduces the risk of over-fitting. Equivariance refers to a property where the magnitude of a change to the input will result in the same magnitude of change to the output. In image processing, blurring, sharpening and edge-detection can all be performed by using a convolution operation;[1] by using a convolutional network, we are therefore able to learn an optimized kernel for image classification. These properties of convolutions address the three problems identified above. Figure 2.4 shows two convolutional network

[1]Examples of convolution operations on images can be found at: `https://en.wikipedia.org/wiki/Kernel_(image_processing)`

results with a random 3x3 kernel. From the example we can see that even a random kernel is able to detect edges and levels in images.

Figure 2.4: Convolution with random kernel

Pooling is another important idea that can be used in convolutional networks. Pooling is also called "down sampling" if the stride is greater than 1. Instead of using a weighted kernel, pooling uses either a max or average function to calculate a summary over a region. In particular, for each region R, the output of an averaging pooling unit is the mean of element a in the region:

$$s_j = \frac{1}{|R_j|} \sum_{i \in R_j} a_i \tag{2.12}$$

while the output of a max pooling unit is:

$$s_j = \max_{i \in R_j} a_i \tag{2.13}$$

Randomness can also be used in pooling, although it is not very popular yet. Interestingly, Stochastic Pooling [73] and Fractional Max-Pooling [21] have been shown to have some regularization effect.

Applying pooling will lose local responsiveness, as shown in Figure 2.5 (left is the original image, and right is a max-pooled image). The receptive field of a neuron is defined over the input image; for example, as shown in Figure 2.6. In this example, which uses a 1x3 kernel for convolution, the receptive field of a neuron in the first layer is 3, while the receptive field for a neuron in the second layer is 9, which covers the entire input image in this example.

Figure 2.5: Max Pooling example

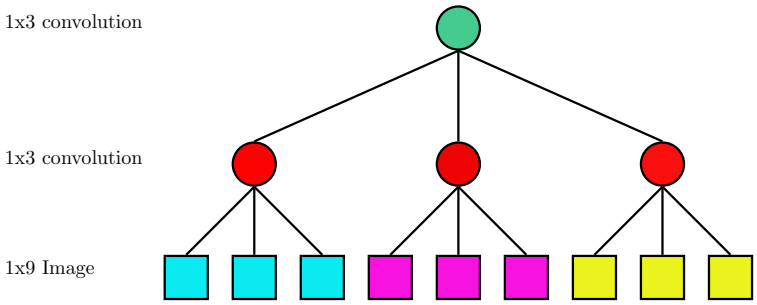

Figure 2.6: Receptive field example

2.1.6 Activation function

Without using non-linear activation functions in each layer, both fully connected and convolution neural networks merely compute linear transformations, hence no matter how many layers are stacked, such a model would still just implement a linear function.

The sigmoid function has been used as the standard nonlinear transformations in neural networks for more than 30 years, but it turns out that training neural networks with these activations is hard. Recent results have suggested that piece-wise linear functions are a better choice in terms of making optimization easier. I discuss this issue in more depth in a later chapter.

2.1.7 Back-propagation

Back-propagation [59] is the fundamental method used for training neural network models. By using back-propagation, one can easily calculate the gradient of the parameters from the top to the bottom layers. The ability to efficiently

compute the gradient allows one to use gradient descent to optimize the parameters in the whole network. For a multi-layer perception, if the network structure is a directed acyclic graph, we can simply use the chain-rule to efficiently calculate the gradient from the top to the bottom layers, as shown in Algorithm 3.

input : A network of l layer, each layer's activation function σ_l and output $h_l = \sigma_l(W_l^T h_{l-1} + b_l)$, output of the network $\hat{y} = h_l$
Compute gradient g of output layer:
$g \leftarrow \frac{\partial \varepsilon(y,\hat{y})}{\partial \hat{y}}$;
for $i \leftarrow l$ **to** 0 **do**
  Calculate gradient for current layer;
  $\frac{\partial \varepsilon(y,\hat{y})}{\partial W_l} = \frac{\partial \varepsilon(y,\hat{y})}{\partial h_l}\frac{\partial h_l}{\partial W_l} = g\frac{\partial h_l}{\partial W_l}$;
  $\frac{\partial \varepsilon(y,\hat{y})}{\partial b_l} = \frac{\partial \varepsilon(y,\hat{y})}{\partial h_l}\frac{\partial h_l}{\partial b_l} = g\frac{\partial h_l}{\partial b_l}$;
  Do gradient descent using $\frac{\partial \varepsilon(y,\hat{y})}{\partial W_l}$ and $\frac{\partial \varepsilon(y,\hat{y})}{\partial b_l}$;
  Propagate gradient to lower layer;
  $g \leftarrow \frac{\partial \varepsilon(y,\hat{y})}{\partial h_l}\frac{\partial h_l}{\partial h_{l-1}} = g\frac{\partial h_l}{\partial h_{l-1}}$;
end

Algorithm 3: Back-propagation

Algorithm 3 shows back-propagation for a single path network. For a multi-path network, a topological sort is required to determine the back-propagation sequence. For a layer with multiple outputs, the gradient is the sum of all gradients propagated down from above.

2.1.8 Momentum

Momentum [55] is a method for accelerating gradient descent, particularly stochastic gradient descent. The idea is to use a moving average of the parameter gradient instead of just using the current real gradient:

$$v_t = \beta v_{t-1} - \eta \bigtriangledown F(\theta_{t-1}) \tag{2.14}$$

$$\theta_t = \theta_{t-1} + v_t \tag{2.15}$$

where θ denotes the model parameters, v is the momentum of the gradient, β is a momentum factor, and η is the learning rate for the t^{th} round of training. Usually, we use a fixed momentum factor of 0.9 during all of the training processes.

Nesterov's Accelerated Gradient [51] is also widely used. Although it was originally derived for non-stochastic gradients, the derivation of Nesterov's Accelerated Gradient as momentum for stochastic gradient descent can be derived as [67]:

$$v_t = \beta v_{t-1} - \eta \bigtriangledown F(\theta_{t-1} + \beta v_{t-1}) \tag{2.16}$$

$$\theta_t = \theta_{t-1} + v_t \tag{2.17}$$

Nesterov's Accelerated Gradient is able to help successfully train neural network models without the need for sophisticated second-order methods [67].

2.1.9 Learning Rate Schedule

Generally, if we use a large learning rate η (Equation 2.14, Equation 2.16, Algorithm 1, Algorithm 2), we can make training converge faster, but too large a choice of η will lead to divergence. Small η values are more likely to get trapped in local minima. To ensure that training converges, we need to reduce the learning rate while training [6].

There are three commonly used methods to schedule the decrement of the learning rate: constant, factored, and exponential decay. The learning rate is scheduled to change after ς steps.

The constant schedule reduces learning rate according to a manually defined step function with arbitrary ς.

The exponential schedule calculates the learning rate as:

$$\eta_t = \eta_0 \cdot \gamma^{t/\varepsilon} \tag{2.18}$$

where η_0 is the initial learning rate, t is the current training round, and γ is a decay factor in the range $(0, 1)$

The factored schedule is similar to the exponential schedule but in a step function format:

$$\eta_t = \eta_0 \cdot \gamma^{\lfloor t/\varepsilon \rfloor} \tag{2.19}$$

By using the constant or factored schedules, we can easily judge the learning rate decay effect by visualizing the learning curve. Commonly, one uses $\gamma = 0.1$ to reduce the learning rate 10 times at each stage.

2.1.10 Weight Decay

Weight decay is also commonly used in the neural network community to perform L2 regularization during training. Regularization is essential for preventing over-fitting and improving model generalization. Formally, the L2 regularizer for a function $F(\theta, x)$ is given by:

$$\Omega(\theta) = ||\theta||^2 \tag{2.20}$$

$$\hat{\varepsilon}(F(\theta, x), y) = \varepsilon(F(\theta, x), y) + \frac{1}{2}\lambda\Omega(\theta) \tag{2.21}$$

The resulting gradient for weight θ is:

$$\frac{\partial \frac{1}{2}\lambda\Omega}{\partial\theta} = \lambda \cdot \theta \tag{2.22}$$

Weight decay is the default regularization method used for training neural networks. Usually we set λ to 0.0004. If there is extra normalization, such as Dropout and Batch Normalization (which will be discussed in a later chapter), a smaller λ will accelerate training.

2.1.11 Dropout

Dropout is a regularization method first proposed in [26]. Dropout dramatically reduces the overfitting of neural networks and improves generalization performance. Dropout has quickly become a default component of neural network training methods. Formally, training with Dropout for layer l is expressed by:

$$r_j^{(l)} = \text{Bernoulli}(p) \tag{2.23}$$

$$\hat{h}^{(l)} = r^{(l)} * h^{(l)} \tag{2.24}$$

where $h^{(l)}$ is the original output of layer l, and $\hat{h}^{(l)}$ is the new output of layer l with Dropout. To obtain a deterministic result at test time, we use the expectation as output:

$$\mathbb{E}[\hat{h}_j^{(l)}] = p * 0 + (1 - p) * h_j^{(l)} \tag{2.25}$$

$$= (1 - p) * h_j^{(l)} \tag{2.26}$$

The expectation is an approximate average of exponentially many dropped out models [65]. This also explains why using Dropout will improve generalization performance, since it can be interpreted as an average over a large ensemble.

2.1.12 Pre-training

Training a deep neural network with a pre-training technique was responsible for renewing interest in "deep learning". Layer-wise pre-trained Restricted Boltzmann Machines [25] and pre-trained auto-encoders [5] were used as weight initializers for supervised deep network models. However, pre-training is now rarely used, since a better understanding of the appropriate activation functions and better random initialization [18] [24] has made direct supervised training more effective.

2.2 Heterogeneous Parallel Computing

Training a neural network requires a massive amount of floating point operations. However, current central processing units (CPUs) are still not fast enough for floating point computing. At Google, a distributed system with at least 2,000 CPUs, called "DistBelief" [17], is used for training deep neural networks. Luckily, for others, we are able to build similar distributed systems by accessing high performance graphics processor units (GPUs) as an alternative to using a massive number of CPUs. We refer systems that use more than one kind of processor as heterogeneous computing [63]. The idea of heterogeneous computing comes from the beginnings of personal computing. Thirty years ago, the first heterogeneous processor, the Intel 8087, known as the "x87 floating-point coprocessor", provided extra float point computing power to Intel 8086 processor to accelerate its mathematical operations. Modern GPUs are able to run many more threads than a CPU. For example, an NVIDIA K80 GPU has 4,992 Compute Unified Device Architecture (CUDA) cores. By running in parallel, this card is able to provide 8.76T single precision floating-point operations per second (FLOPS) [52]. By comparison, one of the most advanced CPUs, the Intel Xeon E5-2699v3 CPU with 18 cores, is able

to provide 0.8T FLOPS [47], which is 10 times slower than a single advanced GPU. The floating point performance of a single K80 GPU is as capable as IBM ASCI White, the fastest supercomputer in the world in 2001 [36].

To train a neural network, we need to accelerate three kinds of operations:

1. element-wise operations, eg. activation,
2. matrix multiplication, eg. $W^T x$, and
3. convolution and max-pooling operations.

NVIDIA provides a highly optimized library cuDNN [11] for standard activation functions, and fully connected, convolution and pooling layers. For special operations, we need to write our own parallel CUDA-C programs.

In a CUDA-C program, a special function, called a "Kernel", is used to specify a single thread function. After we set the grid and thread block to indicate how to map a thread to data, we can then launch the kernel to calculate result [32].

2.3 Benchmark Datasets

To evaluate image classification performance, I consider some standard datasets. I choose two datasets to use as benchmarks: CIFAR-10 and ImageNet. CIFAR-10 is used for fast evaluation classification performance on small images; ImageNet is used for large scale evaluation on a real-world high-resoulution image classification problem. I did not choose MNIST here because even a simple convolutional network is able to achieve 0.77% error on the test set; it is meaningless to fit such a saturated dataset.

2.3.1 CIFAR-10 and CIFAR-100

The CIFAR-10 and CIFAR-100[2] dataset [33] is a tiny natural image dataset that contains 10/100 classes. Each image in CIFAR-10 and CIFAR-100 is a 32x32 RGB image (Figure 2.7). There are 50,000 training images and 10,000

[2]CIFAR-10 and CIFAR-100 Homepage: `http://www.cs.toronto.edu/~kriz/cifar.html`

test images. Images are taken from natural scenes, then reduced to a small size and low resolution.

The weakness of the CIFAR-10 and CIFAR-100 dataset is obvious: the images are quite different from real world images, both in size and resolution. Some of the images are even unrecognisable for a human being.

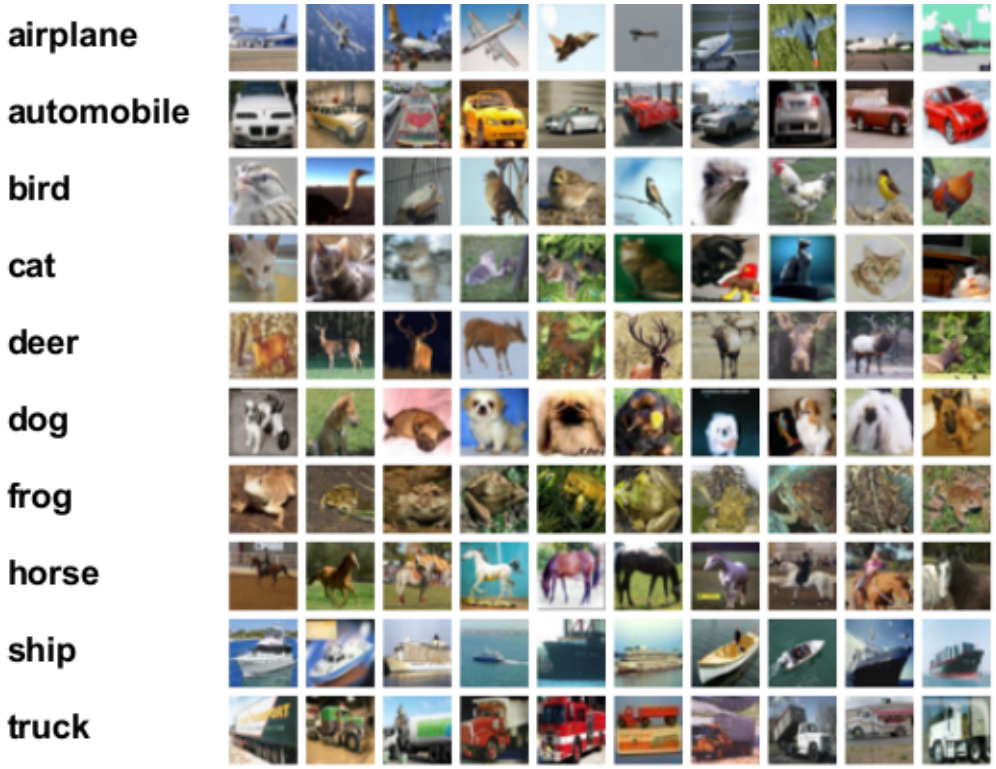

Figure 2.7: CIFAR-10 and CIFAR-100 classes and sample image (Taken from CIFAR-10 homepage)

There are special normalization methods for improving classification performance on CIFAR-10/100, including global contrast normalization and ZCA whitening [33]. In my experiment settings, I focus on representation learning and perform none of these special preprocessing steps. All networks are trained directly from raw pixel values.

