Neural Networks Assignment 2

Following Report Guidelines Author Names, Emails, and Group Number are Excluded

April 28, 2020

1 Convolutional Neural Networks

1.1 Introduction

In this section, we chose the MNIST which is is a large database of handwritten digits as an example to show our understanding about the convolutional neural networks,as know as CNN, in the TensorFlow.

Basically, CNNs contains three components. Firstly, convolutional layers apply a specified number of filters to convolve with the image. A single value will be generated by some mathematical operations on each scanning. Generally, this layer will also apply ReLU activation function to the value in order to make the network nonlinear. Secondly, pooling layers downsample the output of convolutional layers to reduce the dimensionality of the feature map. Two common strategies are often used here which are mean and max pooling. Mostly we use max pooling which keeps the maximum value of that subregions and discards all other value. Thirdly, dense layers use the outputs of last layer to perform classification. In this layer, every node is connected to every node in the preceding layer which is also called 'fully connected'.

Generally, a typical CNN is composed of a set of convolutional modules that extract features. Each module has a convolutional layer followed by a pooling layer. The last module is followed by dense layers. The final dense layer contains a single node for each class. Softmax functition will produce the likelihood of how likely it is that the image falls into each class.

1.2 Data

The MNIST dataset can be easily imported from the keras's database. It contains 60,000 training examples and 10,000 test examples of the handwrittend digits 0-9. Each image is monochrome and has 28×28 pixels.

1.3 Network

There were only two kinds of layer in the example that we haven't discussed yet. They are 'Flatten layer' which flattens the input but does not affect the batch size and 'Dropout layer' which consists randomly setting a fraction of input units to 0 at each update during training time and helps prevent overfitting.

1.4 Results

The training accuracy and loss after each epoch are shown in figure1. Besides, the test accuracy and loss of our example are 0.9906 and 0.027846, respectively.

2 Recurrent Neural Networks

2.1 Introduction

In our daily life, for example, the order of words is an extremely important part during the communication. Traditional artificial networks are not able to take in the information of order. A recurrent neural network is used to deal with sequence data with considering the order. The output from previous step will be propagated as input of the current step. In order words, the current state, which is the most important feature of a RNN, is not only determined by the new input but also affected by the output of last layer.

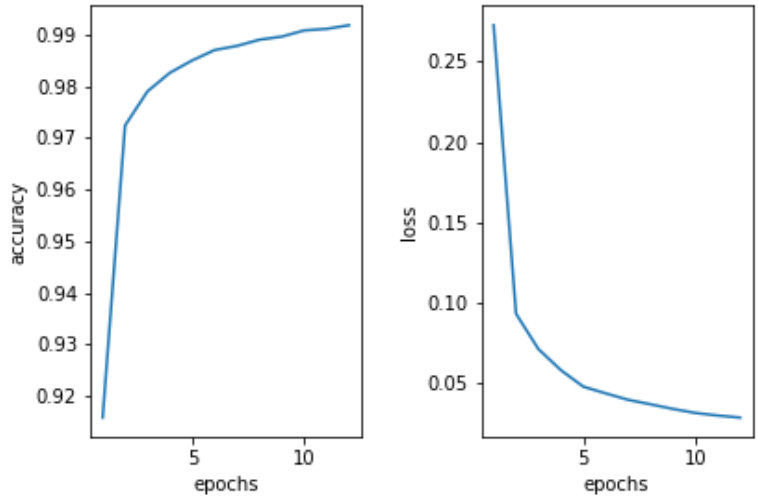

Figure 1: Training result of MNIST

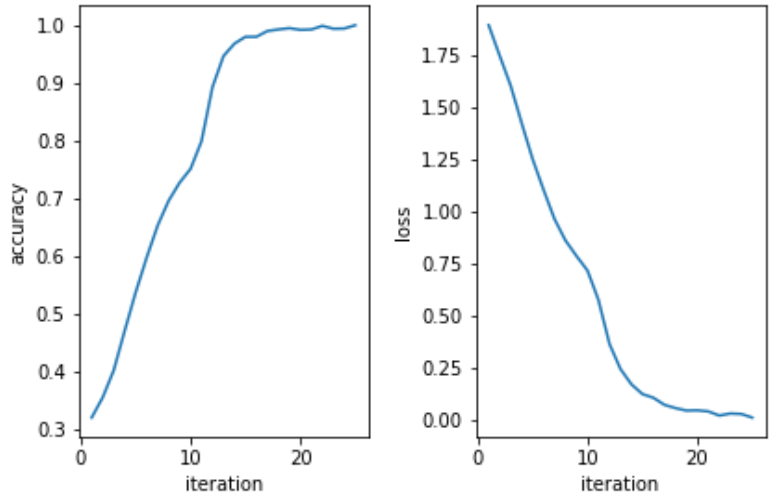

Figure 2: Result of addition network

2.2 Data

We chose integers addition, such as calculating $77 + 200$ then comparing the output of the network with 277, as an example for this section. We have done 30 iterations of calculations and we did 45,000 training samples and 5,000 testing samples during each iteration.

