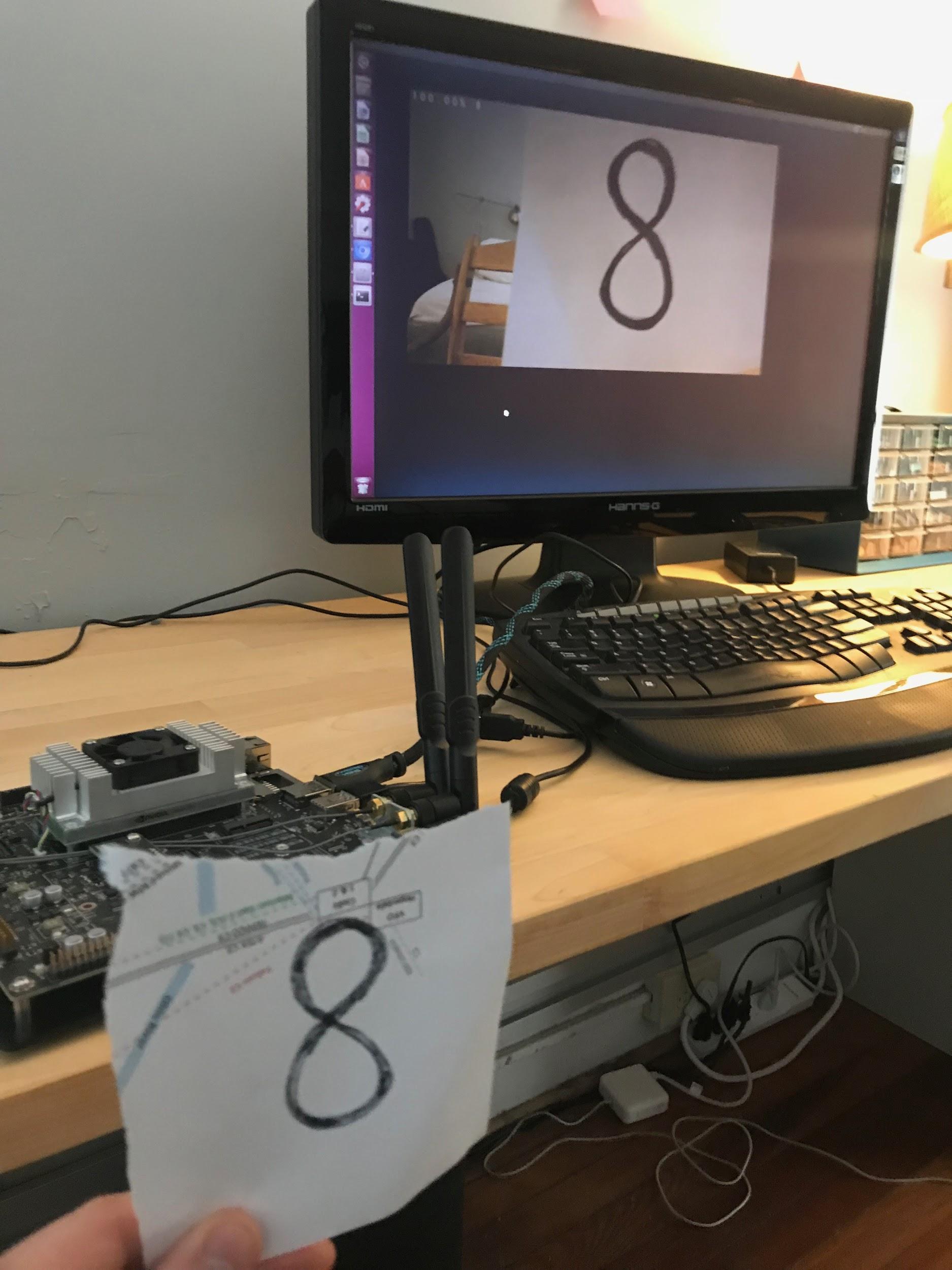
# Deep Learning Analytics Interview Challenge

By: Zachary Clement

1/6/18



1.0 Quick Start

2.0 Background

3.0 Convolutional Neural Networks

4.0 Problem Formulation

5.0 Tools, Models, Data, and Design Choices

6.0 Results

7.0 Project Process Implementation

References

Appendix A: LeNet Caffe Code

Appendix B: Visualization

## 

## 1.0 Quick Start

To get the image classification program running:

1. Turn on Jetson TX1
2. Open Terminal and navigate to Document/jetson-inference/build/aarch64/bin
3. In Terminal, type:

NET=LeNet

1. In Terminal, type:

./image-camera \

--prototxt=$NET/deploy.prototxt \

--model=$NET/snapshot\_iter\_22500.caffemodel \

--labels=$NET/labels.txt \

--input\_blob=data \

--output\_blob=softmax

A camera image should start on the screen. Write down a number on a piece of white paper and hold it up to the camera (note: use a sharpie. Make sure the lines are thick). The TX1 will tell you what number you are holding up along with its confidence in the classification.

