Michael Wisnewski

Professor Mathews Jacob

Machine Learning

December 10th, 2019

Machine Learning Final Project

1. **Baseline Network**

I created a baseline network to operate as a reference point for all other experiments. This model contains 3 convolutional layers, 2 pooling layers, and 2 fully connected layers. After employing all techniques designed to increase model performance, this simple model achieved a training accuracy of 100% and a test accuracy of about 99%. It is fair to say that MNIST is completely solved by this model. The experimentation here helped me develop a valuable CIFAR classifier.

1. **Model Exploration**
   1. **Optimizer**
      1. **Adam**
         * **Learning Rate = 0.00001**

**A picture containing sky, window, photo

Description automatically generated**

* + - * **Learning Rate = 0.0001**

**A picture containing window, photo

Description automatically generated**

* + - * **Learning Rate = 0.001**

**A picture containing sky, window, photo

Description automatically generated**

* + - * **Learning Rate = 0.01**

**A picture containing window, sky, photo, showing

Description automatically generated**

* + - * **Learning Rate = 0.1**

**A picture containing window, sky

Description automatically generated**

* + 1. **SGD**
       - **Learning Rate = 0.00001**

**A picture containing sky, different, photo, window

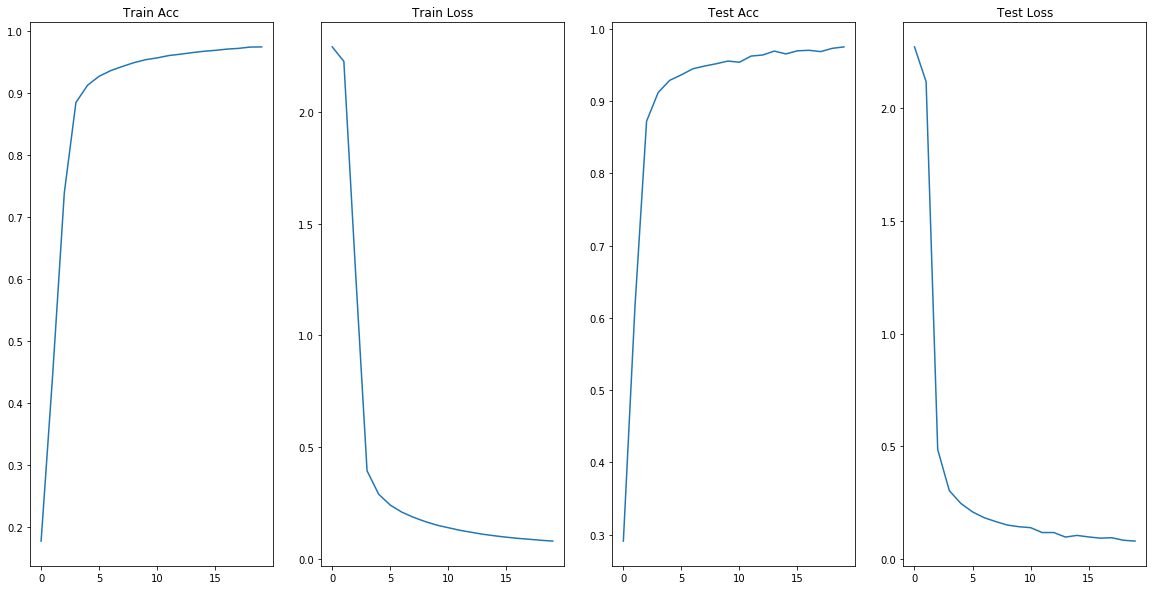
Description automatically generated**

* + - * **Learning Rate = 0.0001**

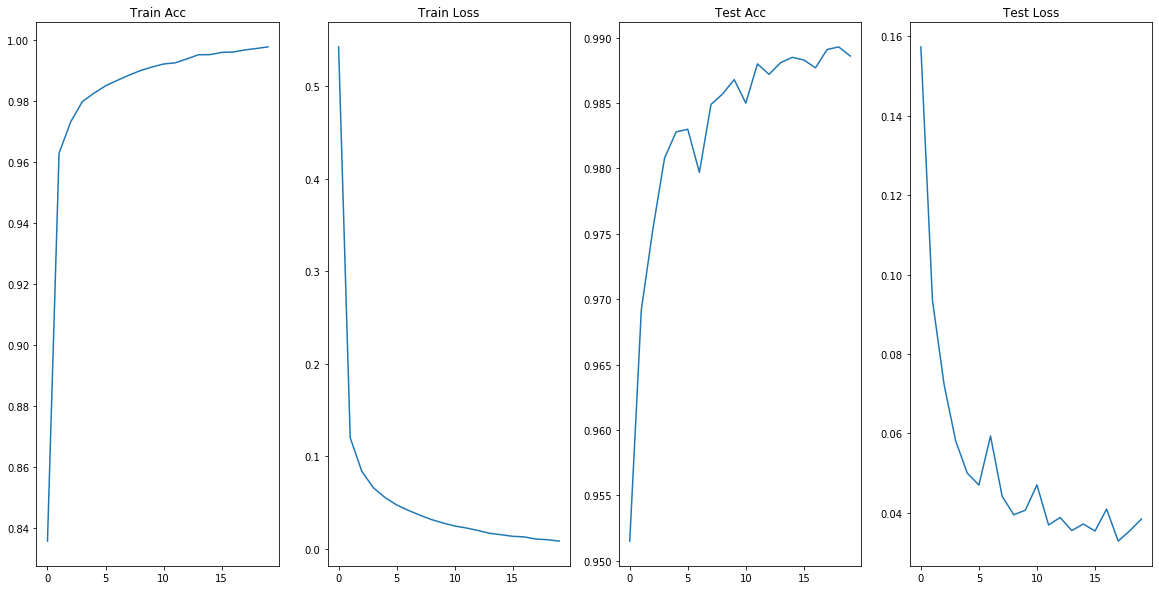
**A picture containing sky, different, photo, object

Description automatically generated**

* + - * **Learning Rate = 0.001**

****

* + - * **Learning Rate = 0.01**

****

* + - * **Learning Rate = 0.1**

**A picture containing window

Description automatically generated**

* + 1. **Observations**
       - The adam optimizer packages many of the techniques/tricks that enable excellent learning in deep neural networks. This includes SGD, RMSProp, Adagrad, momentum, etc. It is the most effective optimizer without a doubt.
       - SGD performs surprisingly well at astoundingly high learning rates. While training my MNIST classifier, I actually found SGD with learningRate = .1 was even more effective than Adam. I am unsure of why this is.
  1. **Activation Function**
     1. **Sigmoid**