2.3 Network

The example use two LSTM layers, which refer to long short term memory and have LSTM cells. These cells have input gate, forget gate and output gate. The various gates enable the cell to take in new input as well as the previous output, discard part of the old information and extract new output state in order to prevent gradient vanishing and gradient exploding.

2.4 Results

The results are shown in figure2.

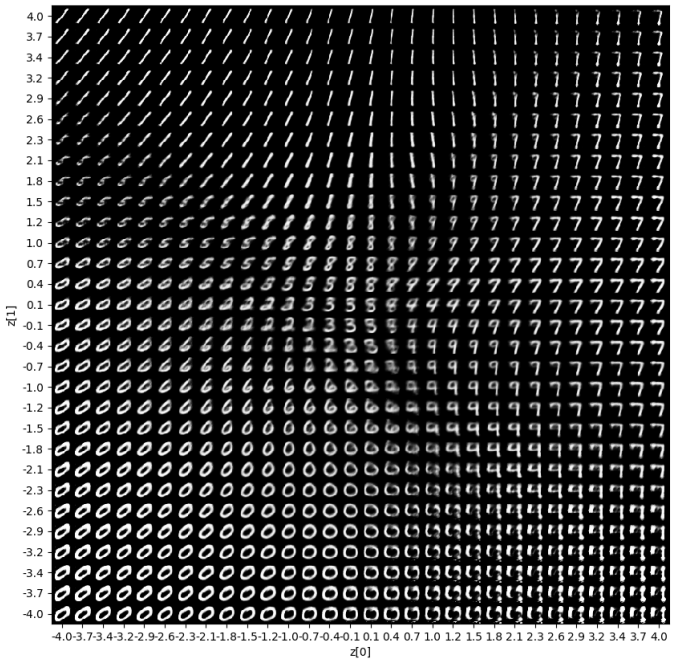

Figure 3: VAE reconstructed MNIST image

3 AutoEncoders

3.1 Introduction

In many deep learning tasks, we need to deal with various forms of specific and complex high-dimensional data, such as images, texts, audio and so on. However, the information behind these complex forms can be greatly expressed in a lower dimension. AutoEncoders can be used to compress the core features of data extraction, which can simplify complex original data. Generally, AutoEncoders consist of two parts which are encoder and decoder. The encoder uses a set of network to compress the data and extract lower dimensional data as outputs. The decoder decompresses the outputs of encoder and reconstruct the original data. The training process will optimize the reconstruction loss and find a good way to compress the data.

3.2 Data

We take Variational AutoEncoder(VAE) compressing MNIST digits images as an example.

3.3 Network

The example uses VAE to encoder the images. The difference between VAE and traditional AutoEncoder is that the latent vectors of hidden layer in VAE are expressed by a distribution with its mean value and standard deviation.

3.4 Results

One of the results is shown in figure3.

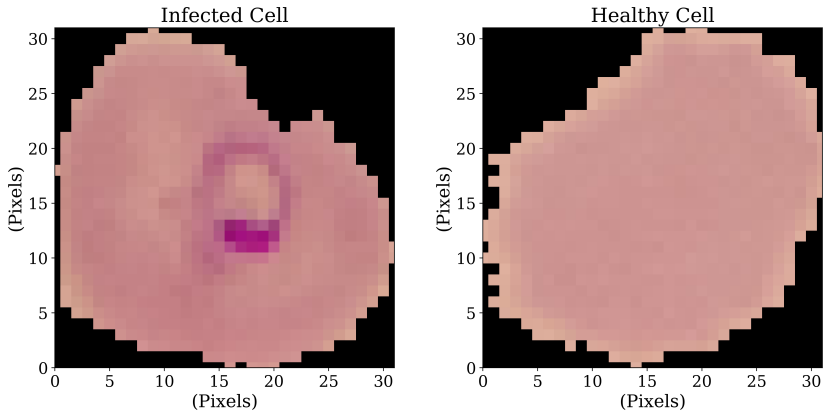

Figure 4: Example of parasitized (left) and uninfected (right) cells. Depending on the smearing technique, both infected and healthy cells may appear blue or purple rather than the pink shown here.

4 The Challenge: Detecting Malaria Parasites

4.1 Introduction, Data

Here we apply deep learning concepts to the challenge of identifying the protozoan parasites that cause Malaria in images of red blood cells. Counting cells infected with such parasites is not only useful in diagnosing Malaria, a disease that killed an estimated 2.855 million from 2010 to 2015 [1], but can also help in characterizing drug effectiveness and resistance [2]. The Lister Hill National Center for Biomedical Communications (LHNCBC), part of National Library of Medicine (NLM), developed a smartphone application to compile blood smear slides in conjunction with a conventional microscope [2]. Images from 50 healthy patients and 150 infected patients were gathered at the Chittagong Medical College Hospital, Bangladesh and classified at the Mahidol-Oxford Tropical Medicine Research Unit in Bangkok, Thailand [3]. The resulting database includes 27,558 RGB images, an equal number (13,779) parasite-infected and healthy cells normalized to be 32 $\times$ 32 pixels. Examples of healthy and infected cells are given in Fig. 4.