## 2.0 Background

Deep learning is pushing the frontier in areas such computer vision, speech recognition, natural language processing, and even structured data analysis. The use of deep learning has exploded in recent years due to a confluence of factors, including advances in graphics processing units (GPUs) utilization, deep learning algorithms, and available software tools. Over the years, a number of significant breakthroughs, some of which are discussed in this paper, have enabled these recent advances. Image classification - a subset of computer vision - is one of areas where performance and accessibility have significantly improved. Convolutional neural networks (CNNs) have proven to significantly outperform other image classification algorithms.

New hardware and software tools make implementing neural networks easier than ever before; these tools are designed to push computation to specialized GPUs and optimize their performance. Two examples of hardware are AWS Nvidia machines (host PC) and the Nvidia Jetson TX1 (distributed). In addition to hardware improvements, freely available software (i.e. cuDNN, Caffe, and DIGITS) make implementing standard image classification relatively simple. These tools often come with trained versions of classic models and provide easy access to classic data sets. Additionally, many pretrained models are openly available on Github.

The histories of computer vision architectures and data sets are rich. Some of the more famous architectures include LeNet, AlexNet, and GoogleNet. Along with commonly cited architectures, there are a number of commonly referenced data sets for image classification: MNIST and ImageNet to name a couple. MNIST is a data set of seventy thousand images of handwritten numbers 0 through 9. ImageNet is a data set of over ten million images. These datasets are often used for benchmarking network architecture performance. Many popular architectures and datasets are freely available. Recent breakthroughs in deep learning such as ResNets and Inception layers are built on the foundation of pioneering work. it is important to acknowledge the early contributors.

## 3.0 Convolutional Neural Networks

Today, image classification is almost exclusively performed using CNNs, which are considered the current state of the art in computer vision. As previously mentioned, CNNs come is a wide variety of architectures, each presenting a variety of trade-offs - most importantly between performance and computational expense. Much of the active research is to increase performance while reducing computational costs. CNNs work by applying filters to images that detect edges and features. Rather than dictating the filters, the parameters of the filters are learned through gradient descent. The two main advantages of using CNNs over simple fully connected layers are parameter sharing and sparsity of connection. CNNs, and deep neural networks in general, are considered data hungry with performance generally improving as more data is presented to the network. In the absence of more data, often hand engineering the networks and tuning model hyperparameters can improve CNN performance.

CNNs usually contain three types of layers: convolutional layers, pooling layers, and fully connected layers. Along with the different types of layers, there are a number of hyperparameters which are usually specific to the type of layer.

*convolutional layers (conv)*

Convolutional layers are the workhorse of CNNs. The convolutional layer applies a filter to the previous layer. The filters are used to detect edges and features in the image. Hyperparameters to choose include filters size, padding, and stride. The larger the stride, the more an image shrinks. Padding solves issues of shrinking images and throwing way info from the edges. The filters are made up of parameters that are learned through gradient descent.

*pooling layer (pool)*

Pooling layers makes feature detection more robust, but shrink the height and width of the image. Today, the most common type of pooling layer is a max pooling layer. Average pooling is another type of pooling layer that is used much less today. Filter Size and stride are the hyperparameters to choose in a pooling layer. There are no parameters to learn. Pooling layers are often combined with convolutional layers in series.