The tests errors of reference models are given in Table 2.1.

2.3.2 ImageNet

ImageNet is a dataset with high resolution images of natural scenes. In total there are 14,197,122 images in 21,841 classes. The classes are organized

Model	CIFAR10	CIFAR-100
Human [1]	6%	n/a
Deeply-Supervised Nets[39]	8.23%	34.57%
Maxout Network [20]	10.35%	38.57%
Stochastic Pooling Convnet[73]	15.13%	n/a
Dropout Convnet[26]	15.6%	n/a
SVM with HoG, etc manual feature	58%	n/a
KNN with Hog, etc manual fetaure	75%	n/a

Table 2.1: CIFAR-10 and CIFAR-100 reference test-error

into a WordNet hierarchical structure, where each node of the hierarchy is depicted by hundreds or thousands of images[3]. The ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [60] is an image classification contest that led to the creation of this dataset. Here I use "ImageNet" to indicate the ILSVRC classification and localization dataset, which is a subset of the full ImageNet dataset. This subset contains 1.28 million image in 1,000 classes for training, 50 thousand images for validation, and 100 thousand images for testing. The labels for the test data are isolated on a test server, while the validation labels are accessible to the public.

Figure 2.8: Eskimo husky class sample

Figure 2.9: Alaskan malamute class sample

ILSVRC is the largest openly accessible image dataset with labels. Figure 2.9 shows two images from two different classes in the ImageNet dataset. It is considered to be the hardest image classification problem currently available. Error is measured by top-5 error, which means each model should generate 5

[3]ImageNet: `http://www.image-net.org/`

top predictions for a given image; if one of these 5 predictions is correct, there will be no penalty.

Model	Top-5 Error (Ensemble)
Inception-BN [30]	4.90%
Deeper VGG/PReLU [24]	4.94%
Human [60]	5.1%
GoogLeNet (2014 Winner) [68]	6.7%
VGG (2014 2nd)[64]	7.3%
Clarifai (2013 Winner, no open method)	11.7%
AlexNet (2012 Winner)[34]	16.4%
Fisher Vector + SVM (2011 Winner) [53]	25.7%

Table 2.2: ILSVRC reference validation top-5 error

Table 2.2 shows the top-5 errors for reference models. Note that these results are all generated by multi-model averaging and single image multi-test. Single model and single image test results will produce worse scores.

Chapter 3

Toolkit System Design

3.1 Introduction

Practical toolkits are important in current deep learning research. High performance and flexible toolkits have significantly boosted the research and application progress of deep learning.

Most open source deep learning systems are wrappers on top of a low-level heterogeneous numerical library, for example Lasagne[1] and Keras[2]. In this chapter, I will focus on two systems I co-developed that fully support both low-level and high-level functions.

The main challenge of toolkit system design includes:

1. Cross domain knowledge.

   Designing a good deep learning toolkit requires knowledge in deep learning, numerical computing, heterogeneous parallel computing, system I/O and distributed computing.

2. Balance in flexibility and performance.

   For a special use toolkit, for example, supporting only linear models, it is easy to optimize performance. However, deep learning models require flexible architectures. Flexibility raises the requirements for memory optimization and computation optimization.

[1]Lesagne: https://github.com/Lasagne/Lasagne
[2]Keras: https://github.com/fchollet/keras

As part of my thesis research I co-developed two deep learning system: cxxnet[3] and MXNet[10].

The cxxnet project was started in 2013 by Tianqi Chen and me, and ended in mid 2015. For the cxxnet project, I designed and implemented the I/O module, updater module and layer module, and some of the backend heterogeneous code. The cxxnet toolkit has been starred by more than 1,000 times on Github, notable users include UCSD and Tencent.

The MXNet project started in mid 2015. For the MXNet project, I mainly designed and implemented the operator module and the NDArray module. I also contributed to MXNet on Android and MXNet.js (the JavaScript version of MXNet). The MXNet has been starred more than 2,500 times on Github.

3.2 cxxnet

3.2.1 Module Design

The goal behind designing a deep convolutional neural network toolkit was to make a reusable toolkit with minimal dependencies. Also I wanted to modularize the toolkit to make it easy to use as a fundamental infrastructure in real systems. None of the currently available open source toolkits satisfied these demands, so, with my collaborators, I decided to develop such a toolkit.

At a high level, cxxnet is composed of four separable modules (in UML format, Figure 3.1):

- Data Iterator Module
- Layer/Connection Module
- Updater Module
- Network Module

The data iterator module is used to load serialized data. Instead of using a database, I find that a binary fixed size sequence storage is simple but efficient enough. Also contiguous loading from a disk with threaded cache can maximize

[3]cxxnet: https://github.com/dmlc/cxxnet

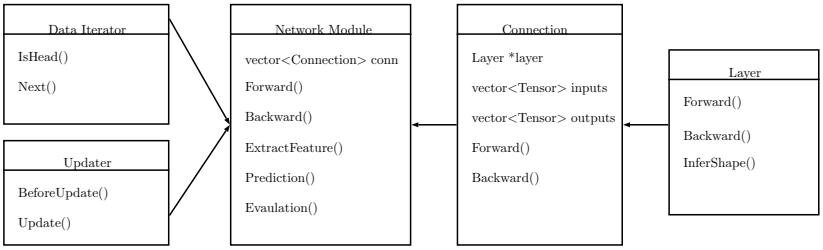

Figure 3.1: Overview of simplified system

the I/O performance. By using this design, I can achieve similar performance to using SSD directly, while also avoiding the use of a database.

Layers and connections form the basic elements in a neural network. The layer/connection module is highly reusable with a uniform logic. The layer component provides the standard forward and backward operations. I have made the layer stateful in cxxnet, which means that all weights, gradients and internal random masks are stored inside of a layer.

The updater module provides an abstraction of first order gradient optimization methods. For a given weight and gradient, it runs gradient descent or Nesterov's accelerated gradient algorithm. The updater is also associated with a parameter server to support multi-GPU or distributed training.

The network module provides a way to schedule the running of the layers. This can be organized by a simple queue or a directed acyclic graph. An example is shown in Figure 3.3. The schedule is determined by a topological sort of the dependency graph. For forward propagation through the network, the schedule ensures sequential running layer by layer. For back propagation, it follows the reverse dependence order. The network module is also treated as glue for different modules: the data iterator feeds a data batch to the network, the network module conducts forward and backward propagation of each layer, then the updater is scheduled to update weights in each layer. Some additional wrapper functions like prediction and feature extraction are also provided in network module.

Each module instance is implemented by inheriting from an interface class so that a module class can be replaced without any other change to the code.

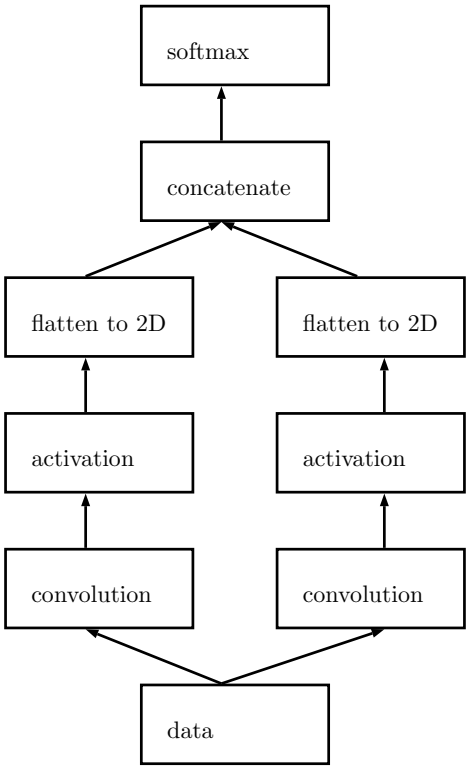

Figure 3.2: Forward pass computation graph of a sample neural network with 2 branches.

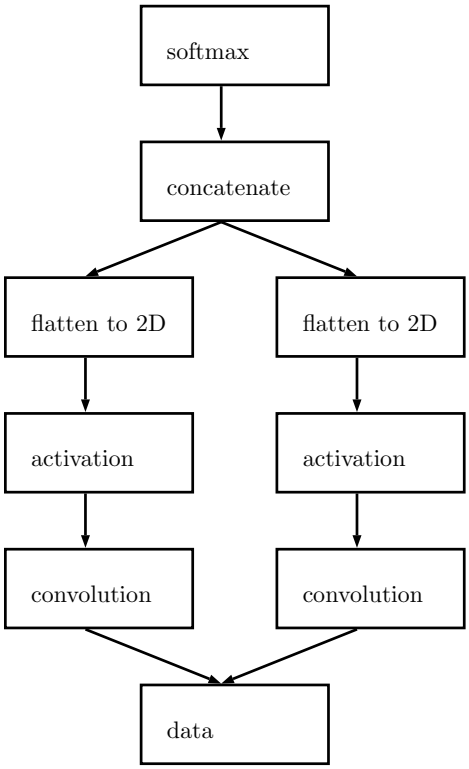

Figure 3.3: Backward pass computation graph of a sample neural network with 2 branches.

3.2.2 Multi-GPU and distributed support

As the complexity of neural networks is growing, and we have more data available for training, it has become infeasible to train on a single GPU; we need multiple GPUs to make training faster.

There are two ways to utilize a multi-GPU architecture: model parallelism and data parallelism. The main idea behind model parallelism is to partition the model, then use different workers to train different parts of model; for example, by running different layers on different cards. Data parallelism is achieved by partitioning the data; for example, by dividing a large batch into several smaller batches, then using different workers to train on different batches.

The benefit of model parallelism is that the model will converge in the same manner as training on one card. However, transmitting data between

different cards will require high traffic bandwidth. Moreover, a busy card can hold up other cards if its results are not ready. Therefore, it is hard to achieve a reasonable speed-up ratio by just using model parallelism.

Unlike model parallelism, data parallelism involves using each card to run a full model but with smaller batch sizes. A full batch is partitioned into smaller batches, then run on different cards. In this way each batch can be processed at full speed; the drawback is that the weights have to be synchronized after each local gradient update. During synchronization, all cards remain idle while we transmit the model weights between cards. If the model is large, this will also require very high traffic bandwidth. There is also a problem with ensuring convergence to a high quality solution if we do too many data partitions.

In cxxnet, I decided to use data parallelism because recent network structures have been significantly reduced and involve reasonable numbers of parameters. This allows nearly a linear speedup to be achieved by using data parallelism.

3.3 MXNet

After developing cxxnet, I then developed a second generation toolkit, MXNet, with my collaborators. MXNet combines the advantages of other recent toolkits. A comparison is shown in Table 3.1.

MXNet was initiated and designed in collaboration with authors from the cxxnet project, Minerva [71] and purine2 [44]. The MXNet project reflects what we have learned from these past projects. It combines the best aspects of existing toolkits, while being efficient, flexible and memory efficient.

3.3.1 System Design

Tensor Interface

MXNet provides a foundation for tensor computation that works closely with the binding languages's own tensor libraries. For example, MXNet's python binding is compatible with `numpy.ndarray`. Furthermore, MXNet simplifies GPU programming; an example is shown in the left of Figure **??**. In ad-

	MXNet[10]	Caffe [31]
Core Language	C++	C++
Binding Language	Python/R/Julia/JavaScript	Python/Matlab
Device	CPU/GPU/ARM	CPU/GPU
Multi-machines	$\surd$	$\times$
Tensor Interface	$\surd$	$\times$
Symbolic Differentiation	$\surd$	$\times$
	Torch7 [13]	Theano [2]
Core Language	Lua	Python
Binding Language	-	-
Device	CPU/GPU/FPGA	CPU/GPU
Multi-machines	$\times$	$\times$
Tensor Interface	$\surd$	$\surd$
Symbolic Differentiation	$\times$	$\surd$

Table 3.1: Comparison to other popular open source deep learning platforms

dition, MXNet can automatically parallelize code execution, as discussed in Section 3.3.2 below.

Symbolic Interface

The symbolic interface further simplifies the development of deep learning algorithms. One can construct a neural network in a few lines of codes by using the provided layers and operators. MXNet also supports tensor computations on the symbols; for example, assume `out1` is the output of a network, and `out2` is the output from another network of the same size. Then one can define a combined network that sums the results by using `combined = out1 + out2`. MXNet provides automatic differentiation, so it is able to train any constructed symbols. For example, one can create a feed forward network from the symbolic description then fit it to data by using training on a GPU: `mx.fit(mx.FeedForward(combined, mx.gpu()), train_data, ...)`

3.3.2 Engine

All workloads, including both computation and data transfer are pushed into the backend dependency engine for execution. The function *Push* accepts three parameters: the operation, the list of variables this operation will read and the list of variables it will write. For example, to execute $A = B \times C$, one calls

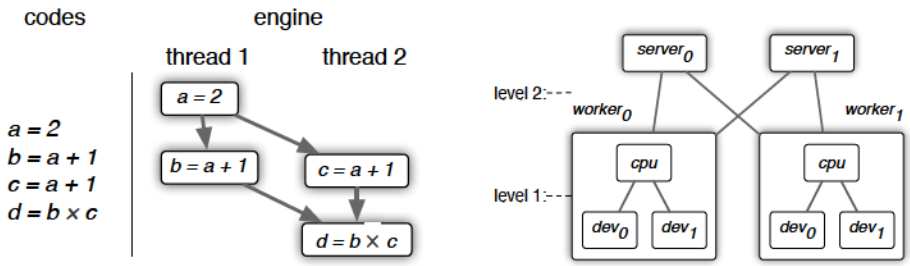

Figure 3.4: Left: The engine constructs a dependency graph and then it executes using multithreads; Right: data synchronization by a two-level parameter server.

```
Push({A=B*C}, {B,C}, {A}).
```

Unlike the predecessor toolkit Minerva [71], MXNet's engine can specify an additional write dependency to ease memory management, and allow special operations such as generating random numbers. Consider an example of initializing two tensors $A = \text{uniform}(-1, 1)$ and $B = \text{uniform}(-1, 1)$. The dependence between these two operations is that they both mutate the random seed. Now one can guarantee the correct execution order by placing a write dependence to the random seed variable.

The function *Push* returns immediately after the operation is pushed into the execution plan. It is the engine's job to guarantee the actual computations are finished when the results are needed. Such an approach to lazy evaluation is transparent to the users, they do not need to aware of it. In addition, it eliminates the dependence to the binding language's performance, which merely issues commands to the engine. This approach further offers the engine more optimization opportunities.

