Neural Networks with Input Specified Thresholds

Fei Liu
Stanford University
liufei@stanford.edu

Junyang Qian
Stanford University
junyangq@stanford.edu

Abstract

In this project report, we propose a method to modify the ReLU layers in neural networks. The modification involves two stages: we first convert the bias parameters into threshold parameters, and then increase the number of threshold parameters to balance the "linear side" and the "nonlinear side" of the model, which could potentially make the model more flexible and powerful. Image classification tests are performed on the Tiny ImageNet dataset.

1. Introduction

Neural networks are powerful models for visual recognization and image classification. In this project, we will focus on image classification problems with neural networks.

For the image classification problem, neural networks use a score function to rank the classes. The score function maps the input, which is the image data, to the output, which is a list of scores of the corresponding classes. In principle, we could assume that there exists some form of a score function in reality, which is rather complicated and highly nonlinear, that we human use to really classify and recognize objects. However, it is far from being understood how such processes are done by human brains. Nonetheless, our task is only to find an "approximation" of the true underlying score function with neural networks.

From a high level point of view, the score functions in neural networks are composed of simple functions in form of hierarchical layers. These simple functions include affine functions, which we refer to as the "linear side" of the model, and nonlinear functions such as the tanh function $\tanh(x)$, the sigmoid function $\sigma(x) = 1/(1 + e^{-x})$ and the ReLU function $\max(x, 0)$, which we refer to as the "nonlinear side" of the model.

Among the nonlinear functions, the ReLU function receives the most attention for several reasons, one of which is that the gradient is not close to zero on most part of the domain as the tanh function and the sigmoid function do, which alleviates the dead neuron problem caused by stagnant gradient flows as would most likely to appear if the tanh function or the sigmoid function are used. However, the ReLU function still has zero gradient on the negative real axis. To address this issue, several approach are proposed, such as the PReLU [2], where the slope of the ReLU function on the negative real axis is treated as a parameter instead of fixed to be zero.

In this report we use a different approach to deal with the zero gradient issue of the ReLU function, which we refer to as the "bias to threshold conversion". The main idea is that, we replace the zero-threshold in the ReLU function with the bias-threshold, which convert the the bias terms into the threshold terms. This is our first modification of the existing neural network model, which reassembles the [affine - ReLU] layer into the [linear - threshold] layer.

Our second modification, which we call the "input specified thresholds", increases the number of threshold parameters so that the number of threshold parameters are comparable to the number of weight parameters, which indicates a more balanced choice between the "linear side" and the "nonlinear side" of the model.

The rest of the report are organized as follows. In Section 2 we describe our modified models in details. In Section 3 we present the test results for the Tiny ImageNet Challenge. Conclusions are given in Section 4.

2. Description of the Modified Models

In this section, we present the modified models in details. We will give a complete description for the case of fully connected neural networks, and then briefly talk about the case for the convolutional neural networks.

2.1. Modifying the ReLU Layers in Fully Connected Neural Networks

We start with modifying the ReLU layers in fully connected neural networks. There are two modifications:

1. **Bias to Threshold Conversion**
   Converting the bias terms into threshold terms.

2. **Input Specified Thresholds**
   Increasing the number of threshold parameters to be comparable to the number of weight parameters.

The details are discussed as follows.

2.1.1 Bias to Threshold Conversion

In this section we describe the bias to threshold conversion technique.

Suppose $x \in \mathbb{R}^{1\times D}$ is the input, $w \in \mathbb{R}^{D\times 1}$ is the weight and $b \in \mathbb{R}^{1\times 1}$ is the bias. An "affine - ReLU" forward pass will do the following

1. Compute $y = xw + b$.
2. Compute $z = \max(y, 0)$.

All together, we have

$$z = \max(xw + b, 0). \tag{1}$$

The potential problem is that, if $xw + b < 0$, then during the backward pass, the gradient will be cut off locally, which could potentially cause the dead neuron problem. To fix it, we move the bias b to replace the zero-threshold and we consider

$$z^{\text{new}} = \max(xw, b). \tag{2}$$

We see that, in (2), b serves as a threshold instead of a bias as in (1). The advantage is that, no matter what values of w, x, b are, at least part of the variables will carry on the gradient flow during the back propagation, which could potentially diminish the "dead neuron" phenomena.