*fully connected layer (fc)*

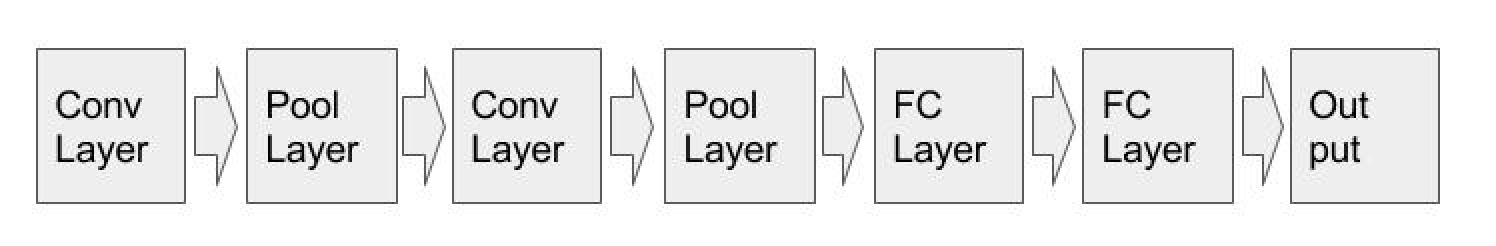
Fully connected layers connect every neuron in one layer to every neuron in another layer. Often, the output from a conv layer will be the parameters fed into the fully connected layer. The hyperparameters to set include number of fc layers, number of units in each layer, number of output parameters, solver type, and activation function.

Along with the various layer types, there are a wide range of hyperparameters to consider. Adjusting hyperparameters is meant to either improve performance (reduce overfitting, reduce loss, etc), reduce computation expense, or both. With such as large number of parameters to choose, tuning a model can represent a large portion of the work of optimizing a neural network. We can look to existing architectures for design ideas. Here we will examine two interesting architectures that demonstrate the evolution of CNNs overtime - the LeNet and GoogleNet architectures.

**The LeNet[[1]](#footnote-0)**

Originally published in 1998, the LeNet Model was designed to recognize handwritten digits. This network has roughly 60,000 parameters, which is comparatively small to modern architectures which can have 100 million parameters. The original model was trained on grayscale images. There is no padding, so each time a convolutional layer is applied the image shrinks. Height and width of images tends to go down, while the number of channels go up. The original model used sigmoid and tanh nonlinearity function; however, the model implemented in this project uses RELU (as seen in appendix A caffe code).

The LeNet paper was groundbreaking because it Introduced the idea of chaining a series of convolutional and pooling layers together before a series of fully connected layers. AlexNet - another famous architecture published in 2012, is essentially a more modern implementation of LeNet that uses more layers and GPU optimization.



**Figure 1. General LeNet Architecture**

**GoogleNet[[2]](#footnote-1)**

The original paper was published in 2014. GoogleNets major advance was applying inception layers. Inception layers uses various filters, padding, and max pooling, doing each convolution in parallel and then concatenate the output of each. The trade off is the computational costs are very high. One trick used to control computation is the use of of 1x1 convolutions (aka bottleneck layers) to reduce dimensionality. These save on computation and enable much deeper networks. As we can see from the visualization in Appendix B, the GoogleNet architecture is significantly more complex than the LeNet architecture.

## 4.0 Problem Formulation

This project focuses on image classification. The goal of this project is to demonstrate a suite of data science skills by implementing a convolutional neural network (CNN); the steps for execution are summarized in section 7.0. The project is documented so that it can be easily reproduced as well as to show understanding of the fundamental concepts.

The goal of this project is to demonstrate manifold data science skills (Data science problem formulation, devops, data wrangling, analysis, and communication) with the end product being a complete deep learning system on the Jetson TX1. The submitted TX1 will be able to identify handwritten numbers, 0 - 9, using its built in camera. The project will use MNIST data to train multiple models. The best performing model will then be deployed to the TX1. In order to demonstrate a more typical workflow, the project will use a AWS server as the host PC.

This project will:

1. Get TX1 running with minimum viable product (GoogleNet trained with ImageNet)
2. Download MNIST data onto host PC
3. Train and Test various models with the MNIST data
4. Analyze various models performance
5. Deploy best performing model to TX1
6. Demonstrate TensorRT inference of handwritten numbers on Jetson TX1
7. Submit a report with background info and process details

Note: A complete deep learning systems (GoogleNet train ImageNet) is working after step 1. This represents the state of the art of 2014 for basic image classification. With this in mind, this project does not attempt to implement a more advanced network on a larger dataset (i.e. transfer learning, data augmentation, ResNets); rather, this project attempts to demonstrate a richer set of data science skills and basic knowledge of underlying concepts.

The project is documented so someone with limited computer skills could replicate the entire project with minimal effort. Additionally, descriptions are given of hardware and software tools and techniques to help the reader understand the various components and how they fit together. This project embodies the philosophy to not reinventing the wheel; it utilizes a significant amount of others work and is built on top of that work. References are provided for all sources.