Combined Operators

Multiple operators can be grouped into a single operator to reduce both system overhead and temporal memory allocation. For example, even though the expression $2 \times A + 1$ contains two operations, it can be executed by a single BLAS (Basic Linear Algebra Subprograms) function call without allocating temporary memory for $2 \times A$. Furthermore, MXNet has hand crafted templates for more complex operators, such as a layer in a neural network, to further

improve efficiency.

Parallel Execution

A well-optimized operator is often able to use the full power of a device, e.g. the matrix multiplication in BLAS often achieves 100% CPU utilization. But there usually exists multiple resources such as CPUs, GPUs, and memory/PCIe buses. The engine therefore uses multiple computation threads to execute the pushed operations to ensure better resource utilization. To achieve this, the engine first builds a directed acyclic graph (DAG) using the read and write dependencies. Next, it performs a topological sort then organizes the results by randomly and repeatedly swapping two sequentially unconnected nodes, to avoid creating bad deterministic plans. Then the engine traverses the graph and assigns unconnected nodes to different threads for parallel execution. An example is shown in the left of Figure 3.4.

Squeezing Memory Consumption

Device memory is a major limited resource, particularly for consumer level GPU cards. For example, there is a 3x improvement in GFLOPs (giga floating-point operations per second) from NVIDIA GTX 580 GPU to NVIDIA GTX 980 GPU, while the memory capacity has only been increased from 3GB to 4GB. A naive allocation strategy that allocates memory for every node is not desirable. On the other hand, saving the results to main memory every time can severely harm performance due to limited (PCIe) bus bandwidth. With the dependency graph, we can compute the lifetime of each operation, namely the time period between the creation and the last time being used, and reuse memory for non-intersected nodes. However, an ideal allocation strategy requires $O(n^2)$ time complexity, where n is the number of nodes. MXNet's engine therefore uses two heuristics with linear time complexity to allocate memory. The first heuristic is called *inplace*. It simulates the procedure of traversing the DAG, and keeps the number of dependent nodes that are not used so far. Once a node's number goes 0, the engine recycles its memory. The main disadvantage of this procedure is that it only simulates a single

thread and may add extra dependencies (overlapped memory) between nodes that can be run in parallel. The second heuristic, called *sharing*, solves this problem by allowing two nodes to share memory only if they cannot be run in parallel. In particular, each time it finds the longest path (all nodes there are depended) in the graph that has not been previously assigned, and performs the allocation.

3.3.3 Key-value Store

MXNet uses a key-value (KV) store for synchronizing data across devices. This is used to *push* a KV pair (e.g. gradient g) into the store, to *pull* a value from the store (e.g. weight w), and to set a customized updater to the store for merging the pushed value (e.g. a weight updater $w = w - \eta g$).

The KV store is implemented by a two-level parameter server [42] [41], which is shown on the right of Figure 3.4. A level 1 server manages the data synchronization between the devices on a single machine, while a level 2 server manages multi-machine synchronization.

3.4 Evaluation

I evaluated MXNet on the ILSVRC12 dataset [60], which consists of approximately 1 million images and 1,000 classes. I trained an Inception network [30] with batch normalization [30] (defined in Chapters 4 and 5) using stochastic gradient descent for optimization. I fixed the learning rate, momentum, and weight decay to 0.05, 0.9, 10^{-5}, respectively, and selected the largest batch size that fits into GPU memory. All machines I used were equipped with two or four NVIDIA GTX 980 cards. MXNet is compiled with CUDA 7.5 and CUDNN v3, and used the python binding.

First show the peak memory usage in the left of Figure 3.5. As one can see, the naive allocation is indeed costly. There is a 25% reduction for the *inplace* strategy, while the *sharing* strategy is even better, with a 47% reduction. This reduction is due to the fact that the Inception network has several long parallel paths. Combining both heuristics reduces half of the GPU memory usage.

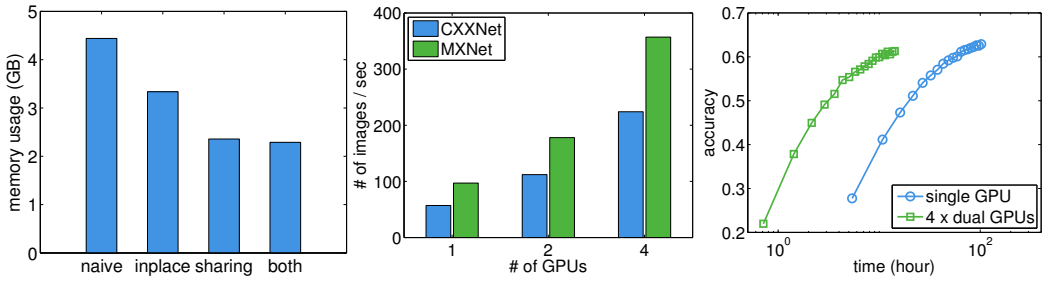

Figure 3.5: Left: peak memory usages for different memory allocation strategies; Middle: performance by varying number of GPUs; Right: validation accuracy versus time when scaling from a single GPU to 4 machines with 8 GPUs in total, each point means one data pass.

Next compared MXNet against its predecessor CXXNet, which only uses a naive engine. The results are shown in middle of Figure 3.5. Both MXNet and CXXNet scale well from 1 GPU to 4 GPUs with 3.9x and 3.7x speedups, respectively. MXNet is always 40% faster than CXXNet however, because MXNet's engine can effectively run multiple operations in parallel, which is an essential advantage for the deep Inception network evaluated.

Finally in right of Figure 3.5 the results for scaling from a single GPU to four machines with 8 GPUs in total. A nearly 7x speedup is achieved thanks to asynchronous communication. However, I also observed a slow down in the rate of convergence, which may due to a larger batch size.

3.5 Conclusion

I have co-developed two deep learning toolkits, CXXNet and MXNet, and evaluated them. The second, more refined toolkit, MXNet, is easy to use, provides both tensor computation and symbolic interfaces, and supports multiple languages. It is efficient in both computation and memory usage, and scales to multiple machines. I have shown promising results on a state-of-the-art large scale deep learning application using a complex deep model. After completing MXNet, the predecessor toolkit CXXNet has been deprecated.

Chapter 4

Structure Design

4.1 Introduction

Designing a good network structure has long been considered a difficult challenge. Although the target task needs to be considered when designing a structure, understanding some general principles is definitely helpful to avoid common errors.

For a convolutional network architecture, there are four important factors to consider: depth (the number of convolutional layers), width (how many filters are in each convolutional layer), filter size, and stride. In the GoogLeNet architecture, there is an additional factor: path. The effect of the path is not well understood, but researchers at Google have invented a complex algorithm to generate such multi-path network topologies to achieve strong performance [68].

If we only consider the traditional "single-path" networks, a simple and naive approach to improving classification accuracy is to make the network deeper and wider. However this is not the best strategy in every scenario. First, a "deeper and wider" network costs much more computation at training and test time. In industrial and commercial scenarios, the training and test times also have to constrained. Also if we want to push the limits of accuracy, current results suggest that it is not necessarily true that deeper and wider networks always generalize better.

In this chapter I will first review the important current network structures that have achieved state of the art results in object recognition, including

LeNet5, AlexNet and VGGNet. Then I will review a preliminary theoretical study of the "depth" of such networks. Next I will introduce an approach for constraining time on a single path network, introducing the "Network in Network" and GoogLeNet's Inception module. This study will lead to the design of a network that requires only 40% of the floating point operations of AlexNet, yet achieves slightly better performance. This new model only requires 7.9MB to store, which is sufficiently small to use on a mobile phone.

4.2 Empirical Structure

4.2.1 LeNet5

LeNet-5 is the first successful attempt to train a "deep" network [38]. Since then, state-of-art network structures have been heavily influenced by LeNet-5. For example, the state-of-art network for MNIST/CIFAR-10/CIFAR-100 in 2013 [20] is similar to the LeNet-5 structure but is deeper and uses a better activation function.

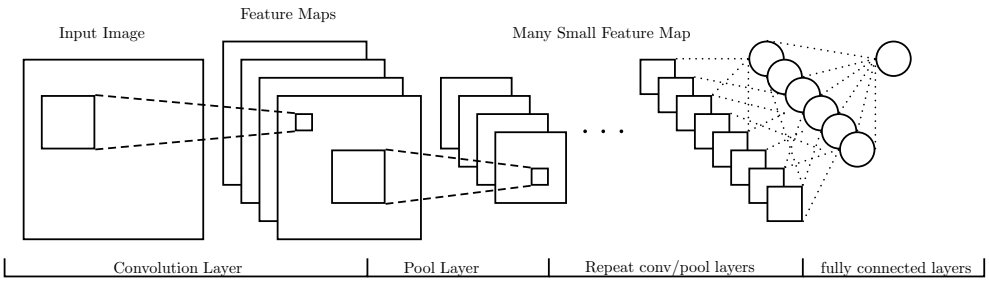

Figure 4.1: Sample LeNet structure

Figure 4.1 shows a sample LeNet structure. In LeNet like structures, each convolutional layer is followed by a pooling layer, then an activation layer. Repeating this structure many times generates many small feature maps. Finally the feature maps are flattened and fed into a traditional multi-layer perception structure. Table 4.1 shows the LeNet-5 structure network structure.

Using the CXXNet toolkit, training LeNet on MNIST takes 28 seconds to achieve 0.77% error on the test-set.

In modern convolutional networks, rather than mapping to a pooling layer directly, the convolutional layers are first stacked. The reasons for the im-

layer	filter size	kernel size	stride
convolution	20	5x5	1
max pooling	-	2x2	2
tanh activation	-	-	-
convolution	50	5x5	1
max pooling	-	2x2	2
tanh activation	-	-	-
fully connected layer	500	-	-
linear classifier			

Table 4.1: LeNet and Maxout MNIST Net

proved performance of recent networks over LeNet is described in the following sections.

4.2.2 AlexNet

AlexNet is short for Alex Krizhevsky's winning network in the ILSVRC-2012 competition [34]. AlexNet has 60 million parameters, consisting of 5 convolutional layers, 3 max pooling layers and 3 fully connected layers. It achieved a 45.7% improvement in top-1 error over the winning ILSVRC-2011 method (using Fisher Vector and SVM) [61].

AlexNet is famous for not only pushing the results on image classification to a new level, but also for the feature representation it produces in the last fully connection layer, which has also boosted performance on other related vision tasks. Transfer learning based retrieval, detection, tracking and image captioning have all been improved simply by taking the AlexNet representation as input.

The AlexNet structure is shown in Table 4.2. Each convolutional layer and fully connected layer is followed by an activation layer. In AlexNet, the activation function used is the rectified linear (ReLU) function instead of tanh. AlexNet also introduced a new channel-wise normalization layer: local response normalization (LRN). The idea of LRN is to normalize adjacent filters in same layer. Formally the normalization $b^i_{x,y}$ to activation $a^i_{x,y}$ in layer i at position (x, y) is defined as

$$b^i_{x,y} = a^i_{x,y} / \left(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a^j_{x,y})^2 \right)^{\beta} \tag{4.1}$$

Due to the limitations of GPUs in 2012, AlexNet introduced a concept of "group" to divide each layer between two cards. As hardware developed in the meantime, we can now fit such networks in a single card so few people use the "group" structure now.

layer	filter size	kernel size	stride
convolution	96	11x11	4
max pooling	-	3x3	2
relu activation	-	-	-
lrn	-	-	-
convolution	256	5x5	1
max pooling	-	3x3	2
relu activation	-	-	-
lrn	-	-	-
convolution	384	3x3	1
relu activation	-	-	-
convolution	384	3x3	1
relu activation	-	-	-
convolution	256	3x3	1
relu activation	-	-	-
fully connected	4096	1x1	1
relu activation	-	-	-
dropout			
fully connected	4096	1x1	1
relu activation	-	-	-
dropout	-	-	-
linear classifier	-	-	-

Table 4.2: AlexNet structure

The contribution of AlexNet includes:

- Using the non-saturating ReLU nonlinearity.
- Using Dropout to prevent over-fitting.
- Stacking convolution layers instead of following max-pooling.

These contributions have influenced all following structure designs. ReLU, Dropout and stacked convolutional layers are basic elements of current convolutional networks in the "deep learning" era. However, local response normalization is not widely used, since it has not proved to be advantageous in further testing [9].

4.2.3 VGGNet

VGGNet is a general name for the network structures produced by the Visual Geometry Group at the University of Oxford. The most famous VGGNet structure is the winner of the ILSVRC-2014 competition, which used 19 layers in a "deep" convolution network. The key finding behind the VGGNet studies is that a 3x3 filter size is the most effective for convolutional layers.

layer	filter size	kernel size	stride
convolution	64	3x3	1
relu activation	-	-	-
convolution	64	3x3	1
relu activation	-	-	-
max pooling	-	2x2	2
convolution	128	3x3	1
relu activation	-	-	-
convolution	128	3x3	1
relu activation	-	-	-
max pooling	-	2x2	2
convolution	256	3x3	1
relu activation	-	-	-
convolution	256	3x3	1
relu activation	-	-	-
convolution	256	3x3	1
relu activation	-	-	-
convolution	256	3x3	1
relu activation	-	-	-
max pooling	-	2x2	2
convolution	512	3x3	1
relu activation	-	-	-
convolution	512	3x3	1
relu activation	-	-	-
convolution	512	3x3	1
relu activation	-	-	-

convolution	512	3x3	1
relu activation	-	-	-
max pooling	-	2x2	2
convolution	512	3x3	1
relu activation	-	-	-
convolution	512	3x3	1
relu activation	-	-	-
convolution	512	3x3	1
relu activation	-	-	-
convolution	512	3x3	1
relu activation	-	-	-
max pooling	-	2x2	2
fully connected	4096	1x1	1
relu activation	-	-	-
dropout			
fully connected	4096	1x1	1
relu activation	-	-	-
dropout	-	-	-
linear classifier	-	-	-

Table 4.3: VGG-E Network

Table 4.3 shows the VGG-E Network structure, which is the deepest VGG Network. Without good initialization methods, such deep networks have to be trained by using pre-training.

However, with careful initialization tricks, one does not need to perform these pre-training steps. We will discuss initialization in later chapter.