4.2 Network

In order to accomplish this goal, we began by constructing a initial convolutional network outlined in Tab. 1. We employed a Keras Sequential model expecting a 32 $\times$ 32 $\times$ 3 input layer for our RGB images. Our first layer after the input had 16 filters and used a 2 $\times$ 2 convolutional window with the default unit stride and a rectified linear unit activation function (i.e., max(x,0)). With padding such that the width and height of the output were equal to the input, this resulted in 32 $\times$ 32 $\times$ 16 output. This output was then pooled in order to decrease size and over-fitting, using maximum values in 2 $\times$ 2 windows resulting in a 16 $\times$ 16 $\times$ 16 output. This was then followed by convolutional layer similar to the first but with a filter depth of 32, resulting in a 16 $\times$ 16 $\times$ 32 output which was reduced with 2 $\times$ 2 max-pooling to be 8 $\times$ 8 $\times$ 32. An additional 8 $\times$ 8 convolutional layer with a depth of 64 and 2 $\times$ 2 max-pooling resulted in 4 $\times$ 4 $\times$ 64 output. We then used a 20% dropout to reduce over-fitting and flatten the output to 1 dimension, 1024 elements. After a densely connected layer of size 512 and a rectified linear unit activation function, we again used a 20% dropout. Finally, a 2 node densely-connected output layer gave infected or healthy cell identification using a softmax activation function (i.e., $e^x/\Sigma_i e^{x_i}$).

4.3 Results

In order to train out network, we specified that 80% of the healthy and infected images serve as a training set and 20% as a test set. We chose an initial batch size of 64 to get an accurate gradient, hopefully small enough to avoid local minima. We employ the 'categorical_crossentropy' loss function and the 'Adam' first-order gradient optimizer [4]. Finally, we train the network over 10 epochs, at which point the classification accuracy of the test set decreases, an indication of over-fitting. The resulting classification accuracy of the resulting network on both the test and training sets are given in Fig. 5. Among the test set, the network quickly peaks in accuracy after 5 epochs at 95.75%, falls, and reaches the greatest accuracy (among the first 10 plotted epochs) of 95.92% at epoch 8. The

Table 1: A summary of the network detailed in Sec. 4.2.

Layer	Output Shape	Parameters (All trainable)
2D Convolutional Layer	$32 \times 32 \times 16$	208
2D Max Pooling	$16 \times 16 \times 16$	
2D Convolutional Layer	$16 \times 16 \times 32$	2080
2D Max Pooling	$8 \times 8 \times 32$	
2D Convolutional Layer	$8 \times 8 \times 64$	8256
2D Max Pooling	$4 \times 4 \times 64$	
20% Dropout	$4 \times 4 \times 64$	
Flatten	1024	
Densely Connected Layer	512	524800
20% Dropout	512	
Densely Connected Layer	2	1026

sample of misclassifications from Fig. 5 indicates that complex medical classifications can present obstacles to the network; it is easy to see how these would be misclassified given their abnormalities.

4.4 Discussion

Next, we wished to modify this initial network in order to maximize the accuracy of the network. Since the performance of the training process is variable from run to run, due in part to the training data being randomly shuffled before each epoch during fitting, it is not appropriate to compare networks unless results are reproducible. For the sake of fair comparison, we used 'PYTHONHASHSEED=0', specified a numpy seed (42) and random seed (12345), and forced TensorFlow to use single thread, since multiple threads are a potential source of non-reproducible results [5]. In this way, while the training data is randomly shuffled before each epoch, the 'random' aspects will be the same for all runs and it is appropriate to compare the results of network modifications. We confirmed this by running the original network multiple times, always getting the pattern in Fig. 5.

4.4.1 Batch Size

Since we were not certain if our batch size was too large to respond to noisy gradients indicating local minima, in Fig. 6 we explored different batch sizes used in fitting, including 4, 8, 16, 32, 64, and 128. We found that 4, while prohibitive in terms of calculation times, had the best performance. A batch size 32, however, had comparable training times and delivered superior accuracy in the test set compared to 64 both in total and at nearly every epoch.

4.4.2 Dimensionality

Next we played with the dimensionality of our convolutional layers. While in our first implementation we employed increasing filter sizes (from 16 to 32 to 64), in Fig. 7 we explore figure sizes that decrease in size (from 64 to 32 to 16), are constant at 64, and are constant at 16. Since a final output of $4 \times 4 \times 16$ has 256 total elements, we also reduced the following densely connected layer to have only 128 nodes for the constant 16 and decreasing filter size networks. We ultimately found that, while the smaller network with a constant filter size of 16 performed noticeably worse, the larger network (filter size 64) and the network that had decreasing filter sizes performed about the same as our initial configuration, with the initial configuration having the maximum achieved accuracy in the test set.