**A picture containing window, photo

Description automatically generated**

* + 1. **TanH**

**A picture containing window, photo, showing

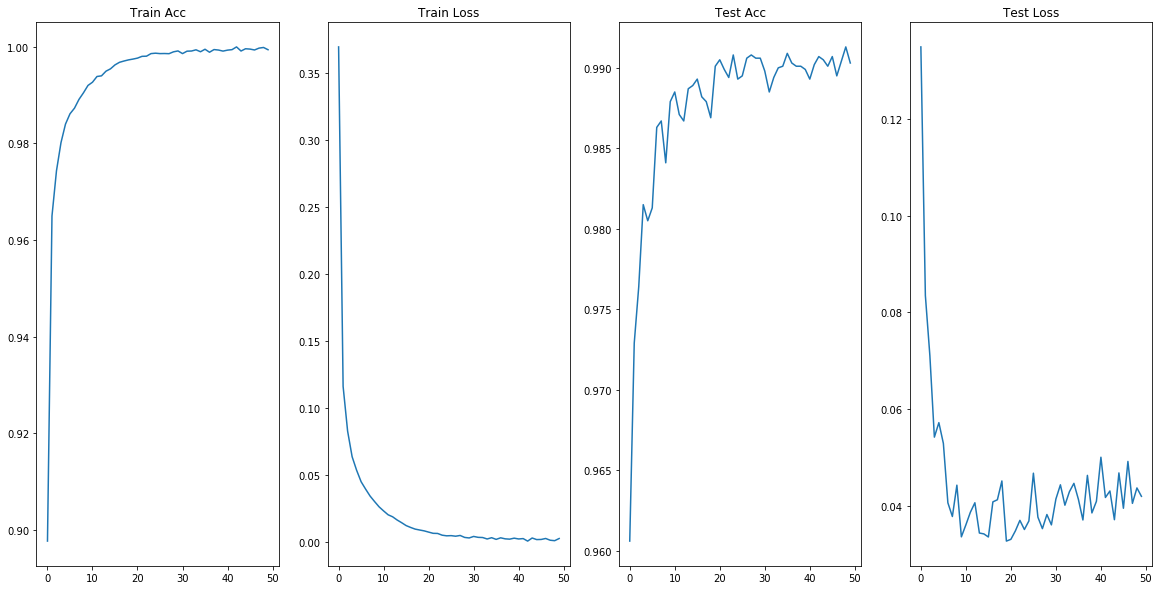
Description automatically generated**

* + 1. **Relu**

**A picture containing window, photo

Description automatically generated**

* + 1. **Observations**
       - Sigmoid is very clearly unfit for training complex models. It does not have the correct properties to serve as a quality activation function.
       - TanH performed surprisingly well. It behaves similarly to sigmoid but solves many of the problems it has with vanishing/exploding gradients.
       - Relu performs the best. This activation function has stood the test of time and works in nearly every scenario. It should be the go-to when it is unclear which should be chosen.
  1. **Early Stopping**
     1. **Strategy**
        + Save model after epoch, only when it outperforms the current best saved model. Does not actually ‘stop’ because models occasionally get worse in order to get better.
     2. **Plots**

****

* + 1. **Observations**

When the model approaches its maximum generalizable capacity, it begins to overfit to the training data. This causes validation accuracy and validation loss to slowly and steadily rise despite training loss decreasing. This approach saves the model when it has minimized validation loss and maximized validation accuracy.

By using an early stopping callback, training will actually halt if the validation loss ever increases. This is undesirable because models are not going to forever improve at all times. If a local minimum is found, it necessarily must increase loss in order to move toward a better, global minimum. By not halting training, the model can continue to improve even if it has seemingly gotten worse.

* 1. **Data Augmentation**
     1. **Strategy**
* Use Keras’ built in ImageDataGenerator function. This generates a significantly larger dataset by cropping, zooming, and rotating the default training data.
  + 1. **Plots**

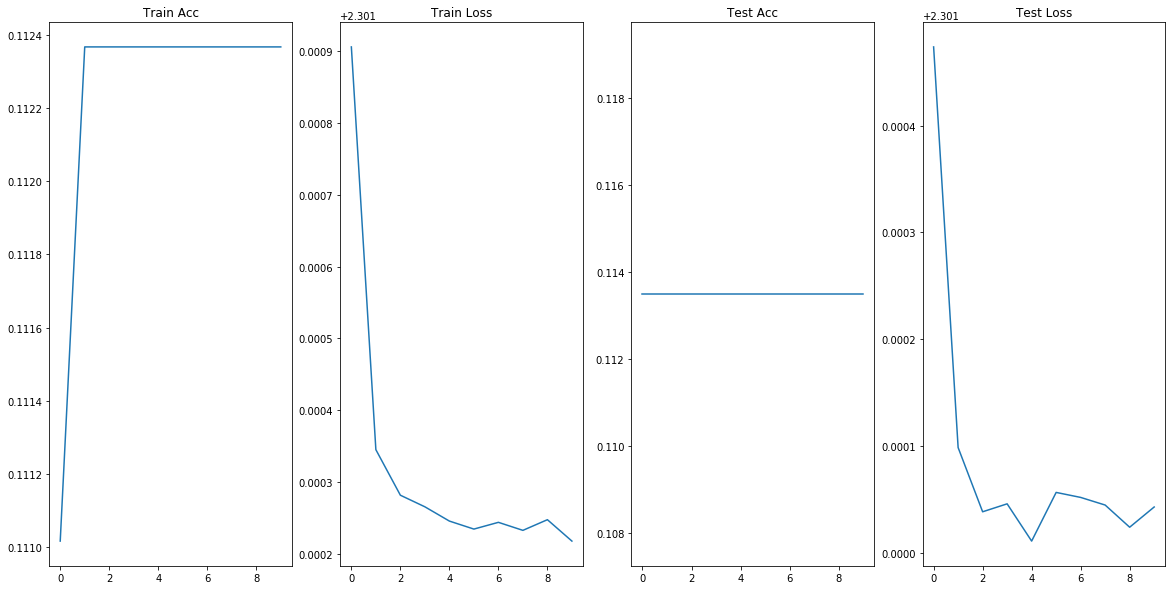
**A close up of a window

Description automatically generated**

* + 1. **Observations**

Surprisingly, this had a quite negative impact on the model’s generalizability. The model quickly decayed after an accuracy of roughly .85 was reached. I believe this is due to the MNIST dataset being extraordinarily simple. It is considered a trivial problem at this point and using such excessive computational force on the problem is likely just not effective. This approach is likely invaluable for more difficult, real-world problems.