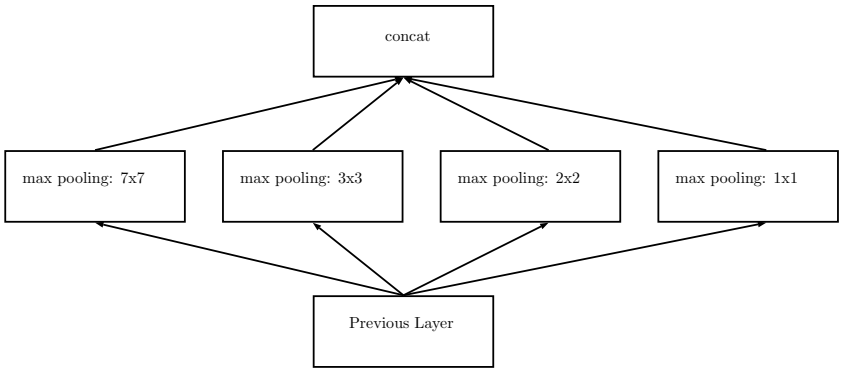

Figure 4.2: Spatial Pyramid Pooling

Based on the VGGNet network structure and better initialization methods,

the researchers as MSR Asia were able to train even deeper and wider networks that included spatial pyramid pooling [23]. With this architecture, they were the first group to surpass human-level performance on ImageNet classification. The detailed changes they developed include:

- For input size 224x224, use a convolutional layer with 96 filters, filter size=7x7, and stride=2 instead of the original two 3x3 convolution layers and one max pooling layer.
- For input size 112x112, only keep one max pooling layer.
- For input size 56x56, add two 3x3 convolution layers, and add 128 more filters for each layer.
- For input size 28x28, add two 3x3 convolution layers, and add 256 more filters for each layer.
- For input size 14x14, add two 3x3 convolution layers, and add 384 more filters for each layer.
- Replace the last max pooling layer to spatial pyramid pooling structure (Figure 4.2).

Currently, except GoogLeNet, all announced networks that surpass human-level performan on the ImageNet task are variants of VGGNet, but with more filters (wider), more convolutional layers (deeper), and a few additional augmentations.

4.3 Maximum Depth Structure

After witnessing so many successes of "deep" networks, a natural question is whether deeper is always better? How deep a network should one use? Unfortunately, there is currently no evidence to support the naive suggestion that "deeper is better". Some specially gated network structures, like the Highway network [66], can make a 100 layer network trainable, but the performance gain is not worth the computing cost.

One of my team-mates in the National Data Science Bowl competition has developed an unpublished preliminary theory of how to design very deep convolution networks [7]. He proposes two constraints for finding the maximum depth of a network, and proves that there is an unique optimal solution under certain conditions. In practice, network structures generated by this theory achieve state-of-art results on CIFAR-10/CIFAR-100. In the National Data Science Bowl competition, this network structure's result is about 4% better than VGGNet. The two novel constraints are quite straightforward:

1. The receptive field size should not be larger than the image size.
2. The value c-value $= \frac{\text{Real filter size}}{\text{Receptive field size}}$ should not be too small.

The first constraint is necessary to permit generalization: if a filter spans the full image, it no longer needs to learn generalizable patterns. Empirically, adding an extra layer that lets the receptive field size exceed the image size increases the risk of over-fitting and hurts performance [7].

The second constraint arises from learning capacity considerations. It based on the observation that if the spatial relationship in the input pattern exceed the filter size, then this convolution will lose its learning capability [7].

Formally, the maximum depth problem is: Given a z by z image, the filter size is set to k by k, and the minimum c-value is set to t. Then the full network is divided in n stages, where for each stage a, there are a_i stacked convolutional layers followed by one max pooling layer. The objective and constraint can be written as follows:

$$\max_{n,\{a_i\}} \sum_{i}^{n} a_i \tag{4.2}$$

$$\text{st.} \quad \sum_{i=1}^{l} a_i 2^{i-1} \leq \frac{2^l k}{t(k-1)} \text{ where } l = 1, 2, ..., n \tag{4.3}$$

$$\sum_{i} a_i 2^{i-1} \leq \frac{z}{k-1} \tag{4.4}$$

The optimal solution given by the report [7] is then determined as follows. Assume the image size $z = 2^{m-1}k/t$, and relax $\{a_i\}$ from positive integers to positive real number, the optimal solution is:

$$n = m \tag{4.5}$$

$$a_1 = \frac{k}{(k-1)t}, a_2 = \cdots = a_n = \frac{1}{2a_1} \tag{4.6}$$

This is the first public algorithm that indicates a principled approach for designing deep architectures for image processing tasks. Again, this result is preliminary and there should remain opportunity to make further improvements.

4.4 Network in Network and Inception

The Network in Network [43] is a new model design compared to the previous models. It brings two innovations:

1. Using a multilayer perception convolution.

2. Using global average pooling.

The multi layer perception convolution uses multiple 1x1 convolutions to increase the depth of the network and increase the learning capacity of a single convolution layer. The consequence of more learning capacity is the potential to over-fit, so the multi layer perception convolution layer is often regularized by Dropout.

The use of global average pooling contributes a significant change over the previous models: Instead of reducing the feature map to a small size then flattening it to generate a feature vector, global average pooling runs average pooling on a large feature map to generate a final low dimensional feature vector. Unlike other network structures with a multi layer perception on the feature vector, this approach only needs to use a linear classifier over a global average pooling feature vector.

The Inception module was first proposed as part of the GoogLeNet model [68], which was the winner of the ILSVRC-14 competition. Unlike the previous structures, GoogLeNet uses Inception modules—a multiple path convolution

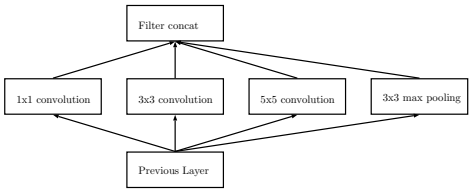

Figure 4.3: Naive Inception module

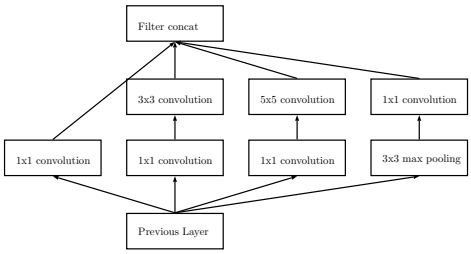

Figure 4.4: Inception with dimension reduction

sub-network—instead of a single convolution layer. Figure 4.3 shows the basic Inception module.

GoogLeNet and the Inception module are heavily inspired by the Network in Network model outlined above [43]. The "reduce" layer in the Inception module is exactly an instance of a Network in Network. Overall, the GoogLeNet model consists of stacking Inception modules, then using global average pooling to generate a feature map, plus one or more linear classifiers. GoogLeNet reduces the number of parameters by 90% from AlexNet.

Using multiple linear classifiers at different depths in a network is also called "Deeply supervised network" [39].

It is believed that the GoogLeNet structure is generated by an algorithm internal to Google rather than handcrafted [68]. I have observed some changes between the 2014 and 2015 GoogLeNet (Table 4.4), the latter of which surpasses human level performance in ImageNet classification:

1. A batch normalization (defined in Chapter 5) layer is used after every convolution layer.

2. The 5x5 convolution is abandoned. Instead, a dual 3x3 convolution is used in each Inception module.

3. Average pooling is introduced into the Inception module.

Overall, the Inception Network achieves excellent performance using a limited number of parameters. In practice, training of the Inception network is able to be parallelized, so by using multiple GPUs, one is able to train a state-of-art model quickly.

type	kernel/ stride	#1x1	#3x3 reduce	#3x3	double #3x3 reduce	double #3x3	pool + proj
convolution	7x7/2			64			
max pool	3x3/2						
convolution	3x3/1		64	192			
max pool	3x3/2						
Inception (3a)		64	64	64	64	96	avg + 32
Inception (3b)		64	64	96	64	96	avg + 64
Inception (3c)	stride 2	0	128	160	64	96	max
Inception (4a)		224	64	96	96	128	avg + 128
Inception (4b)		192	96	128	96	128	avg + 128
Inception (4c)		160	128	160	128	160	avg + 128
Inception (4d)		96	128	192	160	192	avg + 128
Inception (4e)	stride 2	0	128	192	192	256	max
Inception (5a)		352	192	320	160	224	avg + 128
Inception (5b)		352	192	320	192	224	max + 128
avg pool	7x7/1						

Table 4.4: Inception Network Structure Surpass Human Vision

4.5 Constrained Time Structure

In the AlexNet paper [34], the authors write that "All of our experiments suggest that our results can be improved simply by waiting for faster GPUs and bigger datasets to become available." Over the next 3 years, given the emergence of faster GPUs, great performance improvements have indeed been observed by stacking more layers based on AlexNet. However these deeper and wider models are more time-consuming to train than AlexNet. In industrial applications, both classification performance and time are important considerations. The Microsoft Research Asia Vision group has investigated the trade off [22], by considering possible layer replacements. Formally, we can estimate complexity of a convolutional network in big-O notation as:

$$O(\sum_{l=1}^{L} n_{l-1} \cdot s_l^2 \cdot n_l \cdot m_l^2) \tag{4.7}$$

where L is total number of layers, n_l is number of filters in layer l, s_l is filter size and m_l is output feature map size.

The MSR Asia group investigated the influence on performance of several

factors, including filter number (width), layer number (depth) and filter size. By replacing a layer type without changing the big-O complexity, they have achieved some insight into the design trade offs for a single path convolutional network:

1. Depth is important. To constrain time, one can replace a 5x5 convolution layer with two 3x3 convolution layers, or replace one 3x3 convolution layer with two 2x2 convolution layers.

2. The number of filters can also be reduced to constrain time. With the same complexity, a 3x3 layer with fewer filters performs better than a 2x2 layer with more filters.

3. The pooling stride should be set to 1, while making the convolution stride greater than 1 will enable the use of more filters.

4. Delayed pooling improves performance.

Following these practical observations, I modified the AlexNet (Table 4.2) into one 7x7 convolutional layer and 10 2x2 convolutional layers without increasing complexity. However these new models require much more memory due to the increase in depth, while also using more parameters than AlexNet. These considerations make it difficult to accelerate training for such a model.

4.6 Tiny ImageNet Network

Although convolutional neural networks have become a core technology for modern computer vision tasks, if one wants to run such networks on robots or mobile phone, we have to carefully constrain memory, parameter count, and computation cost. A goal would be to have a model with similar or better performance than AlexNet, but with better memory cost, less computation and less parameters. Currently none of the previous models are able to satisfy these requirements:

1. The VGGNet has nearly the best performance, but the resource requirements are extensive and desktop computers are not able to handle it.

2. The original constrained time model has better performance than the baseline AlexNet, but it costs too much memory and the model size is too large.

3. GoogLeNet/Network in Network limits the parameter number, but the computation cost remains too high.

From these observations, GoogLeNet provides the best starting point because we then only need to constrain computation cost. However the performance trade offs of Inception module in GoogLeNet are still not well understood.

We want to know how different parts of the Inception module influence the resulting classification performance. To investigate this issue, I broke the Inception module into several components, forming variants of the basic module, then replace all Inception modules in the basic network by using new variants. To train these models, I used the same hyper parameters for the ILSVRC classification task but with fewer learning rate schedules.

Details of these new Inception modules are given in Figure 4.5:

- Module-A is the baseline. It is similar to VGG-16 but with fewer layers. The entire network is built by single path 3x3 convolution layers.

- Module-B adds a dual 3x3 path to Module-A.

- Module-F uses a single path 3x3 convolution layer but the filter number is the sum of Module-B's two convolution layers' filter.

- Module-C changes one path in Module-B to a 5x5 convolution layer.

- Module-E changes the 5x5 path in Module-C to a 1x1 convolution.

- Module-D adds a 1x1 convolution dimension reduction layer (Network in Network) to Module-A.

- Module-J adds an extra Network in Network path to Module-D.

- Module-G adds an averaging pooling path to Module-A.

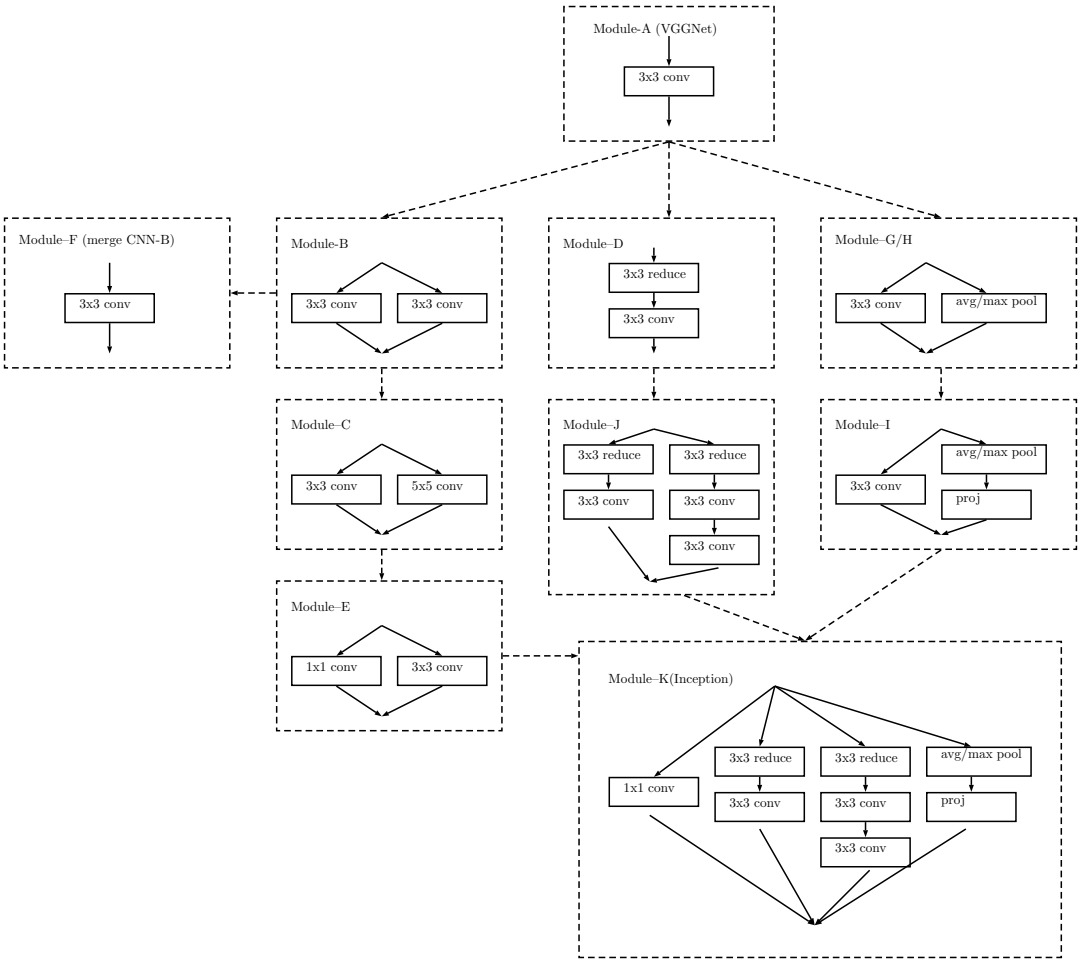

Figure 4.5: Experimental Inception module combination

- Module-H adds a max pooling path to Module-A.
- Module-I adds a projection layer on top of the pooling layer.
- Module-K merges Module-E's 1x1 path, Module-J's dual 3x3 Network in Network path, and Module-I's pooling/projection layer.