4.4.3 Dropout Rate

Dropout, randomly excluding nodes/their connections from the neural network by setting some fraction to zero, has proven benefits to reducing over-fitting [6]. However, the exact percentage used is critical as there is a 'sweet spot' to maximize performance for a given data size and networks working using dropout can even under-perform networks without dropout for smaller data sets if a given architecture and dropout rate combination is not ideal [6]. In Fig. 8, we explore dropout fractions including 10%, 20% (as in our initial configuration), 30%, 40%, and 50%, and without using dropout at all. We find that our initial dropout produced the best accuracy for the test set, and indeed outperformed the same network without dropout.

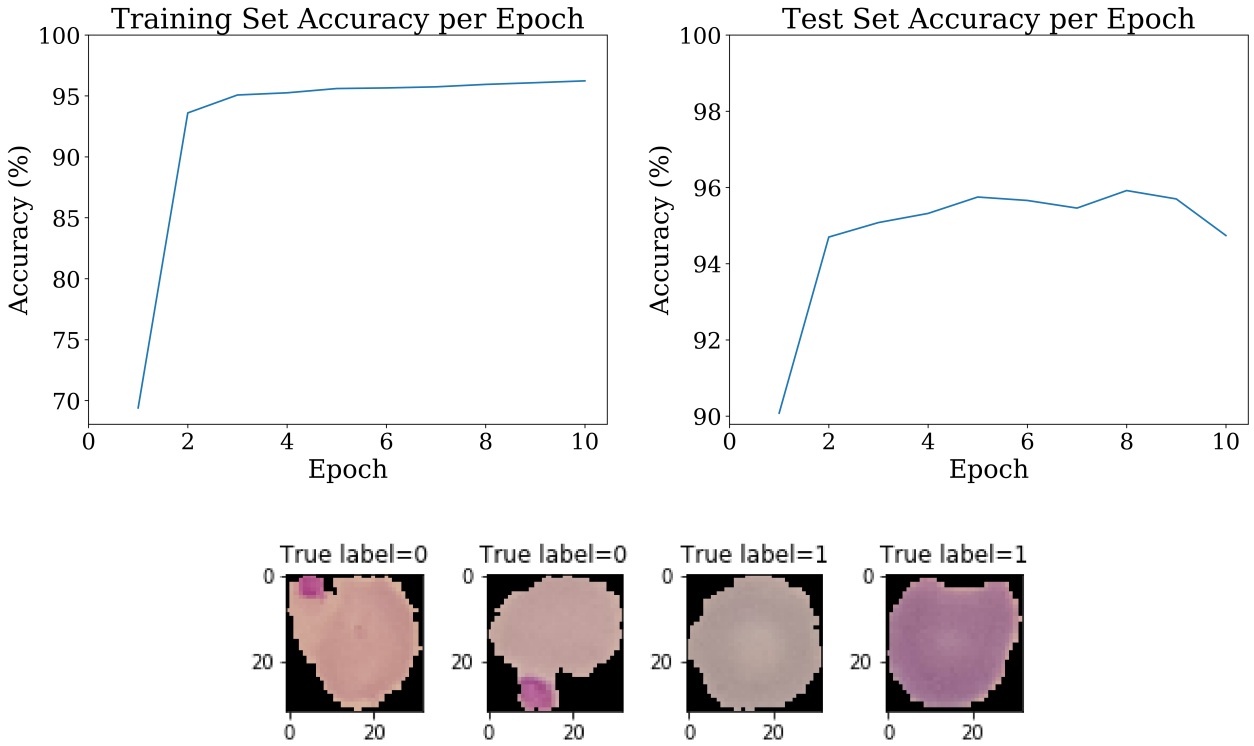

Figure 5: Percent correct classifications for the training (top left) and test (top right) data sets as a function of epoch for the 'original' neural network as described in 4.2 without any modifications and example misclassifications (bottom). The left two cells are medically healthy but were classified by the network after training as infected with malaria. The right two are infected cells but were classified as healthy by the network.