## 5.0 Tools, Models, Data, and Design Choices

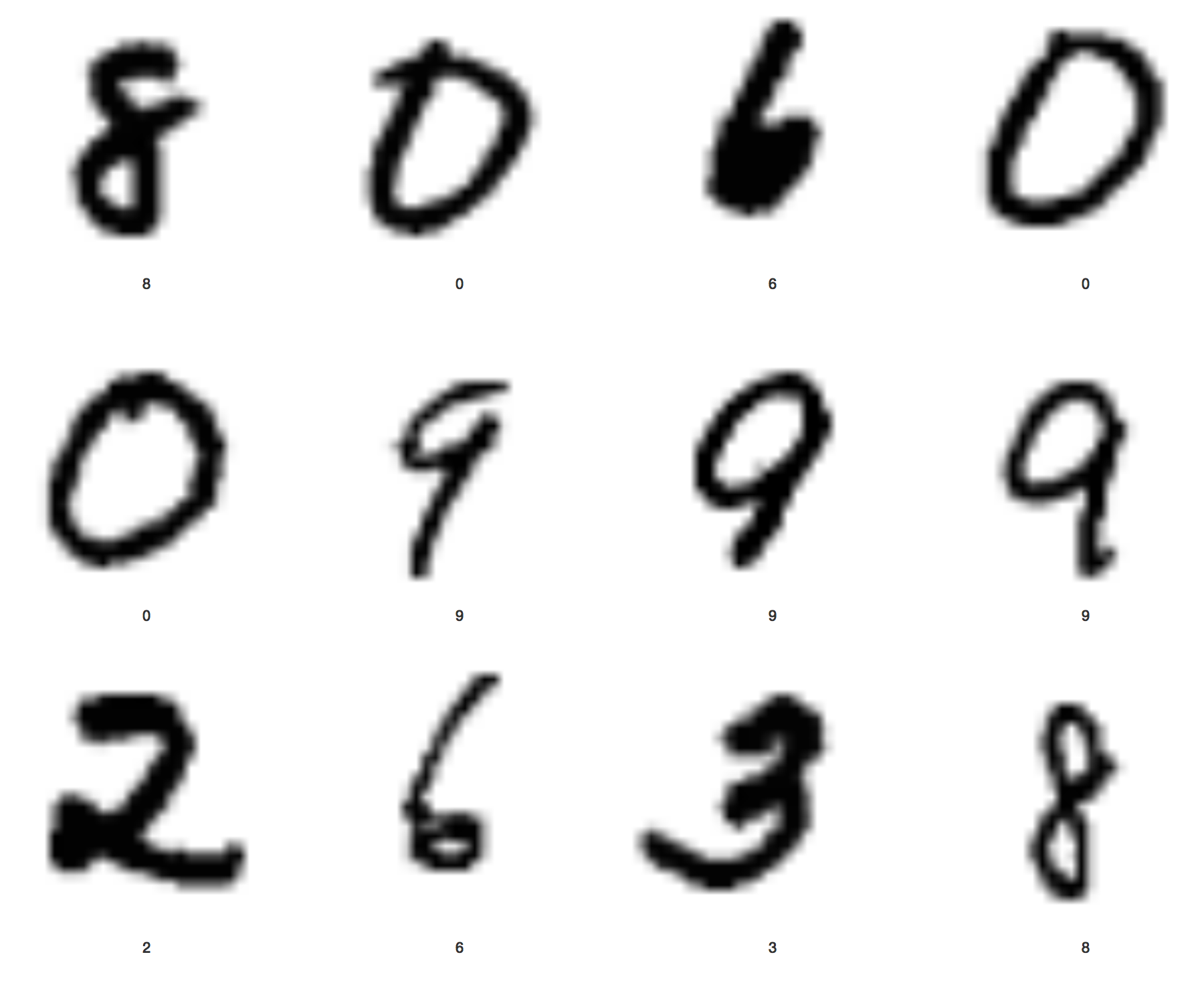
The project uses specific tools, models, and data. Additionally, specific design choices were made throughout the project. This section discusses the general process for implementing an image classification system as well as the specific choice made to implement this project.

The primary tools used for this project include:

* Jetson TX1
* AWS server running ubuntu 16.04
* CUDA toolkit, Caffe, DIGITS, and TensorRT
* Various public git repos

There is a general process for image classification and applied machine learning more generally: 1) generate an idea, 2) code it up, 3) perform experiment, 4) repeat. The process is highly iterative. It is common to start with relatively simple implementations and only increase complexity as needed to meet performance standards. Further, the project should progress in a systematic way often referred to as orthogonalization.

This project uses the MNIST data set. Setting up the data is an important consideration. The MNIST data set includes 60,000 training and 10,000 test images. I set aside 20%, or 12,000 images, from the training set for the dev set. Figure 2 shows an example of twelve different input images of the MNIST dataset.