Figures 1 and 2 show a visualization of the bias to threshold conversion, where

$$x = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \in \mathbb{R}^{1\times 2},$$
$$w = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \in \mathbb{R}^{2\times 1}, \quad b \in \mathbb{R}^{1\times 1}.$$

At first sight, one may think that, if we use (2), then the bias term is dropped during the forward pass since b now serves only as a threshold, which could possibly cause the network to be less powerful. It turns out that this is not the case. We will show that, it is no harm to convert the bias term into the threshold term for the intermediate layers.

For simplicity, let us consider a two-layer neural network, or a one-hidden-layer neural network. For an input $x \in \mathbb{R}^{1\times D}$, the final class score is given by

$$s(x) = \max(xW_h + b_h, 0)W_c + b_c, \tag{3}$$

where $W_h \in \mathbb{R}^{D\times D_h}, W_c \in \mathbb{R}^{D_h\times D_c}$ are the weight matrices and $b_h \in \mathbb{R}^{1\times D_h}, b_c \in \mathbb{R}^{1\times D_c}$ are the bias vectors. D_h is the number of neurons in the hidden layer and D_c is the number of classes. Now note that

$$\begin{aligned} s(x) &= \max(xW_h + b_h, 0)W_c + b_c \\ &= (\max(xW_h, -b_h) + b_h)W_c + b_c \\ &= \max(xW_h, -b_h)W_c + b_hW_c + b_c \\ &= \max(xW_h, -b_h)W_c + (b_hW_c + b_c). \end{aligned}$$

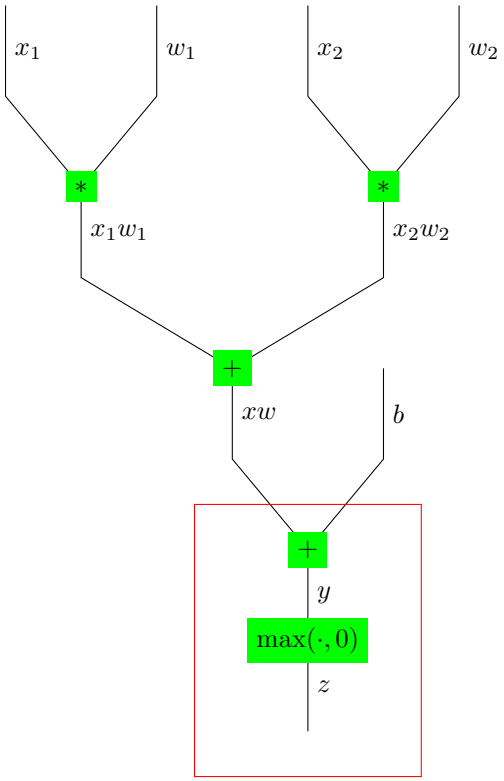


Figure 1. Computational graph of $z = \max(xw + b, 0)$. The part inside the red rectangle is to be modified.

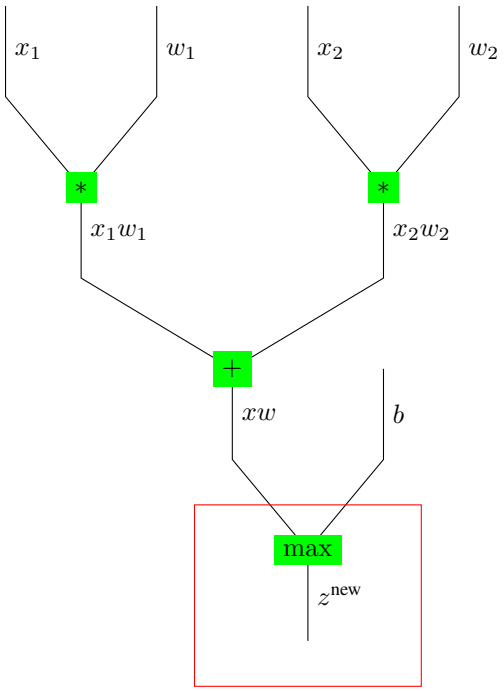


Figure 2. Computational graph of $z^{\text{new}} = \max(xw, b)$. The part inside the red rectangle is what has been modified.