The experimental results are shown in Table 4.5. I have some observations based on these experiments:

1. Module-K (Inception module)'s performance is best overall.
2. Replacing one of the double 3x3 convolution paths to a 5x5 convolution path improves classification performance a little, but almost doubles the

Module Type	Train Error	Test Error	Time Cost
Module-A	8.216%	12.50%	1x
Module-B	6.67%	11.54%	4.94x
Module-C	6.29%	11.24%	8.42x
Module-D	12.08%	15.77%	1.48x
Module-E	6.88%	10.90%	2.26x
Module-F	6.93%	13.34%	2.64x
Module-G	83.68%	83.14%	3.64x
Module-H	86.69%	85.14%	3.64x
Module-I	7.19%	11.75%	2.45x
Module-J	9.71%	14.16%	3.38x
Module-K	5.45%	10.20%	7.35x

Table 4.5: CIFAR-10 Result on each combination module

computation cost. I think this is the reason that [30] switches to a double 3x3 path from original combination of a 3x3 path and a 5x5 path.

3. Adding a 1x1 convolution projection to the pooling path is important for allowing successful training.

4. Although the 1x1 convolution dimension reduction layer doubles the depth of module, just adding dimensionality reduction does not improve performance.

5. The results for Module-E indicate that an extra 1x1 path plays an important role in improving classification performance. Meanwhile, compared to other changes, the computational cost for this change is reasonable.

Now, I can take things a step further based on the work of [22]. Above I assessed the independent contributions to performance of each path in an Inception module. I found that a 1x1 convolution path provided the most obvious performance gains within reasonable additional time costs. In an opposite consideration, if time complexity allows, I am able to add a 1x1 convolution to the network to improve performance. In light of these considerations, I designed a tiny network structure, shown in Table 4.6.

This tiny network requires only 40% of the floating point operations used by AlexNet for a forward pass. A detailed comparison is given in Table 4.7.

type	kernel/stride	output	1x1	3x3	pool
convolution	7x7/2(A), 9x9/4(B)	56x56x64			
max pool	3x3/2	28x28x64			
convolution	3x3/1	28x28x96			
module-1		28x28x64	32	32	
module-2		28x28x80	32	48	
module-3	stride 2	14x14x160		80	max
module-4		14x14x160	112	48	
module-5		14x14x160	96	64	
module-6		14x14x160	80	80	
module-7		14x14x144	48	96	
module-8	stride 2	7x7x240		96	max
module-9		7x7x336	176	160	
module-10		7x7x336	176	160	
avg pool	7x7/1	1x1x336			

Table 4.6: Tiny ImageNet structure

The memory cost is calculated assuming a batch size of 128.

Model	Parameter	Model Size	Memory	Test Error	Complexity (Equation 4.7)
AlexNet	60M	212 MB	1X	19.80%	1X
VGG-16	138M	528 MB	3.5X	10.69%	22.6X
GoogLeNet	5M	45.3 MB	3.4X	10.01%	3.07X
Kaiming Net	95M	303 MB	2.2X	15.70%	1.28X
TinyNet-A	1.54M	7.9 MB	1.2X	21.00%	0.40X
TinyNet-B	1.56M	7.9 MB	1.2X	17.00%	0.42X

Table 4.7: Network comparison

Finally I used TinyNet-B as the final design for TinyNet. I designed this model based on the following observations:

1. Depth contributes to memory cost directly.

   If one doubles the depth of the network, in the current design, the memory cost will be doubled. Therefore, it is not wise to make the network very deep. If one wants to make the network deeper without adding extra complexity, each 3x3 convolution layer can be changed to two 2x2 convolution layers.

2. Use an extra 1x1 path.

I introduced an extra 1x1 path to improve classification performance—a lesson learnt from the investigation of the Inception module. If using the filter setting from GoogLeNet directly, the complexity will be 1.3 times that of AlexNet. Instead, I reduced nearly half of the original filters from AlexNet.

3. A higher resolution input leads to better results.
   Compared to TinyNet-A, which uses a 128x128 input image size, TinyNet uses an input image size of 256x256 and performs 9.5% better with nearly no additional cost. The only difference between TinyNet-A and TinyNet occurs in the first layer kernel size and stride.

I have successfully deployed TinyNet using the CXXNet toolkit on an Android smartphone. Some image labeling tests on random real world images are shown in Figure 4.6.

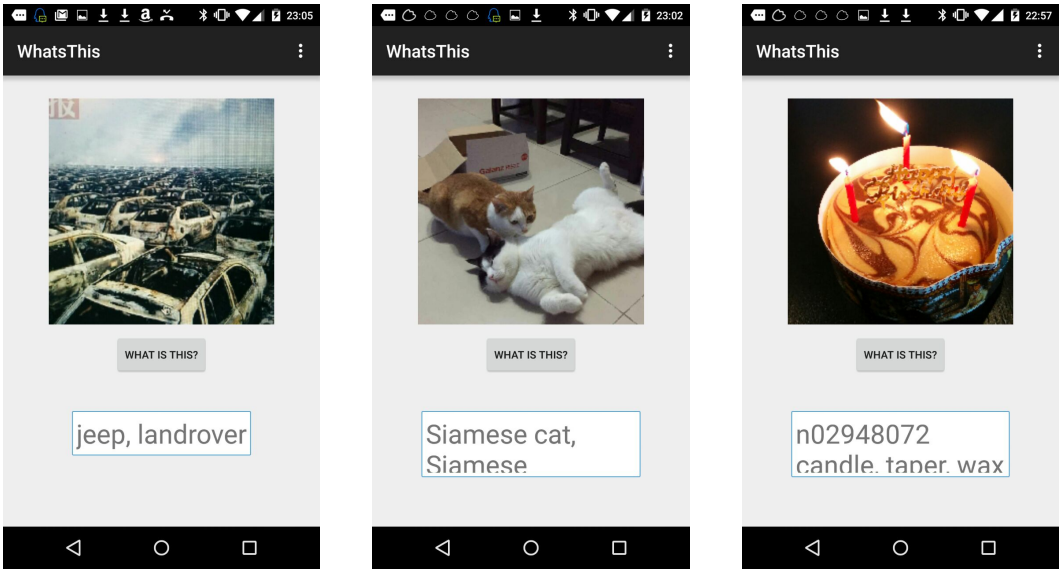

Figure 4.6: TinyNet on Android Phone

4.7 Conclusion

In this chapter, I investigated the structure of convolutional networks for image classification. By reviewing current work on network structure and the recent

evolution of network design, I uncovered some general principles for how to design a suitable convolution network. Finally, I designed an ImageNet network, called TinyNet, that is much more efficient than AlexNet while still achieving improved classification accuracy.

In next chapter, we will discuss how activation function influence classification performance.

Chapter 5

Activation Functions

5.1 Introduction

In addition to depth, one of the key characteristics of modern deep learning system is the use of non-saturated activation functions (e.g. ReLU) to replace their saturated predecessors (e.g. sigmoid, tanh). Saturated means the activation function is bounded and with 0 gradient at the bound. The advantage of using non-saturated activation function lies in two aspects: The first is to solve the so called "exploding/vanishing gradient problem" [27]. The second is to accelerate the convergence speed.

Among the non-saturated activation functions, the most notable is the *rectified linear unit* (ReLU) [50], shown in Figure 5.3. Briefly, it is a piecewise linear function that clips the identity function, pruning the response on the negative domain to zero while retaining the identity on the positive domain. It has the desirable property that its activations are sparse: it is commonly believed that the superior performance of ReLU comes from the sparsity [19]. However, there remain two important questions regarding the ReLU activation: *First, is sparsity the most important factor for a good performance? Second, can we design a better non-saturated activation that can exhibit superior performance to ReLU?*

In this chapter, I will review the classical saturated activation functions and the recent rectified linear activation function. Then, inspired by Dropout, I will introduce randomness into the activation function to define a modified form of ReLU activation. I also propose the Sigmoid* function, which avoids vanishing

problem of the original sigmoid function. My findings suggest that, despite being the most popular activation function, ReLU is not the end of story: three types of (modified) leaky ReLU consistently outperform the original ReLU. However, the reason for their superior performance still lack rigorous justification from a theoretic standpoint.

5.2 Saturated Activations

For models of biological neurons, activation corresponds to the expected firing rate as a function of the total input currently arising from incoming signals via synapses [16]. By observing neuronal activation in animals, the sigmoid function is often selected as an activation function, to squash the output signal into $(0, 1)$ [35]. The hyperbolic tangent is also often used as an activation function, since it squashes the input into the range $(-1, 1)$. Figure 5.1, Figure 5.2 show the sigmoid and hyperbolic tangent functions and their gradients.

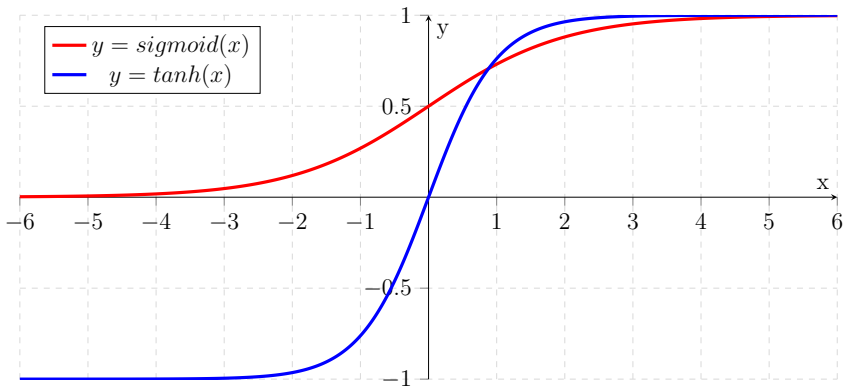

Figure 5.1: Sigmoid Activation and Tanh Activation

For a long time it is believed both the sigmoid and hyperbolic tangent functions have the problem of "exploding/vanishing gradients" when back-propagating through a deep network.

It is easy to show that the maximum value of the sigmoid gradient is 0.25, which means that if weight $\|w\| = 1$, the lower gradients in the network will decrease in factor $\frac{1}{4}$ after each sigmoid layer. The hyperbolic tangent still has the same problem of gradient vanishing because it reaches its maximum value of 1 when x is 0. Otherwise, if there is no constraint on the weight normal,

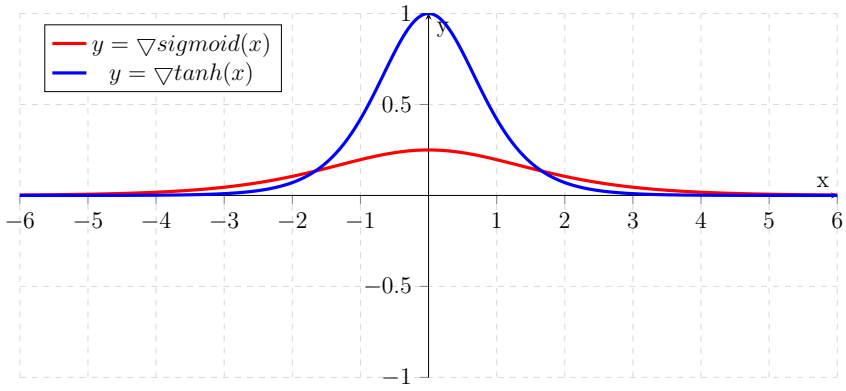

Figure 5.2: Sigmoid gradient and Tanh gradient

the weights can become very large, and backpropagation leads to gradient explosion. Also, each layer's input changes during training, which makes the problem even harder. This phenomenon is also referred as *internal covariate shift* [30].

Rectifiers address these problems, but the use of rectification makes the neural network less biological. Batch normalization is also used to address these problems from another perspective.

5.3 A Generalized Family of Rectified Activation Functions

I have investigated a broader class of activation functions, namely the rectified unit family. In particular, I have investigated the leaky ReLU and its variants. In contrast to ReLU, which completely drops the negative part, the leaky ReLU assigns a non-zero slope to this region. The first variant is called *parametric rectified linear unit* (PReLU) [24]. In PReLU, the slopes of the negative part are learned form data rather than predefined. The authors claimed that PReLU is the key factor of surpassing human-level performance on ImageNet classification [60]. The second variant, which was introduced and investigated as part of this thesis research, is called the *randomized rectified linear unit* (RReLU). In RReLU, the slopes of negative parts are randomized in a given range during training, then fixed during testing. On small scale data, like NDSB, I have found that the RReLU activation can reduce over-fitting due to

its randomized nature. These activation functions in the rectified unit family are shown in Figure 5.3.

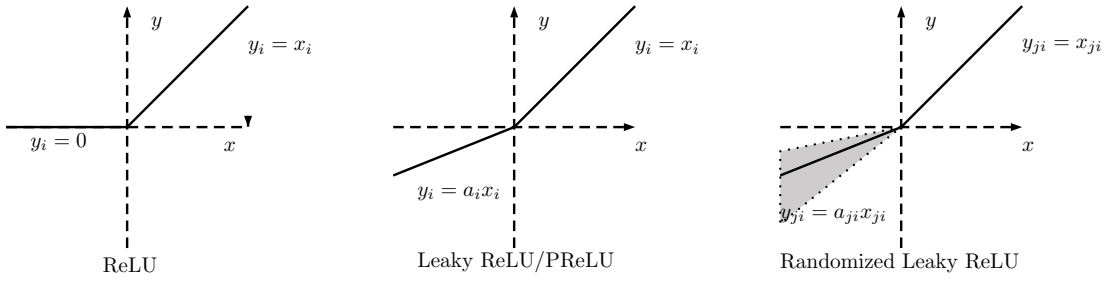

Figure 5.3: ReLU, Leaky ReLU and RReLU

5.3.1 ReLU

The rectified Linear activation (ReLU) was first used in Restricted Boltzmann Machines [50]. Formally, the rectified linear activation is defined as:

$$y_i = \begin{cases} x_i & \text{if } x_i \geq 0 \\ 0 & \text{if } x_i < 0. \end{cases} \tag{5.1}$$

5.3.2 Leaky ReLU

The leaky rectified linear activation (Leaky ReLU) was first introduced in the acoustic model proposed by [46]. Mathematically, it is defined as

$$y_i = \begin{cases} x_i & \text{if } x_i \geq 0 \\ \frac{x_i}{a_i} & \text{if } x_i < 0, \end{cases} \tag{5.2}$$

where a_i is a fixed parameter in range $(1, +\infty)$. In the original paper, the authors suggest to set a_i to a large number like 100. Beyond this setting, I have also experimented with smaller $a_i = 5.5$ in my investigation.