**Figure 2. Example of MNIST data set.**

For this project, I chose to test two models - LeNet and GoogleNet - and then compare their performance against each other on the MNIST data. I used DIGITS to train and test the models. DIGITS is a web based, open source software program. DIGITS simplifies common deep learning tasks such as managing data, designing and training neural networks on multi-GPU systems, monitoring performance in real time with advanced visualizations, and selecting the best performing model[[3]](#footnote-2). DIGITS comes with both LeNet and GoogleNet standard models written with Caffe (see appendix A for caffe code).

Because I am starting with two standard models, a large number of hyperparameters are already selected for me; however, the important hyperparameters that I can set and their initial settings are:

* Image size: 28x28 (LeNet) and 256x256 (GoogleNet)
* Image type: grayscale
* Solver: Stoicastic gradient decent
* Base learning rate: 01
* Batch size: 1
* Number of epochs: 30
* Subtract mean: image

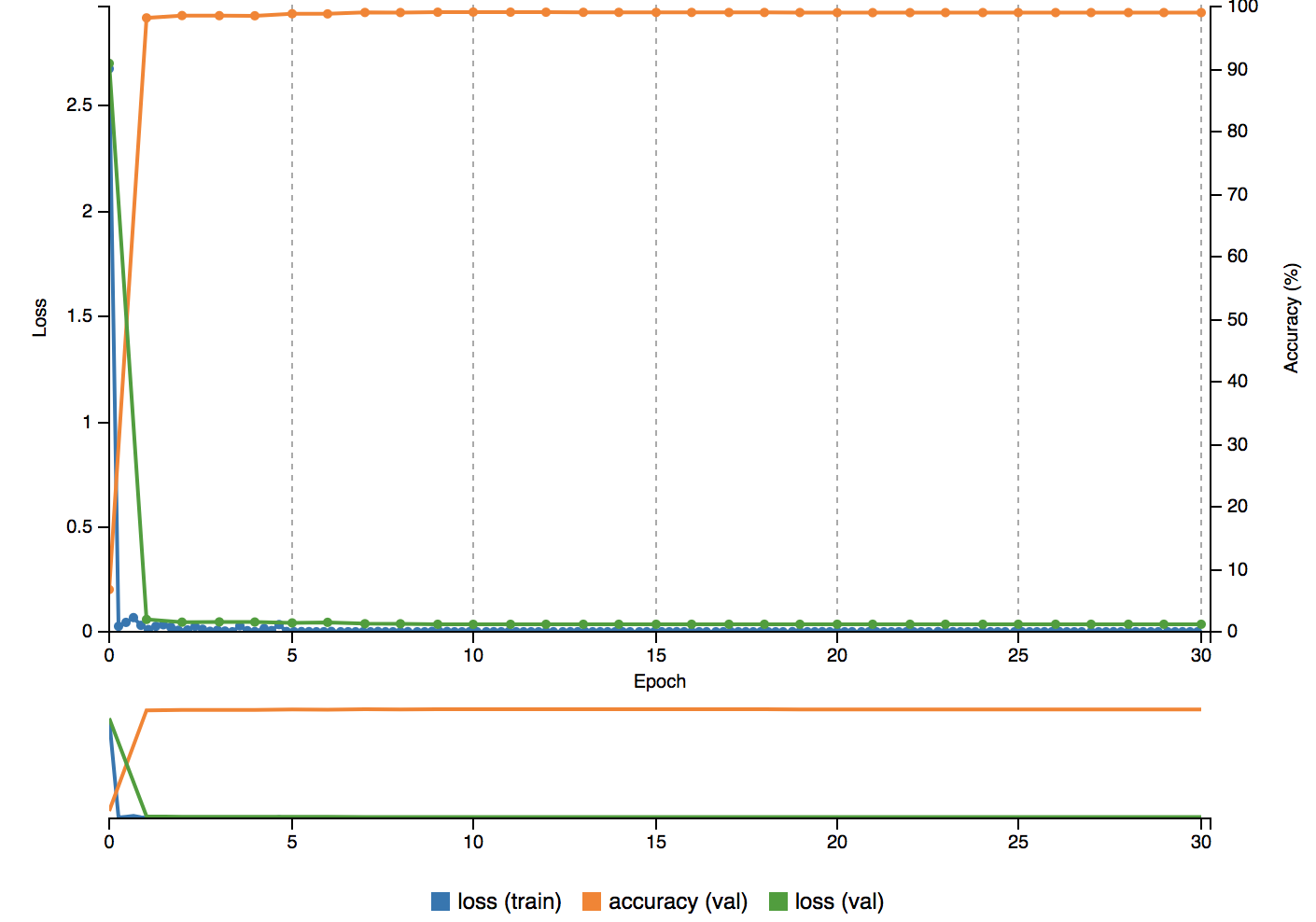