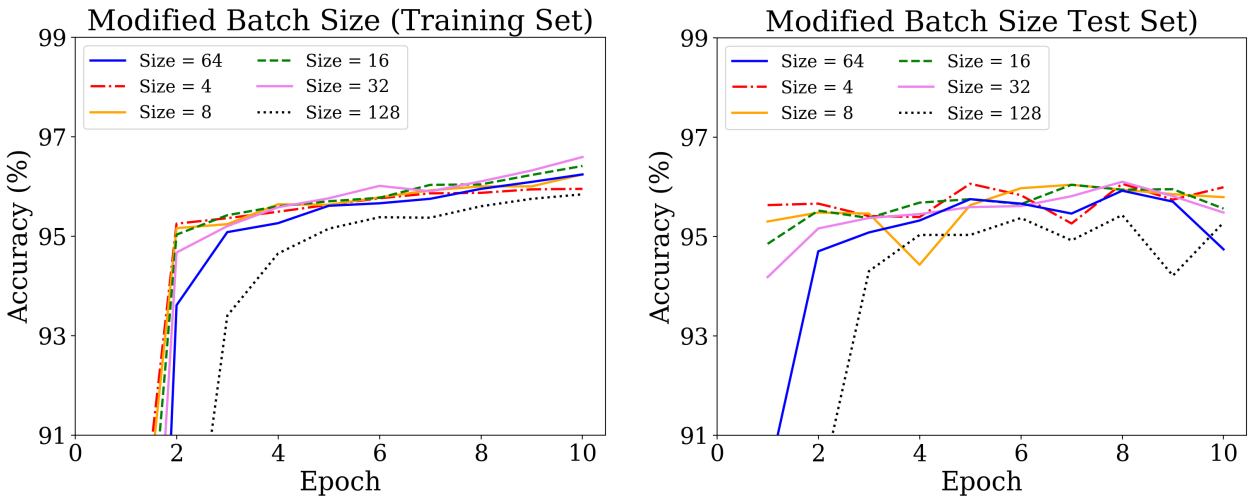

Figure 6: Percent correct classifications for the training (left) and test (right) data sets as a function of epoch for the batch sizes as given in the legend. Our initial configuration included a batch size of 64.

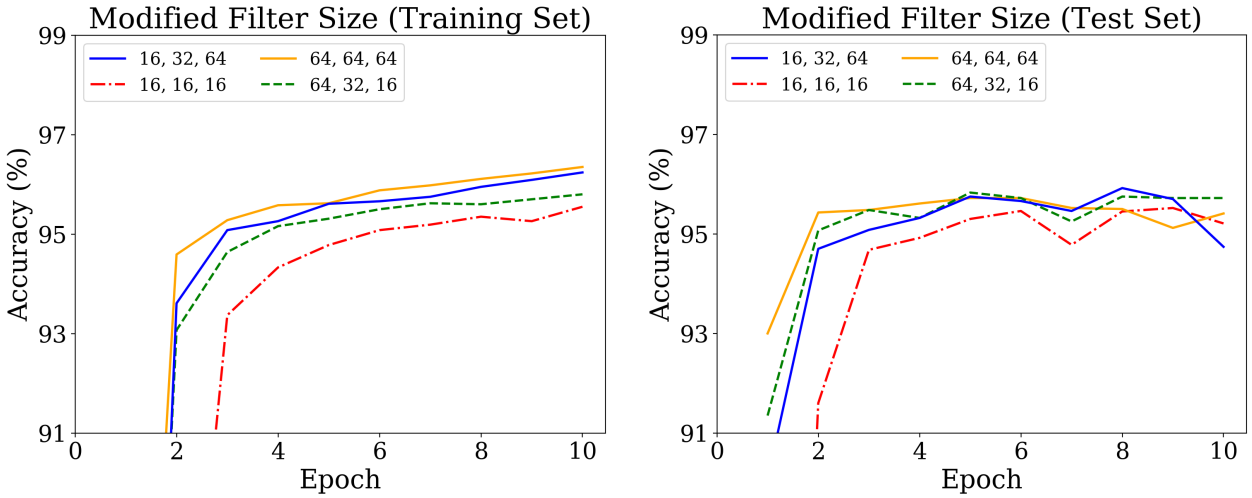

Figure 7: Percent correct classifications for the training (left) and test (right) data sets as a function of epoch for various convolutional layer dimensionalities as specified in the legend. Legend entries correspond to filter sizes of the first 32 × 32 layer, followed by the second 16 × 16 layer, followed by the final 8 × 8 convolutional layer. Our initial configuration included filter sizes of 16, 32, and 64.