5.3.3 Parametric ReLU

The parametric rectified linear (PReLU) was originally proposed by [24]. The authors reported that the PReLU activation led to superior performance than the ReLU activation in a large scale image classification task. The PReLU activation is the same as the leaky ReLU (Equation 5.2) with the exception

that a_i is learned during training via back propagation. Formally, the Leaky ReLU Activation (Equation 5.2) can be rewritten as:

$$y_i = \begin{cases} x_i & \text{if } x_i \geq 0 \\ a_i x_i & \text{if } x_i < 0, \end{cases} \tag{5.3}$$

During back propagation, we need to optimize the parameters a_i simultaneously with the other weights in the network. The gradient of a_i for one layer is:

$$\frac{\partial \varepsilon}{\partial a_i} = \sum_{y_i} \frac{\partial \varepsilon}{\partial y_i} \frac{\partial y_i}{\partial a_i} \tag{5.4}$$

where ε represents the object function, and the term $\frac{\partial \varepsilon}{\partial y_i}$ is the gradient propagated from the higher layer. The gradient of the activation is:

$$\frac{\partial y_i}{\partial a_i} = \begin{cases} 0 & \text{if } y_i \geq 0 \\ x_i & \text{if } y_i < 0. \end{cases} \tag{5.5}$$

5.3.4 Randomized Leaky ReLU

The randomized leaky rectified linear activation (RReLU) is the randomized version of the leaky ReLU. I first proposed and investigated this activation function during the Kaggle NDSB Competition with my teammates. The main feature of the RReLU activation is that, during the training process, the a_{ji} value is a random number sampled from a uniform distribution $U(l, u)$.

The randomized leaky ReLU is inspired by Dropout. For the negative slope, I use a random slope during training, instead of training or setting it arbitrarily. During testing, this activation is similar to averaging models; the resulting procedure resembles an ensemble approach that can hopefully improve performance. Formally, the RReLU is given by

$$y_{ji} = \begin{cases} x_{ji} & \text{if } x_{ji} \geq 0 \\ x_{ji} a_{ji} & \text{if } x_{ji} < 0, \end{cases} \tag{5.6}$$

where

$$a_{ji} \sim U(l, u), \text{ st. } u < 1, l < u \tag{5.7}$$

In the test phase, we take the average of the a_{ji} values in training, as in Dropout [65] , and thus set a_{ji} to $\frac{l+u}{2}$ to achieve a deterministic result.

5.4 Batch Normalization

Batch Normalization [30] has recently been proposed for accelerating GoogLeNet training. Strictly speaking, batch normalization is not an activation function. By normalizing and shifting each mini-batch after every convolution or fully connection layer, an original deep network with saturated activations becomes trainable. This normalization also accelerates training of networks with non-saturated activation functions. During the normalization and shift processes, batch normalization is also attempting to learn some parameters. Formally, for a input batch with m example, $B = \{x_1, ..., x_m\}$, the output of batch normalization, y_i, for example x_i in the batch is calculated in following steps:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i \tag{5.8}$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2 \tag{5.9}$$

$$\hat{x_i} = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \varepsilon}} \tag{5.10}$$

$$y_i = \gamma\hat{x_i} + \beta \tag{5.11}$$

where μ_B and σ_B^2 is mean and variance of current batch, $\hat{x_i}$ is normalized result, γ and β are shift parameter need to be learned. The gradient is calculated as follows:

$$\frac{\partial\varepsilon}{\partial\hat{x_i}} = \frac{\partial\varepsilon}{\partial y_i}\cdot\gamma \tag{5.12}$$

$$\frac{\partial\varepsilon}{\partial\sigma_B^2} = \sum_{i=1}^{m}\frac{\partial\varepsilon}{\partial\hat{x_i}}\cdot(x_i - \mu_B)\cdot\frac{-1}{2}(\sigma_B^2 + \epsilon)^{-3/2} \tag{5.13}$$

$$\frac{\partial\varepsilon}{\partial\mu_B} = \left(\sum_{i=1}^{m}\frac{\partial\varepsilon}{\partial\hat{x_i}}\cdot\frac{-1}{\sqrt{\sigma_B^2 + \epsilon}}\right) + \frac{\partial\varepsilon}{\partial\sigma_B^2}\cdot\frac{\sum_{i=1}^{m} -2(x_i - \mu_B)}{m} \tag{5.14}$$

$$\frac{\partial\varepsilon}{\partial x_i} = \frac{\partial\varepsilon}{\partial\hat{x_i}}\cdot\frac{-1}{\sqrt{\sigma_B^2 + \epsilon}} + \frac{\partial\varepsilon}{\partial\sigma_B^2}\cdot\frac{2(x_i - \mu_B)}{m} + \frac{\partial\varepsilon}{\partial\mu_B}\cdot\frac{1}{m} \tag{5.15}$$

$$\frac{\partial\varepsilon}{\partial\gamma} = \sum_{i=1}^{m}\frac{\partial\varepsilon}{\partial y_i}\cdot\hat{x_i} \tag{5.16}$$

$$\frac{\partial\varepsilon}{\partial\beta} = \sum_{i=1}^{m}\frac{\partial\varepsilon}{\partial y_i} \tag{5.17}$$

Batch normalization is designed to fix the internal covariate shift phenomenon. It makes a deep network with saturated activation functions trainable. Also, it accelerates the training of deep networks with ReLU activation functions.

5.5 Explaining the Difficulty of Training with Sigmoids

Modern neural network initializations are based on controlling the output/gradient variance, to prevent the exponential vanishing/explosion that hampers subsequent training.

5.5.1 Random Initialization Methods

Formally, for a single layer in neural network with activation function f, in a forward pass we have

$$x_l = f(y_{l-1}) \tag{5.18}$$

$$y_l = W_l x_l + b_l \tag{5.19}$$

where y_{l-1} is the previous layer's output, x_l is the current layer l's input, W_l is the weight matrix in layer l, and b_l is the bias respectively.

Consider the hypothesis that all weights are initialized mutually independently, share the same distribution, and are zero mean. Also assume there are n_l connections in layer l. Then in a forward pass we have:

$$Var[y_l] = n_l Var[w_l] Var[x_l] \tag{5.20}$$

and in a backward pass we have:

$$Var\left[\frac{\partial \varepsilon}{\partial y_l}\right] = n_l \frac{\partial x_{l+1}}{\partial y_l} Var[w_l] Var\left[\frac{\partial \varepsilon}{\partial x_{l+1}}\right] \tag{5.21}$$

while

$$\frac{\partial x_{l+1}}{\partial y_l} = f'(y_l) \tag{5.22}$$

Xavier's Initialization

Xavier's initialization [18] is based on two hypotheses:

1. A dense neural network with symmetric activation f will have $f'(y_l) \approx 1$ when $y_l = 0$

2. The variance is constant in both the forward and backward directions; that is:

$$Var[y_l] = Var[y_{l-1}] \tag{5.23}$$

$$Var\left[\frac{\partial \varepsilon}{\partial y_l}\right] = Var\left[\frac{\partial \varepsilon}{\partial y_{l-1}}\right] \tag{5.24}$$

Considering these two hypotheses, we have

$$n_l Var[w_l] = 1 \tag{5.25}$$

$$n_{l+1} Var[w_l] = 1 \tag{5.26}$$

Therefore, an approximate solution for $Var[w_l]$ is

$$Var[w_l] = \frac{2}{n_l + n_{l+1}} \tag{5.27}$$

Kaiming's Initialization

Kaiming's initialization [24] is specialized for ReLU under the hypotheses:

1. The mean of x_l is zero, which, with Equ. 5.20, implies

$$Var[y_l] = n_l Var[w_l] E[x_l^2] \tag{5.28}$$

2. If the activation f is ReLU, w_{l-1} has a symmetric distribution around 0, and $b_l = 0$, we will have:

$$E[x_l^2] = \frac{1}{2} Var[y_{l-1}] \tag{5.29}$$

$$Var[y_l] = \frac{1}{2} n_l Var[w_l] Var[y_{l-1}] \tag{5.30}$$

For a network with L layers, the variance of the output will then be

$$Var[y_L] = Var[x]\left(\prod_{l=1}^{L}\frac{1}{2}n_l Var[w_l]\right) \tag{5.31}$$

Similarly, for back propagation, with the previous hypotheses, the variance of the gradient is given by:

$$Var\left[\frac{\partial \varepsilon}{\partial x_1}\right] = \frac{\partial \varepsilon}{\partial x_L}\left(\prod_{l=1}^{L}\frac{1}{2}\hat{n}_l Var[w_l]\right) \tag{5.32}$$

where $\frac{1}{2}\hat{n}_l Var[w_l] = 1$ is sufficient to prevent exponential vanishing and exploding.

5.5.2 The Reason for Sigmoid's Failure to Converge

Assume the network inputs, x, have mean 0. At 0, the Taylor expansion of the Sigmoid function is:

$$f(x) = \frac{1}{2} + \frac{x}{4} + \frac{x^2}{48} + \dots \tag{5.33}$$

To simplify the problem, approximate the sigmoid function at 0 with $f(x) = \frac{1}{2} + \frac{x}{4}$. Then we will have

$$f'(0) = 0.25 \tag{5.34}$$

$$Var[x_l] = \frac{1}{16}Var[y_{l-1}] \tag{5.35}$$

This violates the requirements of the previous initialization methods:

1. Xavier's initialization assumes $f'(y_l) \approx 1$ when $y_l = 0$.
2. Kaiming's initialization assumes $E[x_l^2] = \frac{1}{2}Var[y_{l-1}]$.

To fix these problems, we can modify the original sigmoid activation to

$$Sigmoid^*(x) = 4Sigmoid(x) - 2 \tag{5.36}$$

Figure 5.4 and Figure 5.5 show the difference between the Sigmoid* and Tanh functions.

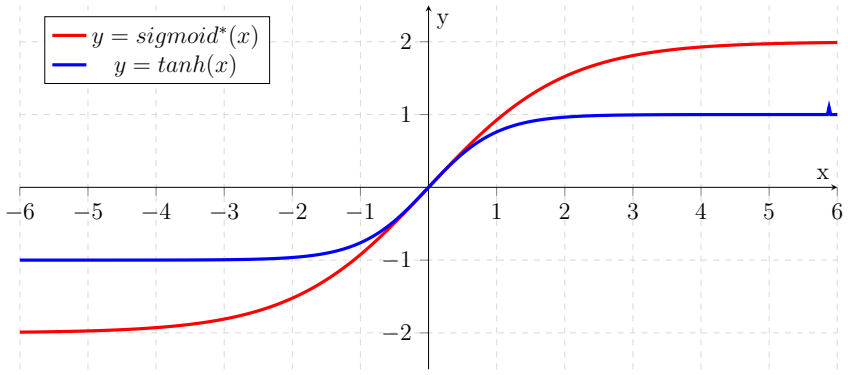

Figure 5.4: Sigmoid* Activation and Tanh Activation

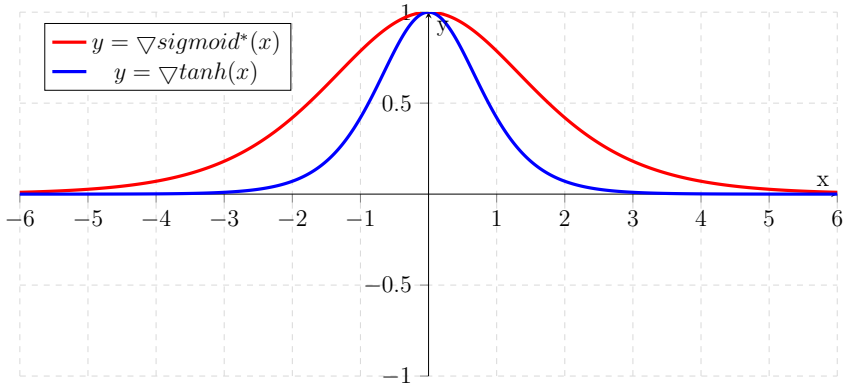

Figure 5.5: Sigmoid* gradient and Tanh gradient

5.6 Experiment on CIFAR-10

I have evaluated these activation functions by using a subnet from the TinyNet model developed above (Table 4.6); the network structure is shown in Table 5.1. I reduce learning rate by factor $\frac{1}{10}$ at epoch 20. Here I removed the first convolution and pooling layers to make the network accept an input of 28x28 pixels.

The classification results using different activations is given in Table 5.2. Here I considered activation with and without batch normalization. From the table one can make the following observations:

1. RReLU is best both with batch normalization and without batch normalization.

2. Saturated activation functions are worse than any member of the rectified family.

type	kernel/stride	output	1x1	3x3	pool
convolution		28x28x64		96	
module-1		28x28x64	32	32	
module-2		28x28x80	32	48	
module-3	stride 2	14x14x160		80	max
module-4		14x14x160	112	48	
module-5		14x14x160	96	64	
module-6		14x14x160	80	80	
module-7		14x14x144	48	96	
module-8	stride 2	7x7x240		96	max
module-9		7x7x336	176	160	
module-10		7x7x336	176	160	
avg pool	7x7/1	1x1x336			

Table 5.1: Tiny CIFAR-10 structure

3. Batch normalization consistently improves performance

4. Batch normalization cannot be used as an activation function on its own.

Activation	Train-Error	Test-Error
sigmoid	90.0%	90.0%
tanh	13.17%	19.84%
relu	2.89%	11.89%
prelu	1.25%	9.70%
leaky relu (a = 7.5)	0.94%	9.26%
rrelu	2.01%	**9.06%**
sigmoid + batch norm	8.33%	12.37%
tanh + batch norm	1.58%	9.71%
relu + batch norm	0.22%	8.30%
leaky relu (a=7.5) + batch norm	0.72%	8.44%
prelu + batch norm	0.45%	8.25%
rrelu + batch norm	0.63%	**7.98%**
batch norm	13.87%	17.24%

Table 5.2: Activation function and result

From the Tanh learning curve (Figure 5.8), ReLU learning curve (Figure 5.6) and RReLU learning curve (Figure 5.7), we can see that batch normalization does improve performance and makes convergence faster.