## 6.0 Results

This section discusses the results from training and testing the LeNet and GoogleNet networks. The primary methods for discussing results are: 1) changes in loss and accuracy over time; 2) overfitting; 3) learning rate; 4) overall run time necessary to train the model. After discussing each network individually, the models are compared and conclusions are discussed. Note: Overfitting would show up as high accuracy on the training and dev sets, but low accuracy on the test set.

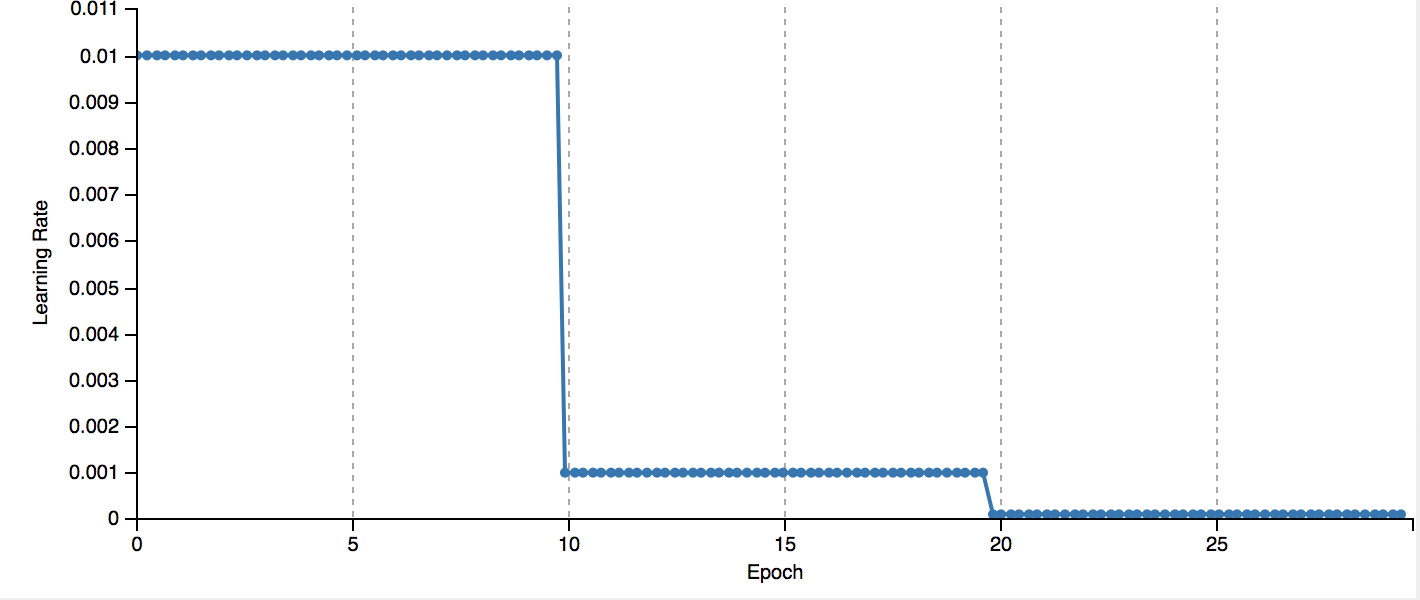
**LeNet Results**

The LeNet model proved to have excellent performance, demonstrated no overfitting, and finished training in less than 5 minutes. Figure 3 below shows the loss in the model quickly drops to near zero while the accuracy quickly jumps to near 100%. Figure 4 shows the training rate throughout the model run. At around the 10th epoch, the learning rate dropped dramatically and by the 20th epoch, the learning rate had dropped to near zero. We see no evidence of overfitting with the LeNet model on the MNIST data. Because of the simplicity of the data set and the significant amount of data, the model performed very well with the initial hyperparameters; therefore, no significant tuning was necessary.

.