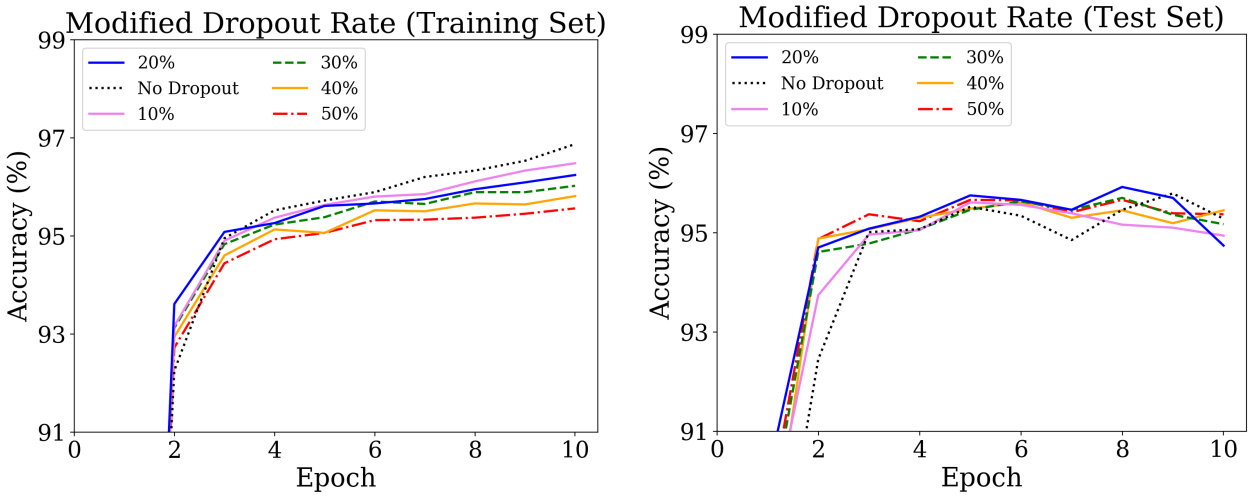

Figure 8: Percent correct classifications for the training (left) and test (right) data sets as a function of epoch for the dropout rates as given in the legend. Our initial configuration included a dropout rate of 20%.

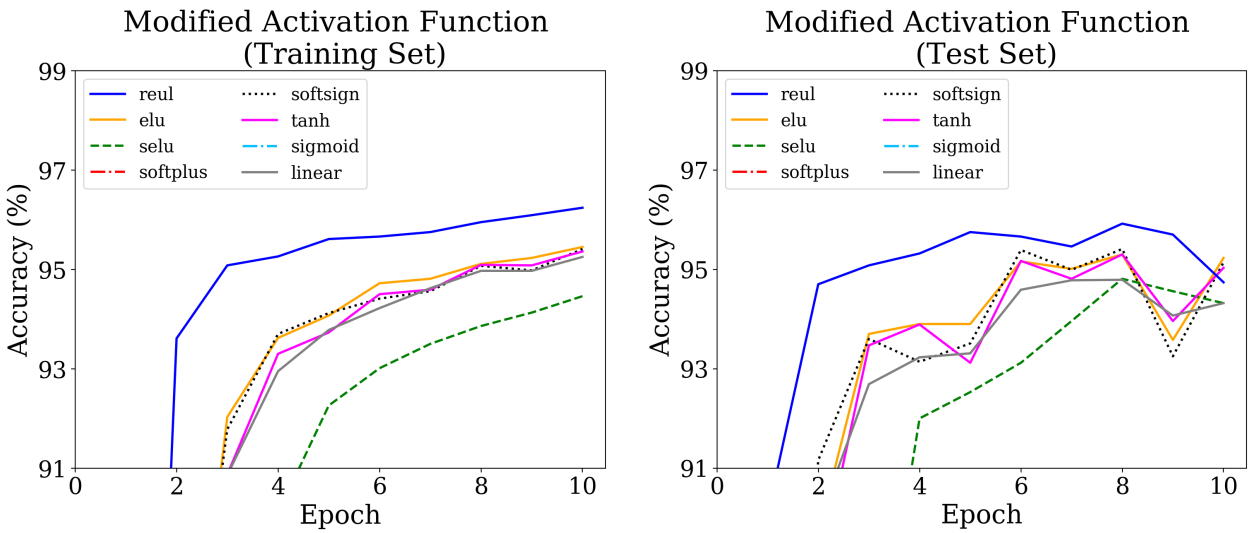

Figure 9: Percent correct classifications for the training (left) and test (right) data sets as a function of epoch for various 2D convolutional layer activation functions as given in the legend. Legend abbreviations are specified in Sec. 4.4.4. Our initial configuration included ReLU activation function for all 3 convolutional layers.

4.4.4 Activation Function

Since many of the example convolutional networks we found used rectified linear unit (ReLU, $f(x) = x$ if $x > 0$ else $f(x) = 0$)) activation functions, we started by employing it for the 3 hidden 2D convolutional layers in our network. In Fig. 9, we explore other possible activation functions, including exponential linear unit (ELU, $f(x) = x$ if $x > 0$ else $f(x) = \alpha \times (e^x - 1)$ where $\alpha = 1$), scaled exponential linear unit (SELU, $f(x) = \beta \times ELU(x, \alpha)$, where α and β are chosen so that input mean and variance are preserved [7]), softplus ($f(x) = log(e^x + 1)$), softsign ($f(x) = x/(|x| + 1)$), tanh ($f(x) = tanh(x)$), sigmoid ($f(x) = 1/(1 + e^{-x})$), and linear ($f(x) = x$) activation functions. In the end, we found ReLU vastly outperformed the other activation functions, which can be attributed to a variety of factors - the input is less condensed in ReLU compared to sigmoid, tanh, etc. (although the identity linear activation function perofrmed unusually poorly) and a vanishing gradient disturbing the training of our network would not so severely disturb ReLU. The training could just be faster, however looking out to the 20^{th} epoch ReLU continues to perform the best.

4.4.5 Convolutional Window Size

In order to explore different features for the network to examine, we tried changing the size of convolutional windows, originally all 2×2 for all 3 2D convolutional layers. In Fig. 10, we attempt uniformly larger windows, windows that increase in size through the network, and windows that decreased in size through the network. We find that convolutional windows that are 3 times bigger and that grow in size throughout the network perform worse than the original network. However, windows that are twice as large (4×4 for all 3 layers) and windows that decrease in size across the 3 convolutional layers (from 6×6 to 4×4 to 2×2) perform about as well as the original, with the decreasing window size network even having a higher maximum accuracy than our initial configuration.