We define

$$\begin{aligned} W'_h &:= W_h, \\ b'_h &:= -b_h, \\ W'_c &:= W_c, \\ b'_c &:= b_h W_c + b_c, \end{aligned}$$

then we have the equality

$$\begin{aligned} s(x) &= \max(xW_h + b_h, 0)W_c + b_c \\ &= \max(xW'_h, b'_h)W'_c + b'_c, \end{aligned}$$

which means that we have found a new set of parameters W'_h, b'_h, W'_c, b'_c combined with the new form (2), which results a modified model that gives the same scores as the original one does. Intuitively, what we did is that, we transform the bias term b_h into a threshold term, and "shift" the bias effect to the next layer. This "shift" can be reflected mathematically in the expression of b'_c. One can see that this technique can be used recursively in the network, i.e., we can "shift" the bias effect all the way down to the last affine layer. Thus it is enough for the intermediate b_i's to serve as thresholds only, and it will not hurt the power of the model.

Note that, after the bias to threshold conversion, a traditional [affine(W, b) - ReLU] layer is reassembled into a [linear(W) - threshold(b)] layer, where the parameter W and b are detached from each other and can be taken care of in different layers. (The notation "affine(W, b)" stands for an affine layer with parameters W and b, similar for the other layer notations.) This detachment motivates us the second modification which will be specified below.

2.1.2 Input Specified Thresholds

In this section we present the idea of input specified thresholds.

Let us take a closer view of the current model after bias to threshold conversion. Suppose we are given a combined layer

$$\text{affine}(W_{\text{prev}}, b) \text{ - ReLU - affine}(W, b_{\text{next}}) \text{ - ReLU},$$

then with the modification in Section 2.1.1, we can reassemble it as

$$\text{linear}(W_{\text{prev}}) \text{ - threshold}(b) \text{ - linear}(W) \text{ - threshold}(b_{\text{next}}).$$

Now, we focus on the two-layer-combo [threshold(b) - linear(W)]. Suppose x is the output vector from the linear(W_{prev}) layer. (Notice that in our setting, x is a single output, which is a row vector. It is not a matrix containing lots of rows as multiple data outputs.) The forward propagation of the two-layer-combo gives

1. Compute $y = \max(x, b)$.
2. Compute $z = yW$.

The $\max(\cdot, \cdot)$ operator above is the elementwise maximum operator, which is python-style rather than standard mathematical notation. To be clear, we list the python code below for clarification

```
# ...

# threshold layer
y = np.maximum(x, b)

# linear mapping layer
z = y.dot(W)

# ...
```

For the layer threshold(b), we see that, x, the output of the layer linear(W_{prev}), is compared elementwisely with the threshold vector b. We call this process as an "output specified" process since the threshold b has a one-to-one correspondence with x.

Examining the model more carefully, we see that, each output of a single neuron (after threshold) is accepted by lots of neurons in the next linear layer, which means, each entry of y is multiplied by lots of entries (a whole row entries) of W_2, instead of a single number. This motivates the idea of "input specified" process – we should assign different threshold values if the receiving neurons in the next layer are different, which could potentially strengthen the power of our model. Specifically, we introduce a threshold matrix B which has the same size with W and we consider the following forward pass procedure

1. Compute $Y = \max(x^T, B)$.
2. Compute $z^{\text{new}} = (Y * W).\text{sum(axis=0)}$.

The operators above are a mixture of standard mathematical operators and python-style operators. We list the corresponding python code below to avoid ambiguity

```
# ...

# new threshold layer, input specified
Y = np.maximum(x.reshape((-1, 1)), B)

# modified linear mapping layer
z_new = (Y * W).sum(axis=0)

# ...
```

In words, this means that the threshold is now paired with the input side (the weight matrix W) instead of the output side (the vector x).