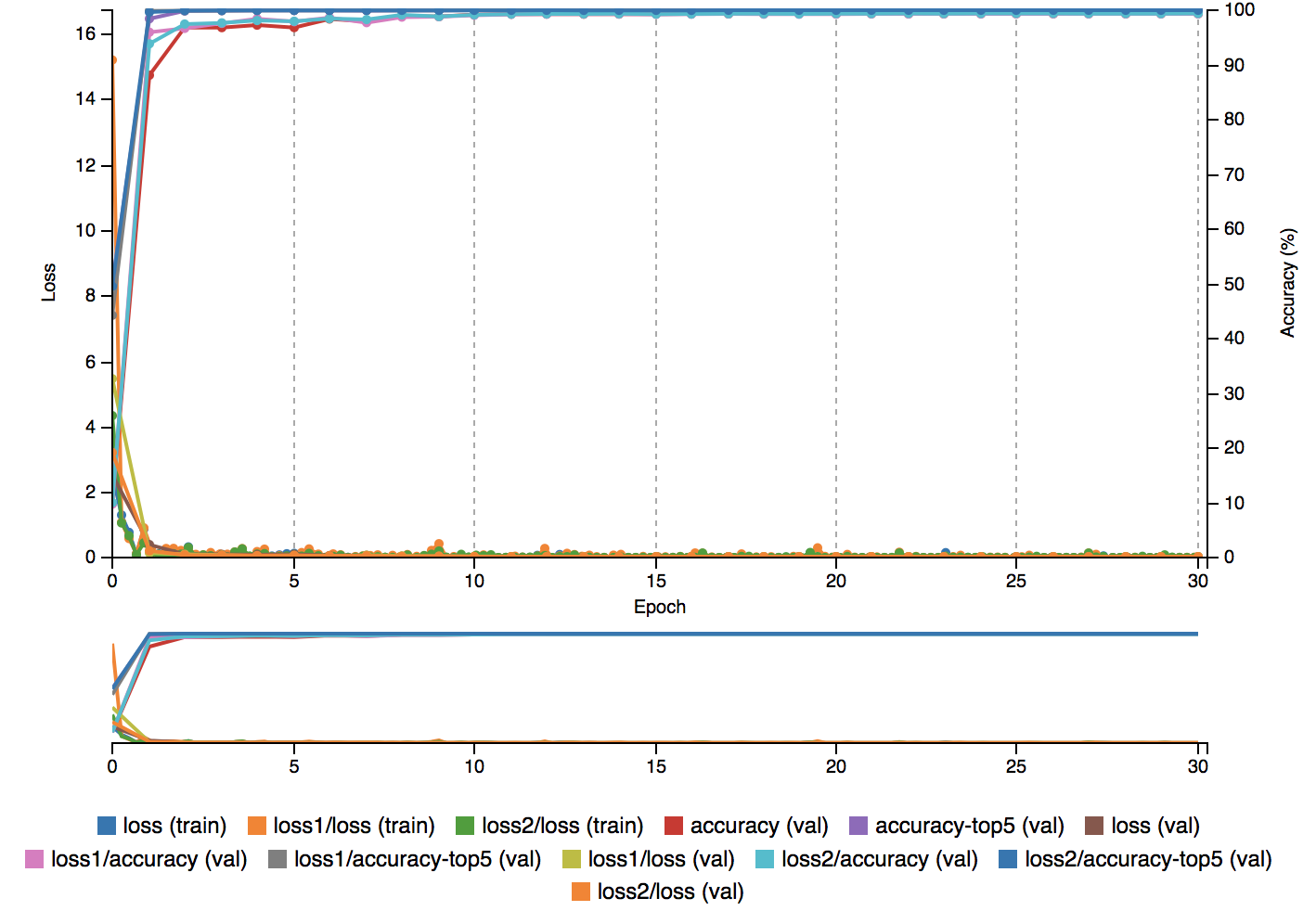
**Figure 3. Loss and Accuracy Over Time of LeNet Model on MNIST Data.**