4.4.6 Pooling

In order to discern between the networks with decreasing convolutional windows and our original 2×2 windows, we next we modified our pooling strategy in order to maximize the performance gain from changing feature sizes. Since we were looking at larger features, it is possible to represent them with reduced data sizes, which might reduce over-fitting and increase performance. By pooling data from larger sized regions we might achieve this. In addition, rather than taking the maximum value from a pooling window, we can also take the average value. In Fig. 11 we explore average pooling as well as doubling the size of the first pooling window to 4×4 both for networks with decreasing convolutional windows and our initial configuration. Unfortunately, we find none of these pooling strategies improves performance beyond 2×2 max pooling.

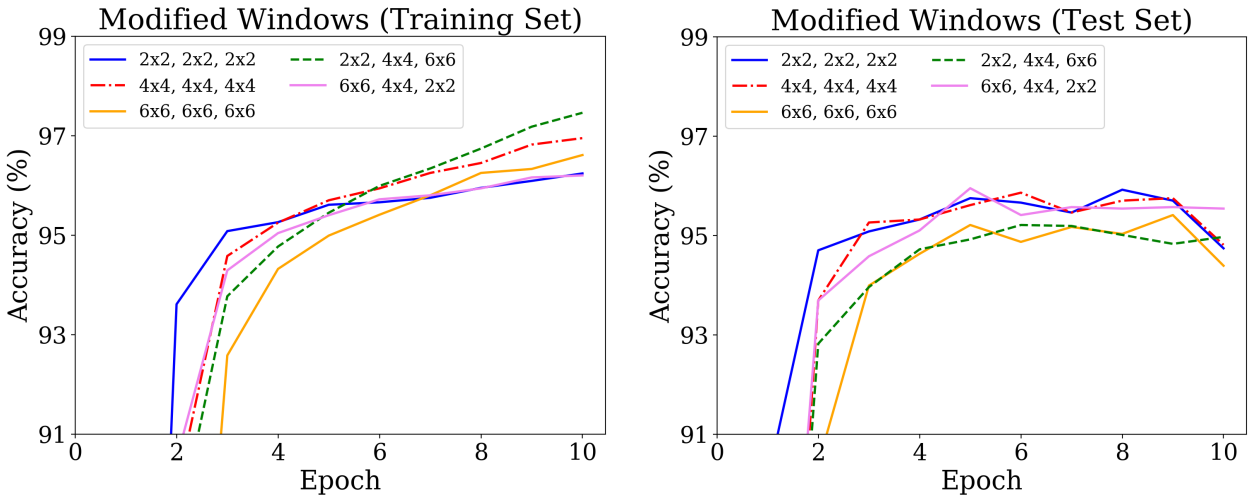

Figure 10: Percent correct classifications for the training (left) and test (right) data sets as a function of epoch for various convolutional windows as specified in the legend. Legend entries correspond to window sizes of the first 32 × 32 layer, followed by the second 16 × 16 layer, followed by the final 8 × 8 convolutional layer. Our initial configuration included window sizes of 2 × 2 for every layer.