Now we visualize the modification process by computational graphs. We consider the following instances

$$x = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \in \mathbb{R}^{1\times 2}, \quad W = \begin{bmatrix} W_{1,1} & W_{1,2} \\ W_{2,1} & W_{2,2} \end{bmatrix} \in \mathbb{R}^{2\times 2},$$
$$y = \begin{bmatrix} y_1 & y_2 \end{bmatrix} \in \mathbb{R}^{1\times 2}, \quad Y = \begin{bmatrix} Y_{1,1} & Y_{1,2} \\ Y_{2,1} & Y_{2,2} \end{bmatrix} \in \mathbb{R}^{2\times 2},$$
$$b = \begin{bmatrix} b_1 & b_2 \end{bmatrix} \in \mathbb{R}^{1\times 2}, \quad B = \begin{bmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{bmatrix} \in \mathbb{R}^{2\times 2},$$
$$z = \begin{bmatrix} z_1 & z_2 \end{bmatrix}, \quad z^{\text{new}} = \begin{bmatrix} z_1^{\text{new}} & z_2^{\text{new}} \end{bmatrix}.$$

Figures 3 and 4 provide the computational graphs for the instances above.

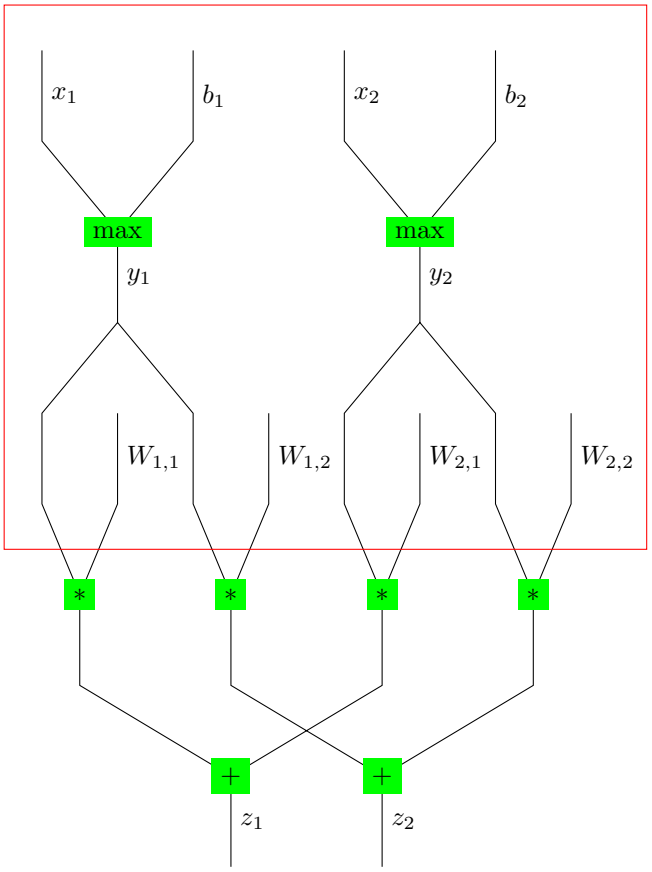


Figure 3. Computational graph of z. The part inside the red rectangle is to be modified.

Figures 5 and 6 provided the condensed versions of the computational graphs, where the parameters are merged into nodes. We can see from Figures 5 and 6 clearly the correspondence of b to x (the output from previous linear layer), and the correspondence of B to W (the weight matrix on the input side).