Comparing the ReLU and RReLU learning curves in Figure 5.9, one can observe that RReLU converges faster. However, with batch normalization

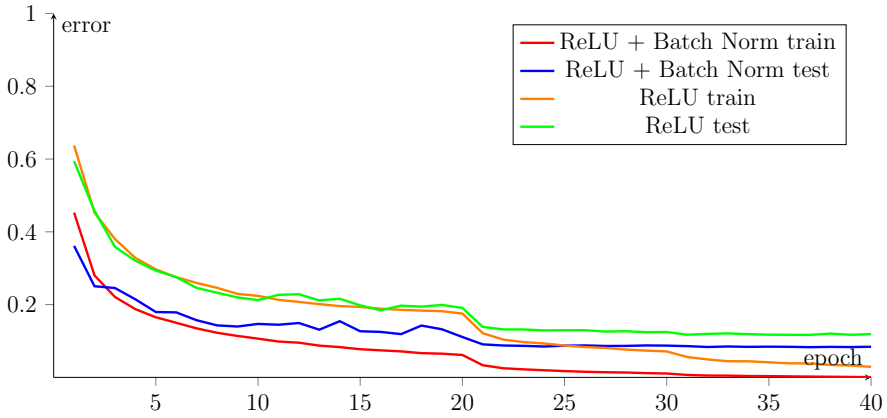

Figure 5.6: ReLU and ReLU with Batch Normalization learning curve

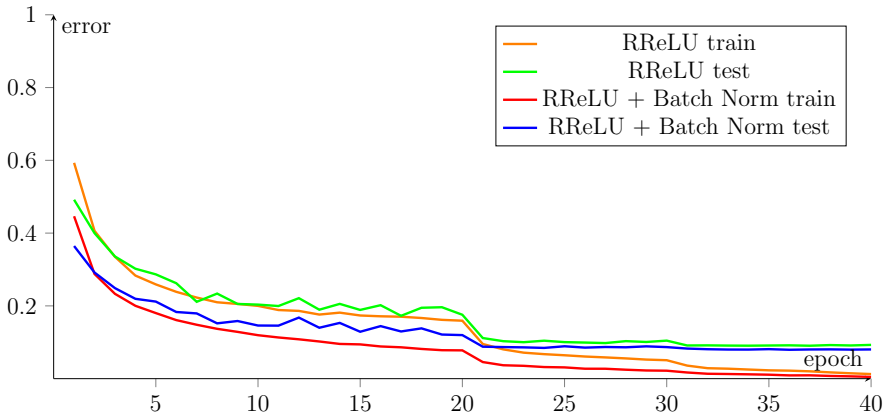

Figure 5.7: RReLU and RReLU with Batch Normalization learning curve

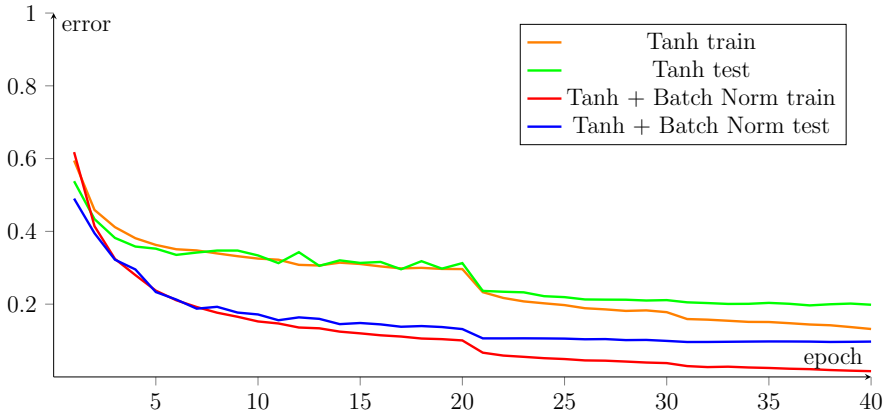

Figure 5.8: Tanh and Tanh with Batch Normalization learning curve

(Figure 5.10) the difference is reduced.

Also I find that batch normalization cannot be used as a non-linear activation function on its own.

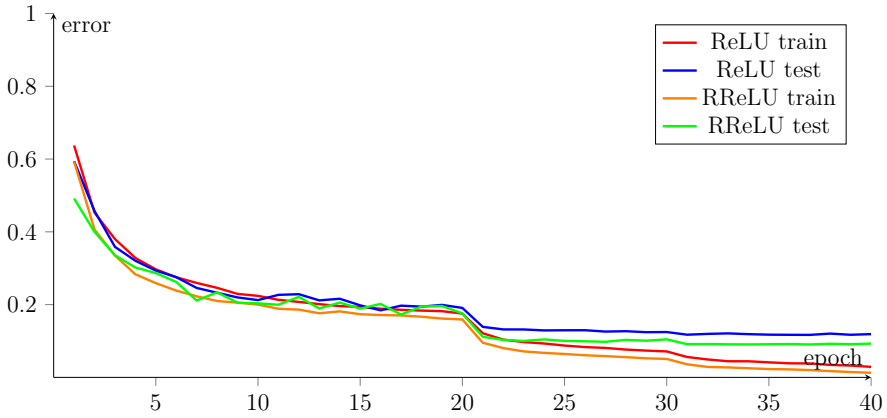

Figure 5.9: ReLU and RReLU learning curve

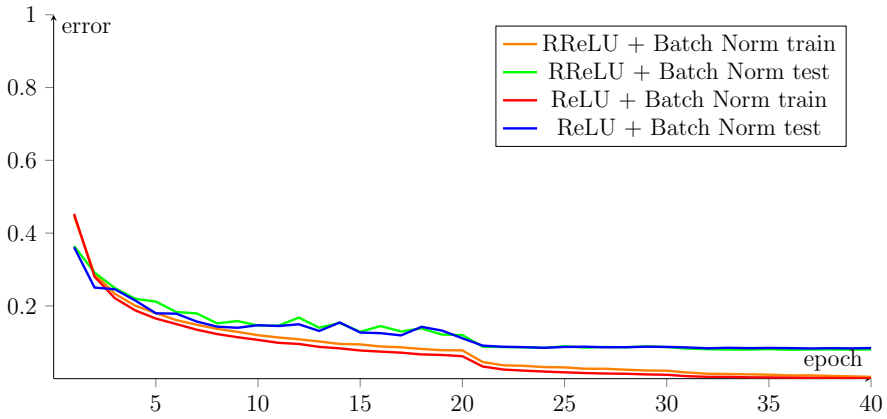

Figure 5.10: ReLU + Batch Norm and RReLU + Batch Norm learning curve

5.7 Experiment on CIFAR-100

In order to address vanishing/explosion problem, in this experiment I choose the 33 layer Inception-BN [30] network, but removed Batch Normalization. I manually reduce learning rate by factor $\frac{1}{10}$ at epoch 40. I choose the CIFAR-100 dataset, since this problem is more challenging than CIFAR-10. Here the networks were initialized by using Xavier's initialization [18]. I also remove dropout in these experiments. The numerical result is shown in Table 5.3

There is no doubt that the Sigmoid activation fails to converge under variance based initialization. Moreover, from these numerical results and learning curves, one can make the following observations:

1. With variance based random initialization, tanh and Sigmoid* still con-

Activation	Train	Test
ReLU	0.998282	0.662619
Sigmoid*	0.893982	0.591162
Tanh	0.969449	0.619924
RReLU	0.997602	**0.716262**
Sigmoid	n/a	n/a

Table 5.3: CIFAR-100 Inception Network with Different Non-linearity

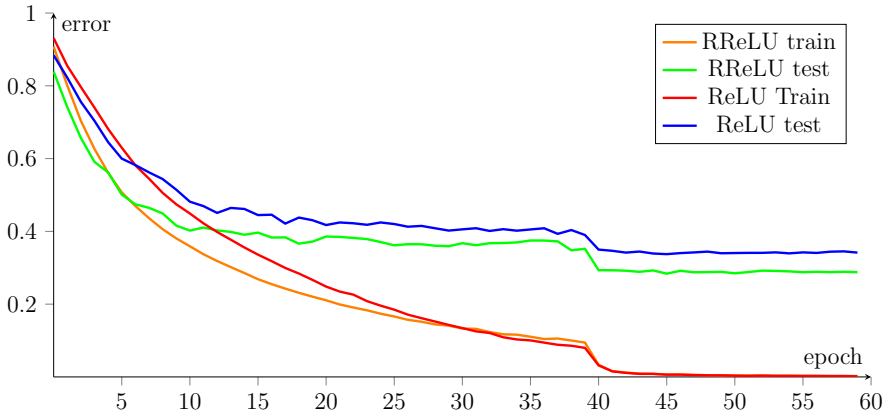

Figure 5.11: ReLU and RReLU Inception Network learning curve

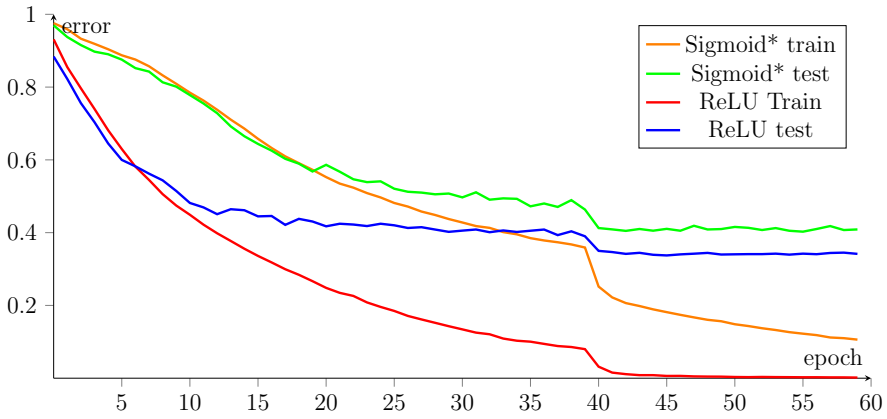

Figure 5.12: Sigmoid* and ReLU Inception Network learning curve

verge well in a 33 layer network. This suggests that the current understanding of the consequences of saturated activations is not accurate.

2. It is observed for activation functions that are able to converge from random initialization, the coefficient of term x in Taylor Series at 0^+ is 1.

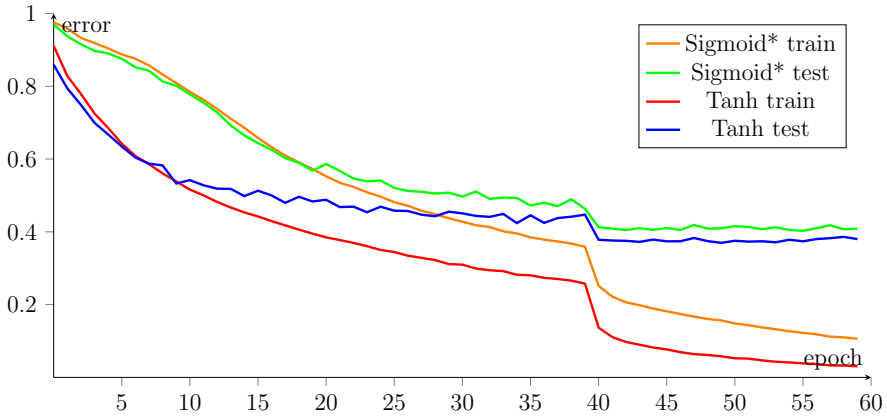

Figure 5.13: Tanh and Sigmoid* Inception Network learning curve

This experiment also suggests that the commonly ignored issue of "random initialization" plays an important role in neural network training. As a consequence, for past 20 years, many have failed to train deep Sigmoid networks. Put together, these observations suggest that we need to jointly consider the effects of initialization, activation and optimization in neural network training.

5.8 Conclusion

In this chapter, I analyzed four rectified activation functions and proposed the Sigmoid* activation function. My findings strongly suggest that the most popular activation function, ReLU, is not the end of the story: Three types of (modified) leaky ReLU all consistently outperform the original ReLU. Under Batch Normalization, RReLU still outperforms ReLU. However, the reasons for this superior performance still lacks rigorous justification from a theoretical standpoint. I also developed an explanation for the vanishing problem with sigmoid activations, and used this to develop a modified Sigmoid* function. This modified activation appears to correct the problem with sigmoid and demonstrates competitive perforance with Tanh. How these activations perform on large scale data still needs to be investigated, which remains an open question worth pursuing in the future.

Chapter 6

Application to Plankton Classification

6.1 National Data Science Bowl

The National Data Science Bowl (NDSB)[1] is a competition where the problem is to classify plankton animal images. Measuring the population of plankton animals is important for understanding the ocean ecosystem. The Hatfield Marine Science Center at Oregon State University provided the training data for this competition. The data [14] consists of grey scale images of various sizes, distributed among a total of 121 classes. The training data consisted of 30336 images, while the test data contained more than 10 thousand images. The classes are not mutually exclusive: minor differences in animal age and side shape will make the image belong to different classes.

I used this dataset to check the generalization capability of deep network models. My team achieved 2nd place over one thousand teams in this competition. The solution I developed for the Science Bowl with my teammates did not involve any prior knowledge or special design for the problem. I designed one core network structure and activation function used on all networks during this competition.

[1]National Data Science Bowl Competition: `https://www.kaggle.com/c/datasciencebowl`

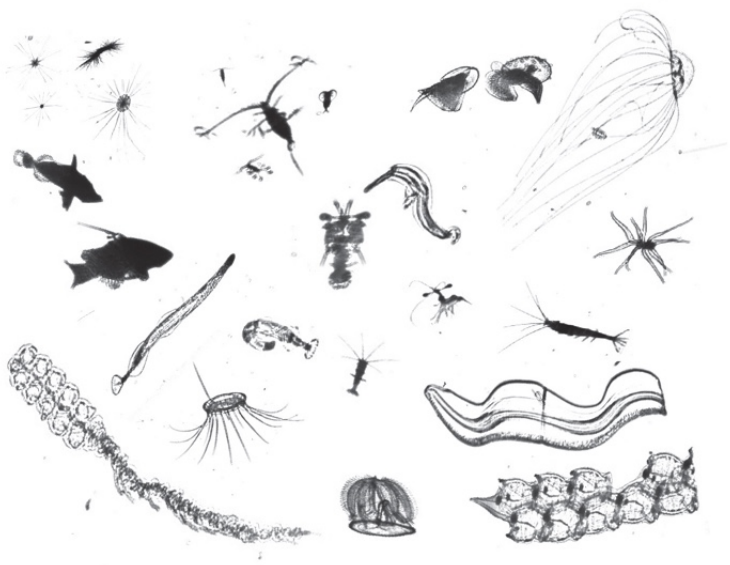

Figure 6.1: NDSB image example (Taken from homepage)

6.2 Network Design

During this competition, I focused on training very deep networks since the aim was to achieve better performance without any time constraints. The network design was mainly based on (Equation 4.5). The detailed network structure is shown in Table 6.1 and Table 6.2. For reference, I also used GoogLeNet (Table 4.4) and the TinyNet (Table 4.6) to evaluate alternative network structures, using the same hyper parameters as for training on the ImageNet data.

The first network's complexity is 12× compared to AlexNet while the second network's complexity is 6× compared to AlexNet. I considered three learning schedules; for learning rate 0.01 I ran 400 epochs, and for learning rate 0.001 and 0.0001 I ran 80 epochs.