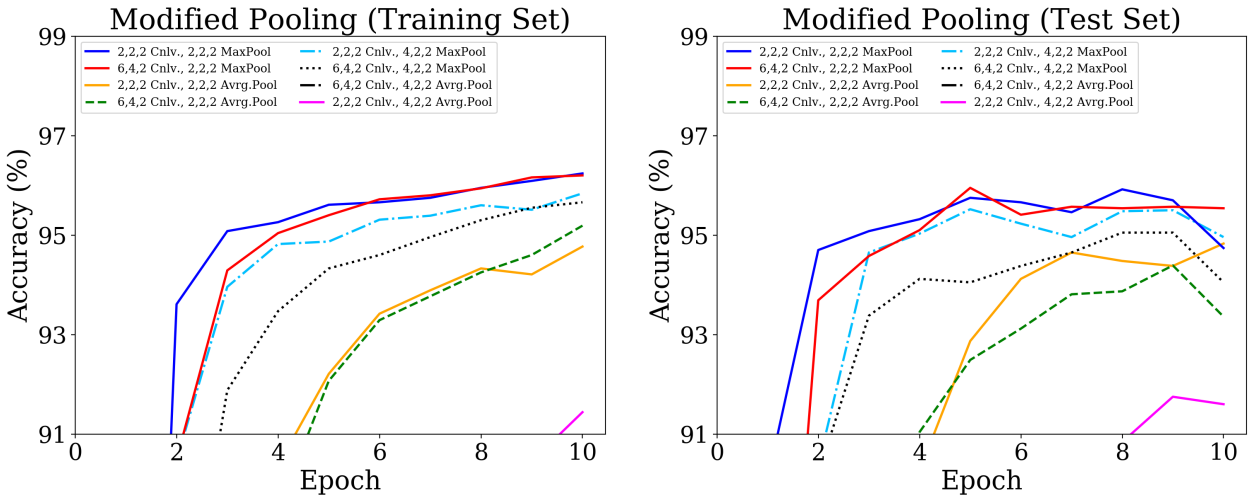

Figure 11: Percent correct classifications for the training (left) and test (right) data sets as a function of epoch for various pooling strategies and convolutional windows as specified in the legend. Legend entries correspond to both window sizes of the 32 × 32, 16 × 16, and 8 × 8 convolutional layers, and pooling sizes used after each layer. Values are given as one side of the square convolutional and pooling windows. 'Max' or 'Avrg' indicate whether the maximum of the average value of the pooled region was used. Our initial configuration included convolutional window sizes of 2 × 2 for every layer and took the maximum value of 2 × 2 windows during pooling after each convolutional layer.

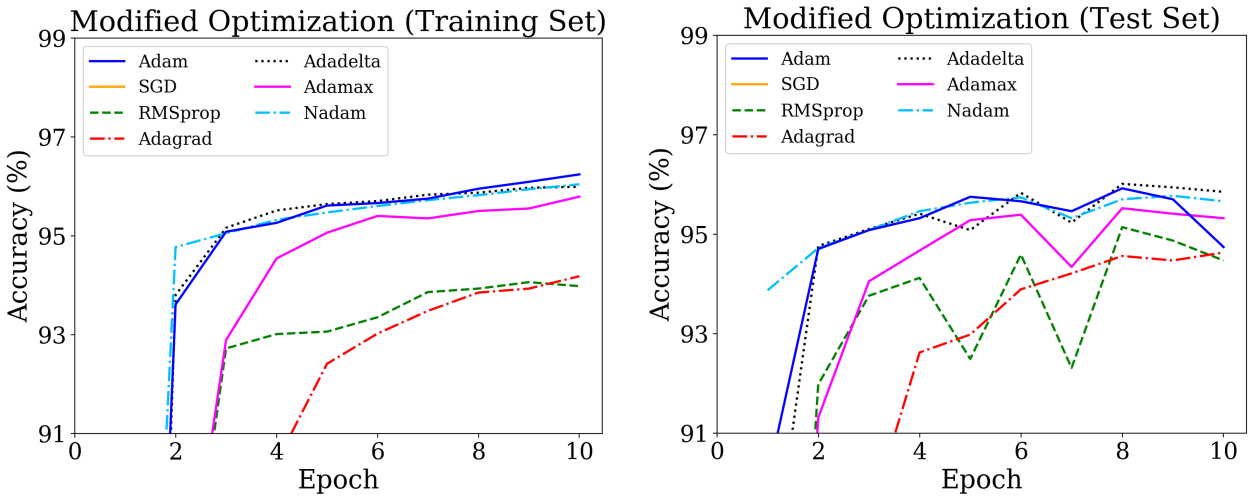

Figure 12: Percent correct classifications for the training (left) and test (right) data sets as a function of epoch for various optimization strategies as described in Sec. 4.4.7. Our initial configuration used 'Adam'.

4.4.7 Optimization

Finally, we attempted modifying our fitting strategy itself. Initially we used 'Adam', **ADA***ptive* **M***oment estimation* [4], which, as discussed in class, computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients. However, other potential modified gradient descent algorithms include:

- 'SGD', stochastic gradient descent as discussed in class updating θ in mini-batches
- 'AdaGrad', as discussed in class with each parameter's learning rate normalized by the RMS of accumulated gradients
- 'RMSProp', as discussed in class normalizing each gradient by a moving average of squared gradients
- 'Adadelta', as discussed in class, including a $\Delta\theta$ term in 'RMSprop'
- 'Adamax', a variant of Adam based on the infinity norm [4]
- 'Nadam', the Nesterov accelerated adaptive moment estimation

In Fig. 12 we explore each of these (using Keras defaults when applicable). We found that Adadelta and Nadam do, in fact, achieve slightly higher performance than Adam, with higher maximum accuracy and higher initial peak accuracy.

References

[1] The World Health Organization. 2016, World malaria report.

[2] Poostchi, M. et al. 2018, Image analysis and machine learning for detecting malaria. Translational Research 194:36–55.

[3] Rajaraman, S. et al. 2018, Pre-trained convolutional neural networks as feature extractors toward improved malaria parasite detection in thin blood smear images. PeerJ 6:e4568.

[4] Kingma, D. et al. 2015, Adam: A Method for Stochastic Optimization. ICLR 2015.

[5] Chollet, F. et al. 2015, How can I obtain reproducible results using Keras during development? Keras FAQ.

[6] Srivastava, N. et al. 2014, Dropout: A Simple Way to Prevent Neural Networks from Overfitting. JMLR 15:1929-1958.

[7] Klambauer, G. et al. 2017, Self-Normalizing Neural Networks. NIPS 2017.