We see that, after adding these new thresholds, the total number of parameters is still of the same order with the unmodified one, specifically, no larger than twice the number

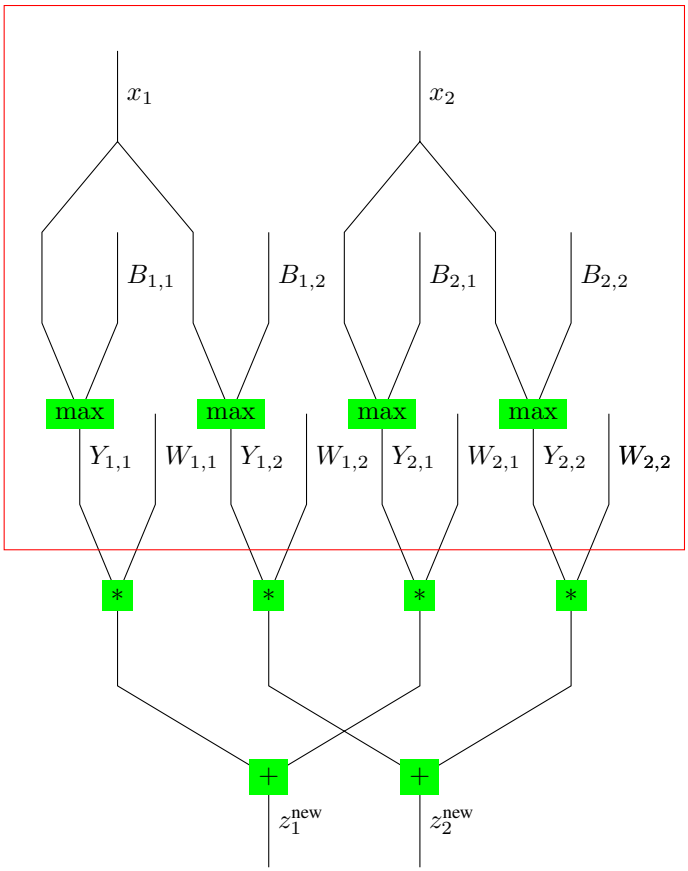


Figure 4. Computational graph of z^{new}. The part inside the red rectangle is what has been modified.

of parameters of the unmodified model, since the threshold matrices B_i's have the same size as the weight matrices W_i's. Moreover, the number of parameters controlling the thresholds is now roughly the same as the number of parameters controlling the weights, which indicates a more "balanced" choice. We know that the thresholds contribute to the nonlinear side of the model while the weights contribute to the linear side. It is reasonable to believe that, a more balanced contribution from the linear side and the nonlinear side could result a "wiser" model.

2.1.3 Back Propagation

We give the back propagation formula of our modified two-layer-combo for a single input as follows

1. Compute $\mathrm{d}W = \mathrm{d}z^{\text{new}} * Y$.
2. Compute $\mathrm{d}Y = \mathrm{d}z^{\text{new}} * W$.
3. Compute $\mathrm{d}B = \mathrm{d}Y * (Y == B)$.
4. Compute $\mathrm{d}x = (\mathrm{d}Y * (Y != B))$.sum(axis=1).

The python code is

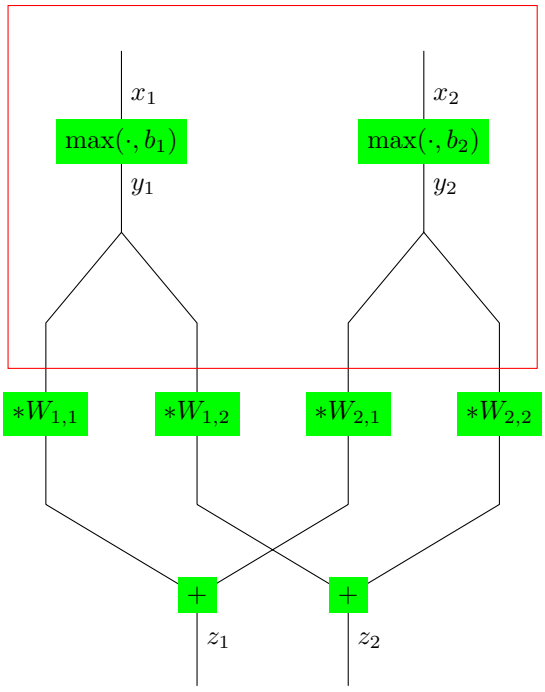


Figure 5. Computational graph of z (condensed version). The part inside the red rectangle is to be modified.