For this problem, I ignored the problem specification, and used simple affine transformations to perform data augmentation. These augmentations included rotation, translation, rescaling, flipping, rescaling and shearing. I used multiple tests to obtain final result: For a single image, I performed 100 random augmentations then averaged the 100 prediction results.

type	kernel/stride	output
convolution	5x5/1	128x128x16
max pooling	3x3/2	64x64x16
convolution	3x3/1	64x64x32
convolution	3x3/1	64x64x32
convolution	3x3/1	64x64x64
convolution	3x3/1	64x64x64
convolution	3x3/1	64x64x128
convolution	3x3/1	64x64x128
convolution	3x3/1	64x64x128
convolution	3x3/1	64x64x128
max pooling	3x3/2	32x32x128
convolution	3x3/1	32x32x256
convolution	3x3/1	32x32x256
convolution	3x3/1	32x32x256
convolution	3x3/1	32x32x256
max pooling	3x3/2	16x16x256
convolution	3x3/1	16x16x512
convolution	3x3/1	16x16x512
convolution	3x3/1	16x16x512
convolution	3x3/1	16x16x512
max pooling	3x3/2	8x8x512
convolution	3x3/1	8x8x512
convolution	3x3/1	8x8x512
avg/max pooling	8x8/1	1x1x512
softmax		1x1x121

Table 6.1: NDSB Network-1

6.3 Result

The result is shown in Table 6.3. These result are reported by the online testing server directly. The GoogLeNet and TinyNet models were able to achieve results that rank them in the top 100 among competitors. The two maximum depth networks, NDSB Network-1 and NDSB Network-2, were able to achieve a top-10 ranking in the competition.

type	kernel/stride	output
convolution	5x5/1	144x144x16
max pooling	3x3/2	72x72x16
convolution	3x3/1	72x72x32
convolution	3x3/1	72x72x32
convolution	3x3/1	72x72x64
convolution	3x3/1	72x72x64
convolution	3x3/1	72x72x128
convolution	3x3/1	72x72x128
convolution	3x3/1	72x72x128
convolution	3x3/1	72x72x128
max pooling	3x3/2	36x36x128
convolution	3x3/1	36x36x256
convolution	3x3/1	36x36x256
convolution	3x3/1	36x36x256
convolution	3x3/1	36x36x256
max pooling	3x3/2	18x18x256
convolution	3x3/1	18x18x512
convolution	3x3/1	18x18x512
max pooling	3x3/3	6x6x512
convolution	3x3/1	6x6x512
convolution	3x3/1	6x6x512
avg/max pooling	6x6/1	1x1x512
softmax		1x1x121

Table 6.2: NDSB Network-2

Network	LogLoss on Test Data
NDSB-Net 1	0.606
NDSB-Net 2	0.609
VGGNet	0.631
GoogLeNet	0.710
TinyNet	0.759

Table 6.3: LogLoss of National Data Science Bowl

6.4 Conclusion

In this competition study, I found that, even though we ignored the problem details in designing the deep network models, the general design of a deep convolution network was able to achieve very good results. The maximum depth network design is able to provide state-of-art results, albeit with significant

computational cost. In this competition, the first place team proposed Cyclic pooling[2], but their single model performance was not significantly better than our best competitors.

[2]First place team method: `https://benanne.github.io/2015/03/17/plankton.html`

Chapter 7

Conclusion

In this dissertation I designed an efficient network structure that is able to replace the widely used AlexNet. I also proposed a new randomized activation function that improves the performance of convolutional networks. By using good structure designs and a new activation function, I was able to design a state-of-art network for plankton image classification. The end-to-end convolutional neural network is able to learn from raw pixel information on this problem, without any hand crafted features.

7.1 Future Work

I think that, in the future, the most important work that remains to be done in this area is to develop a theory of deep network learning. Without any theory, empirical progress makes the research seem more like an evolutionary process rather than science. Also, I think that using convolution networks to perform classification on image sequences, like videos, will also be a topic of growing importance.

7.1.1 Structure Theory

Currently, work on deep learning still relies heavily on empirically successful network structures. Although I have outlined some results in designing deep network architectures, I was still able to design an efficient TinyNet. There remains a gap in the theory behind network structure.

From LeNet-5 to VGGNet, we have witnessed how network depth can

dramatically improve classification performance. We now know how to train much deeper single path networks, but unfortunately, the cost of training these networks remains unacceptable in many commercial settings.

GoogLeNet and its Inception module provide a new view on how to design convolutional networks. Google suggests that there is a theory about sparsity that guides how to design an efficient network topology, but it is not revealed. I think it is important to investigate the theory behind network structure, including sparisty, to better understand why and how it this works inside a neural network.

7.1.2 Activation Function Theory

Similar to the case of network structure, the theory for activation functions is lacking in the literature. The use of saturated activation functions arises from a mere analogy with neurons, and lacks mathematics theory. The piece-wise linear activation functions, also known as "rectifiers", perform much better in practice than saturated activation functions. The current explanations for this success point to the sparsity induced by the ReLU activation, but the Leaky ReLU and its variants outperform the original ReLU, suggesting that the real benefit may not arise from output sparsity. However, with Batch Normalization, the specific activation functions become less important but still worth to investigate.

7.1.3 Convolution Network with Memory

Currently, all of the convolutional neural networks considered in this thesis are feed-forward networks. Such networks are able to classify single static images. For an image sequence, such as a video, one requires a network that can contain a longer memory. One solution to such problems is to use a recurrent neural network. However, current recurrent neural networks are focused on fully connected layers. A low hanging fruit would be to use a pre-trained convolutional neural network as a feature encoder, replacing the classifier with a recurrent network structure, for example, as in long short term memory [28].

A even more interesting question is how to modify a convolution layer with recurrent memory. Comparing an encoder based recurrent structures with this structure is also an open problem that is worth investigating.

7.2 Final Thought

In the three years since AlexNet appeared, traditional image classification has "almost" become a solved problem. Convolutional neural networks now surpass even human level performance on many datasets. The recent emergence of "deep learning" is more like the success of heterogeneous computing: the convolutional network was invented in 1989, and long short term memory was invented in 1997. Without powerful GPUs, it would not be possible to train these large neural network models on such large data sets. Although one can continue to pursue such an evolutionary process, because we will have even more powerful GPU, this is not what we should expect as serious researchers. Nevertheless, deep neural networks are now rapidly being adopted in industrial applications, broadening the use of artificial intelligence at a faster pace than ever before.

Bibliography

[1] Karpathy Andrej. Lessons learned from manually classifying cifar-10. `https://karpathy.github.io/2011/04/27/manually-classifying-cifar10/`, 2011. [Online; accessed 09-June-2015].

[2] Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, James Bergstra, Ian Goodfellow, Arnaud Bergeron, Nicolas Bouchard, David Warde-Farley, and Yoshua Bengio. Theano: new features and speed improvements. *arXiv preprint arXiv:1211.5590*, 2012.

[3] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *Computer vision–ECCV 2006*, pages 404–417. Springer, 2006.

[4] Yoshua Bengio, Ian J. Goodfellow, and Aaron Courville. Deep learning. Book in preparation for MIT Press, 2015.

[5] Yoshua Bengio, Pascal Lamblin, Dan Popovici, Hugo Larochelle, et al. Greedy layer-wise training of deep networks. *Advances in neural information processing systems*, 19:153, 2007.

[6] Léon Bottou. Stochastic gradient descent tricks. In *Neural Networks: Tricks of the Trade*, pages 421–436. Springer, 2012.

[7] Xudong Cao. A practical theory for designing very deep convolution neural networks. 2015.

[8] Ming-wei Chang, Wen-tau Yih, and Christopher Meek. Partitioned logistic regression for spam filtering. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 97–105. ACM, 2008.

[9] Ken Chatfield, Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. Return of the devil in the details: Delving deep into convolutional nets. *arXiv preprint arXiv:1405.3531*, 2014.

[10] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

[11] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[12] Olah Christopher. Neural networks, manifolds, and topology. `https://colah.github.io/posts/2014-03-NN-Manifolds-Topology/`, 2014. [Online; accessed 19-May-2015].

[13] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[14] Robert K. Cowen, S. Sponaugle, K.L. Robinson, and J. Luo. Planktonset 1.0: Plankton imagery data collected from f.g. walton smith in straits of florida from 2014-06-03 to 2014-06-06 and used in the 2015 national data science bowl (nodc accession 0127422). `http://data.nodc.noaa.gov/cgi-bin/iso?id=gov.noaa.nodc:0127422`, 2015.

[15] Navneet Dalal and Bill Triggs. Histograms of oriented gradients for human detection. In *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*, volume 1, pages 886–893. IEEE, 2005.

[16] Peter Dayan and Laurence F Abbott. *Theoretical neuroscience*, volume 806. Cambridge, MA: MIT Press, 2001.

[17] Jeffrey Dean, Greg Corrado, Rajat Monga, Kai Chen, Matthieu Devin, Mark Mao, Andrew Senior, Paul Tucker, Ke Yang, Quoc V Le, et al. Large scale distributed deep networks. In *Advances in Neural Information Processing Systems*, pages 1223–1231, 2012.

[18] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

[19] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier networks. In *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics. JMLR W&CP Volume*, volume 15, pages 315–323, 2011.

[20] Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron Courville, and Yoshua Bengio. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

[21] Benjamin Graham. Fractional max-pooling. *arXiv preprint arXiv:1412.6071*, 2014.

[22] Kaiming He and Jian Sun. Convolutional neural networks at constrained time cost. *arXiv preprint arXiv:1412.1710*, 2014.

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Spatial pyramid pooling in deep convolutional networks for visual recognition. *arXiv preprint arXiv:1406.4729*, 2014.

[24] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *arXiv preprint arXiv:1502.01852*, 2015.

[25] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[26] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[27] Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Master's thesis, Institut fur Informatik, Technische Universitat, Munchen*, 1991.

[28] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[29] Kurt Hornik. Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257, 1991.

[30] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[31] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe: Convolutional architecture for fast feature embedding. *arXiv preprint arXiv:1408.5093*, 2014.

[32] David B Kirk and W Hwu Wen-mei. *Programming massively parallel processors: a hands-on approach*. Newnes, 2012.

[33] Alex Krizhevsky and Geoffrey Hinton. Learning multiple layers of features from tiny images. *Computer Science Department, University of Toronto, Tech. Rep*, 1(4):7, 2009.

[34] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *NIPS*, pages 1097–1105, 2012.

[35] Simon Laughlin. A simple coding procedure enhances a neuron's information capacity. *Zeitschrift für Naturforschung c*, 36(9-10):910–912, 1981.

[36] Lawrence-Livermore-National-Laboratory. Asci white: The world's fastest computer - meeting the challenge of stockpile stewardship. `https://computation.llnl.gov/casc/sc2001_fliers/ASCI_White/ASCI_White01.html`, 2001. [Online; accessed 06-June-2015].

[37] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[38] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[39] Chen-Yu Lee, Saining Xie, Patrick Gallagher, Zhengyou Zhang, and Zhuowen Tu. Deeply-supervised nets. *arXiv preprint arXiv:1409.5185*, 2014.

[40] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*, pages 641–650. ACM, 2010.

[41] Mu Li, David G Andersen, Jun Woo Park, Alexander J Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *Proc. OSDI*, pages 583–598, 2014.

[42] Mu Li, David G Andersen, Alex J Smola, and Kai Yu. Communication efficient distributed machine learning with the parameter server. In *Advances in Neural Information Processing Systems*, pages 19–27, 2014.

[43] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[44] Min Lin, Shuo Li, Xuan Luo, and Shuicheng Yan. Purine: A bi-graph based deep learning framework. *arXiv preprint arXiv:1412.6249*, 2014.

[45] David G Lowe. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 60(2):91–110, 2004.

[46] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *ICML*, volume 30, 2013.

[47] MICROWAY. Detailed specifications of the intel xeon e5-2600v3 haswell-ep processors. `https://www.microway.com/knowledge-center-articles/detailed-specifications-intel-xeon-e5-2600v3-haswell-ep-processors/`, 2015. [Online; accessed 06-June-2015].

[48] Tomas Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *HLT-NAACL*, pages 746–751, 2013.

[49] Tom M Mitchell. *Machine learning*. McGraw-Hill Boston, MA:, 1997.

[50] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In *ICML*, pages 807–814, 2010.

[51] Yurii Nesterov. A method of solving a convex programming problem with convergence rate o (1/k2). In *Soviet Mathematics Doklady*, volume 27, pages 372–376, 1983.

[52] NVIDIA. Tesla gpu accelerators for servers. `http://www.nvidia.ca/object/tesla-servers.html`, 2015. [Online; accessed 06-June-2015].

[53] Florent Perronnin, Yan Liu, Jorge Sánchez, and Hervé Poirier. Large-scale image retrieval with compressed fisher vectors. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3384–3391. IEEE, 2010.

[54] Florent Perronnin, Jorge Sánchez, and Thomas Mensink. Improving the fisher kernel for large-scale image classification. In *Computer Vision–ECCV 2010*, pages 143–156. Springer, 2010.

[55] Boris Teodorovich Polyak. Some methods of speeding up the convergence of iteration methods. *USSR Computational Mathematics and Mathematical Physics*, 4(5):1–17, 1964.

[56] Dean A Pomerleau. Alvinn: An autonomous land vehicle in a neural network. Technical report, DTIC Document, 1989.

[57] Matthew Richardson, Ewa Dominowska, and Robert Ragno. Predicting clicks: estimating the click-through rate for new ads. In *Proceedings of the 16th international conference on World Wide Web*, pages 521–530. ACM, 2007.

[58] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning internal representations by error propagation. Technical report, DTIC Document, 1985.

[59] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5, 1988.

[60] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 2015.

[61] Jorge Sánchez and Florent Perronnin. High-dimensional signature compression for large-scale image classification. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 1665–1672. IEEE, 2011.

[62] Terrence J Sejnowski and Charles R Rosenberg. Parallel networks that learn to pronounce english text. *Complex systems*, 1(1):145–168, 1987.

[63] Amar Shan. Heterogeneous processing: a strategy for augmenting moore's law. *Linux Journal*, 2006(142):7, 2006.

[64] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[65] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15:1929–1958, 2014.

[66] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. Highway networks. *arXiv preprint arXiv:1505.00387*, 2015.

[67] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML-13)*, pages 1139–1147, 2013.

[68] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *arXiv preprint arXiv:1409.4842*, 2014.

[69] Oriol Vinyals, Alexander Toshev, Samy Bengio, and Dumitru Erhan. Show and tell: A neural image caption generator. *arXiv preprint arXiv:1411.4555*, 2014.

[70] Strother H Walker and David B Duncan. Estimation of the probability of an event as a function of several independent variables. *Biometrika*, 54(1-2):167–179, 1967.

[71] Minjie Wang, Tianjun Xiao, Jianpeng Li, Jiaxing Zhang, Chuntao Hong, and Zheng Zhang. Minerva: A scalable and highly efficient training platform for deep learning, 2014.

[72] Naiyan Wang and Dit-Yan Yeung. Learning a deep compact image representation for visual tracking. In *Advances in Neural Information Processing Systems*, pages 809–817, 2013.

[73] Matthew D Zeiler and Rob Fergus. Stochastic pooling for regularization of deep convolutional neural networks. *arXiv preprint arXiv:1301.3557*, 2013.