* 1. **Network Architecture**
     1. **Deep**

****

* + 1. **Wide**

**A picture containing sky, photo, window, showing

Description automatically generated**

* + 1. **Observations**

Wide layers increase the number of trainable parameters much faster than adding several smaller layers. It was difficult to make a network deep enough to match the number of parameters as a wide two-layer network.

Deep networks take a long time to train, even with fewer parameters. This is likely because the gradient calculation is being done with greater frequency, increasing computational complexity.

The wider network is by no means fast but was about the same speed as the deep network with 1 million more parameters.

The Universal Function Approximator theorem states that a single layer of infinite width is capable of approximating any function to arbitrary closeness. This makes me think that wide layers might be more effective than many narrow layers.

It is likely best to find a network that utilizes the benefits of both depth and width.

1. **CIFAR10 Classification Model**
   1. **Development**

While building the MNIST model, I had ample opportunities to heavily experiment with network design and discover what types of CNN architectures worked best. I have previously taken a Deep Learning class with Professor Stephen Baek, where we learned about VGG blocks in CNNs. One VGG block consists of a convolution-convolution-maxpool layer sequence. These are typically repeated numerous times. I decided to use these to build my model.

In this other Deep Learning class, we also discussed weight regularization and the importance of weight initialization. For this reason, I initialize all layers using the he-uniform initializer. This initializes all weights using a truncated normal distribution that is directly determined by the number of inputs to the layer. I also use the L2 norm to regularize the weights at each convolutional layer. This prevents weights from growing too large.

At the end of each VGG block, I add a Dropout layer that drops neurons at a rate of .2. This slows the time until the model begins to overfit, allowing for more real training time. After implementing this, I noticed that training accuracy and test accuracy increase in lockstep until about .77 accuracy when test accuracy begins to slow.

Each VGG block also feeds into a Batch Normalization layer before continuing to the next block. This regularizes the data, preventing shifting of the internal covariance. This should increase convergence time by keeping the data centered nicely in the most effective range of the activation functions.

After four VGG blocks with varying qualities, the data flows into the final stage. Two dense layers of size 128 and 64 using the relu activation function perform the classification. These feed into a softmax layer that generates a distribution at the output.

* 1. **Diagram**

**A screenshot of a cell phone

Description automatically generated**

* 1. **Training**

The model was trained using an Adam optimizer and a learning rate scheduler. While exploring deep learning hyper parameters with the MNIST dataset, I concluded that Adam is the most generally capable optimizer. Adam outperformed SGD most of the time, and I am unsure of the validity of the few instances where SGD bested Adam. The learning rate scheduler allows the learning rate to be adjusted during training. This is crucial as it allows the model to slow the learning rate as it reaches the minimum. The schedule is shown below.

|  |  |
| --- | --- |
| Epoch | Learning Rate |
| epoch < 15 | .0001 |
| 15 <= epoch < 35 | .00001 |
| 35 <= epoch < 65 | .000001 |
| epoch >= 65 | .0000001 |

* 1. **Results**

This model converged to a validation accuracy of .86 after 77 epochs. I am very happy with this accuracy because it achieves only a four percent lower accuracy than the most effective model that strictly uses CNNs. This is a very fast convergence rate, and the model only has 14 million trainable parameters. This is relatively small considering the immense scale of modern architectures. For example, VGG 16 has 138 million trainable parameters.

I think by continuing to poke and prod my model I could easily improve by a percentage point or two. One avenue I wish to explore is the idea of scaling up the images in the preprocessing phase. Having larger images allows for larger convolutional kernels and max pooling layers. Larger kernel sizes let the layer detect larger patterns. Max pooling layers reduce the activation map size by some factor (2 in my case), which institutes a hard limit on the number of VGG blocks I can add. If you shrink the activation map too much it will become meaningless or will even be of zero size. I could not try this approach for this project because the Google Collab runtime simply did not have enough ram to support the scaled dataset. I believe this could be very effective and will certainly try it in the future.