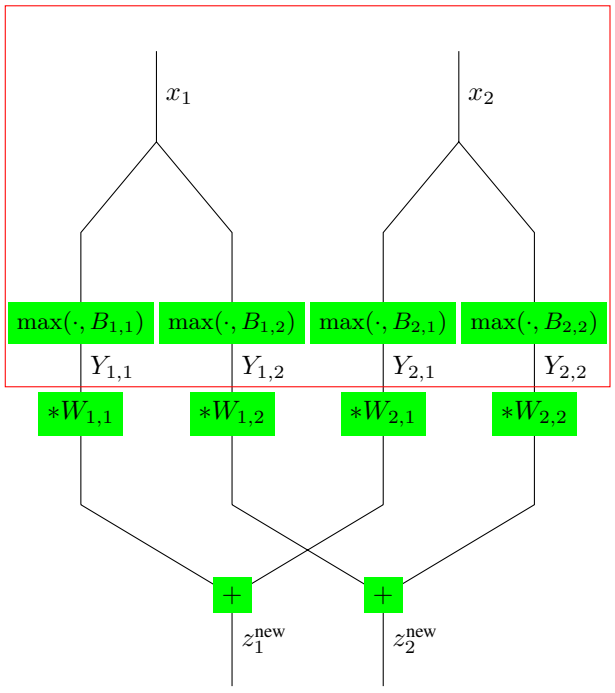


Figure 6. Computational graph of z^{new} (condensed version). The part inside the red rectangle is what has been modified.

```
# ...

dW = dz_new * Y
dY = dz_new * W
dB = dY * (Y == B)
dx = (dY * (Y != B)).sum(axis=1)

# ...
```

For the gradient with multiple inputs, we need to average the gradients for single input and then plus the regularization term if needed.

2.2. Modified Models for the Convolutional Neural Networks

The modification presented in Section 2.1 can be generalized to convolutional neural networks easily. We first use the similar technique to convert the bias term into the threshold term, and then we pair the threshold with the input side. To be specific, suppose we are propagating from a layer with D_1 filters to a layer with D_2 filters with ReLU layer in the middle. For the unmodified model, we have D_1 bias values for the first layer. After the modification, we will have $D_1 \times D_2$ threshold values, each of which corresponds to a pair $(\text{Filter}_1, \text{Filter}_2)$ where Filter_1 is a filter in the propagating-from layer and Filter_2 is a filter in the propagating-to layer. This relationship is similar to what we covered in Section 2.1, since the role of the depth (or number of the filters) of the convolutional layer is the same as the role of the number of neurons in the fully connected layer.

3. Experiments

To take advantage of recent advances in image recognition and get a better comparison of the traditional ReLU and the modified layer, we adopt transfer learning method in the experiments. The image features are extracted from a pre-trained model and used as input to train some additional layers. We compare the accuracies / error rates by two methods.

3.1. Dataset: Tiny ImageNet

We use the Tiny ImageNet dataset. Compared with the original ILSVRC15 dataset [4] (1,000 classes and more than a million of images), this Tiny ImageNet has 200 classes, each with 500 images in the training set and 50 images in the validation set. That amounts to 100,000 images in the training set and 10,000 images in the validation set. In addition, a test set of 10,000 unlabeled images is also provided to evaluate the final performance. Each image is either $64 \times 64 \times 3$ (color) or $64 \times 64 \times 1$ (gray-scale).

3.2. Data Preprocessing and Augmentation

Considering the contribution of background in CNNs, we conduct random cropping and flipping on the data to expose the model to more variations without additional data collection and annotation. In particular, for each image, we

create 5 instances of $56 \times 56 \times 3$ by cropping and for each instance, we create another one by left-right flipping. In this sense, we expand the data size by 10 times and this will help reduce overfitting and improve test performance.

The pre-trained Inception-v3 model that will be mentioned later expects an input of a 299×299 RGB image. Recall that the images in our dataset are either $64 \times 64 \times 3$ or $64 \times 64 \times 1$. For the former, we do a bilinear interpolation of the pixel values for each channel to expand the image to the desired size. For the latter, we first do bilinear interpolation for the single channel and then replicate that two times as the other two channels.

3.3. Model Configuration

We adopt transfer learning using the pre-trained Inception-v3 model [6] - a deep convolutional network built on top of GoogLeNet [5] utilizing Inception architectures. The outline of this network is shown in Table 1. This model is shipped with Tensorflow so that we can easily plug in and extract features from any layer in the network.

type	patch size / stride	input size
conv	$3 \times 3/2$	$299 \times 299 \times 3$
conv	$3 \times 3/1$	$149 \times 149 \times 32$
conv padded	$3 \times 3/1$	$147 \times 147 \times 32$
pool	$3 \times 3/2$	$147 \times 147 \times 64$
conv	$3 \times 3/1$	$73 \times 73 \times 64$
conv	$3 \times 3/2$	$71 \times 71 \times 80$
conv	$3 \times 3/1$	$35 \times 35 \times 192$
$3 \times$ Inception	Type 1	$35 \times 35 \times 288$
$5 \times$ Inception	Type 2	$17 \times 17 \times 768$
$2 \times$ Inception	Type 3	$8 \times 8 \times 1280$
pool	8×8	$8 \times 8 \times 2048$
linear	logits	$1 \times 1 \times 2048$
softmax	classifier	$1 \times 1 \times 1000$

Table 1. Outline of Inception [6] network architecture.

Although we are faced with a very similar task as the Inception model, the number of classes (200 here) and class labels are different. We need to fine-tune and retrain the network to some level. To do this, we replace the last two layers (linear and softmax) of the original network with one of the two architectures listed in the following table, use the features extracted from the input to the original linear layer and only train this part to get our classifier.

	Original Architecture	Modified Architecture
INPUT	Features from Pre-trained Model	
0	Affine	Threshold
1	ReLU	*Constrained Linear
2	Affine	Affine
3	Softmax	Softmax

*Constrained Linear: After thresholding, a single D-dimensional data vector is expanded to a $D \times H$ matrix. Constrained linear here means this is not a free linear mapping $\mathcal{L} : \mathbb{R}^{D \times H} \rightarrow \mathbb{R}^{H}$, but a constrained linear mapping taking the form $\mathcal{L}(X) = 1^{T}(W * X)$.

3.4. Training Methodology and Results

We train the two architectures separately with the same input features extracted from the pre-trained model. A 500-dimensional hidden layer was used for both of the architectures. We used mini-batches of size 50, ran 60 epochs for both experiments with Adam algorithm [3]. The best learning rate found was 5e-4 with an exponential decay of 0.95 per epoch. To alleviate overfitting, we use L2 regularization on the weight matrices with regularization parameter $\lambda =$ 2e-3. The training details for the modified architecture are shown in Figure 7 and 8.

We compare the test error for both architectures and observe $1\% \sim 2\%$ improvement of the threshold-based architecture over the original ReLU one in the validation error across different experiments.

The computation was done using the Tensorflow distributed machine learning system [1] on Amazon Web Services (AWS) that has one NVIDIA GPU with 1,536 CUDA cores and 4GB of video memory.

We achieved 33.8% test error rate, or 66.2% test accuracy.

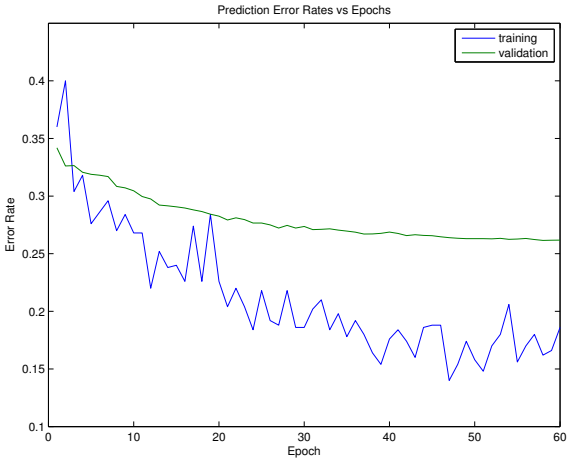


Figure 7. Prediction Error Rates vs Epochs under Modified Architecture.

3.5. Error Analysis

We can see the distribution of within-class misclassifications from Figure 9. For most classes, the number of misclassified images is between 5 and 15. That means the performance of this algorithm is stable across classes.

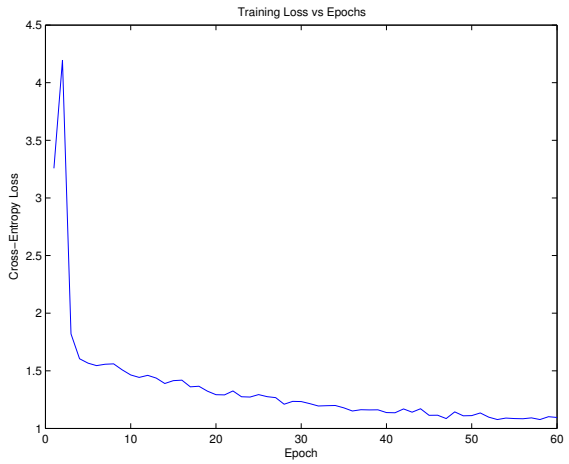


Figure 8. Training Loss vs Epochs under Modified Architecture.

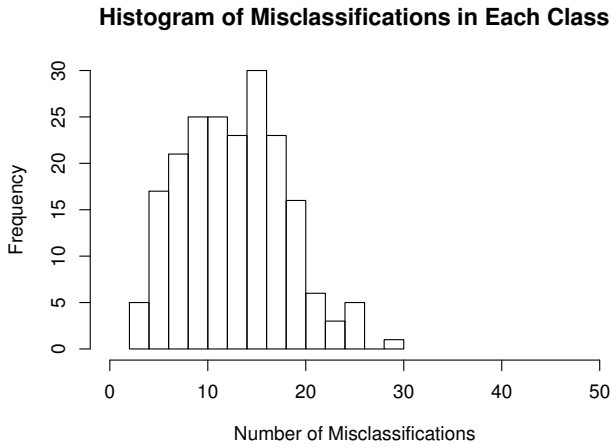
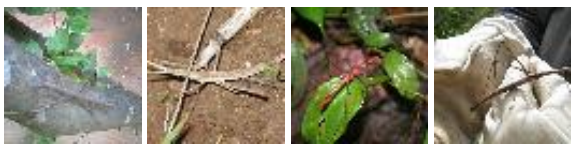


Figure 9. Histogram of number of misclassified images within each true class in the validation set.

If we look at the extreme cases, we find the class with most misclassified instances is *(walking stick, stick insect)*, *(plunger, plumber's helper)* and *(wooden spoon)*, which misses 29, 26, 26 out of 50 predictions respectively. We take some sample images from the training set.

- *walking stick, stick insect*



The top-3 confused ones are

dragonfly (5) mantis (4) slug (2)

- *plunger, plumber's helper*

The top-3 confused ones are

bathtub (6) koala (1) ladybug (1)

- *wooden spoon*

The top-3 confused ones are

frying pan (4) broom (2) drumstick(2)

The results are in accordance with our intuition. For the stick insect class, we see that the number of pixels relevant to this object class is far less than the ones related to the background. They are easily confused with other objects that have similar activity environments. It is very hard even for the neural networks to capture the characteristics in the face of rich background pixels. For the plunger and wooden spoon, they are often misclassified due to similar shape, materials or colors.

4. Conclusion

In this project, we propose a new model with two modifications over the ReLU layer. The first is the bias to threshold conversion. The second is the input specified thresholds.

Tests are performed for the Tiny Imagenet dataset. Satisfying results are achieved.

The current tests only involve the modification for the ReLU layer before fully connected affine layers. It is interesting to implement the modification for the case of convolutional layers.

References

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous systems, 2015. *Software available from tensorflow. org*.

[2] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. *CoRR*, abs/1502.01852, 2015.

[3] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[4] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[5] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[6] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna. Rethinking the inception architecture for computer vision. *arXiv preprint arXiv:1512.00567*, 2015.