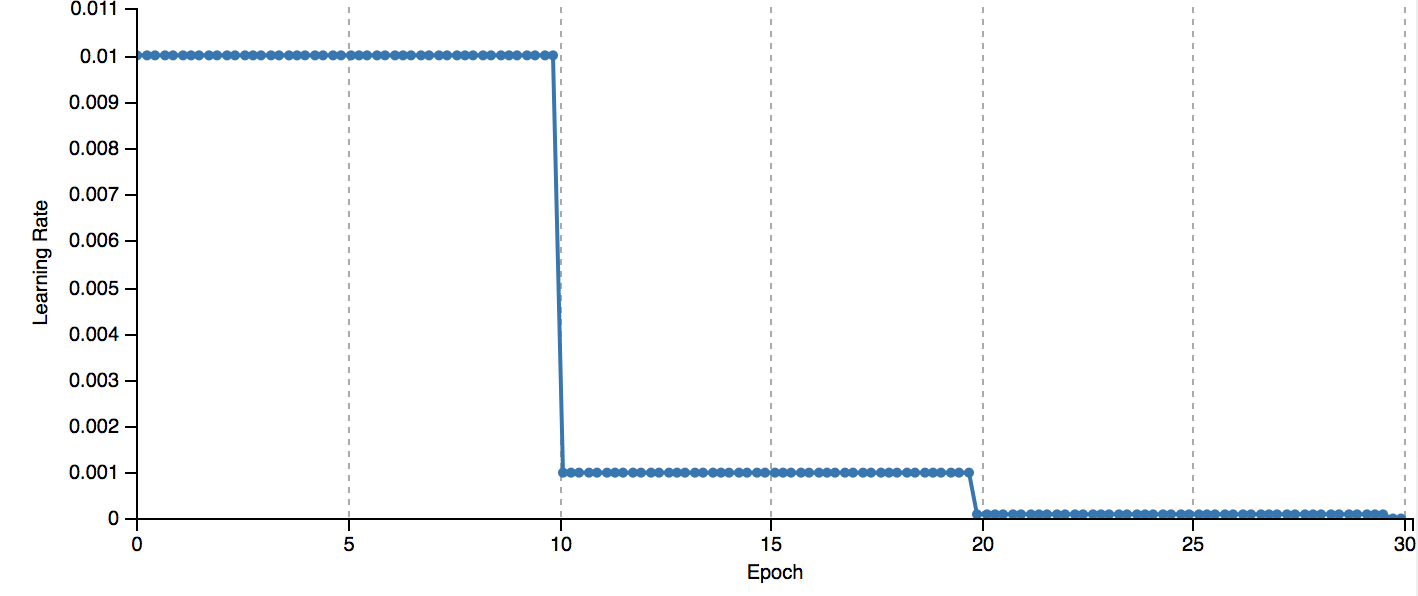
**Figure 4. Loss and Accuracy Over Time of LeNet Model on MNIST Data.**

**GoogleNet Results**

The GoogleNet model also had excellent performance and demonstrated no overfitting; however, it took a comparatively long time train - roughly 5.5 hours. Figure 5 below shows the loss in the model quickly drops to near zero while the accuracy quickly jumps to near 100%. Figure 6 shows the training rate throughout the model run. At around the 10th epoch, the learning rate dropped dramatically and by the 20th epoch, the learning rate had dropped to near zero. Because of the simplicity of the data set and the significant amount of data, the model performed very well with the initial hyperparameters; therefore, no significant tuning was necessary



**Figure 5. Loss and Accuracy Over Time of LeNet Model on MNIST Data.**

****

**Figure 6. Loss and Accuracy Over Time of LeNet Model on MNIST Data**

**Comparing LeNet vs GoogleNet**

Both models quickly achieved high accuracy and low loss. Additionally, the rate of learning changed in similar ways for both models. Neither model had issues with overfitting. The major difference was the time it took to train the models. LeNet trained in less than five minutes while GoogleNet took roughly 6 hours to train. This can be explained by the complexity of GoogleNet compared to LeNet (see appendix B for visualization)

**Conclusion**

Since LeNet was originally designed to recognize handwritten digits and was created long before the explosion in computing power, it makes sense that it not only performs very well, but also has a short run time. The LeNet Model worked so well it wasn’t necessary to do a lot of tuning or hand engineering. Additionally, it wasn’t necessary to implement fancy architectures like ResNets or inception layers. The classification problem was simple enough and there was enough data to work with that the very simple LeNet model performed well. This shows that implementing CNNs is about much more than fancy architectures. Sometimes going with the simple implementation provides the performance that is needed and provides benefits such as lower costs of implementation and is easier to test.

## 7.0 Project Process Implementation

This section provides details of the process used to implement the best performing model (LeNet) as discussed in the results section. While I experimented with other models, only the process for implementing the best performing model (the LeNet model) is described here.

**Step 1: Get TX1 Working**

First, the projected needed a simple demonstration of image classification on the TX1. The jetson-inference library provides “TensorRT-enabled deep learning primitives for running Googlenet/Alexnet on live camera feed for image recognition”. Using the terminal on the Jetson TX1, I used the following commands to get a GoogleNet model pre-trained with ImageNet onto the TX1[[4]](#footnote-3). Note: the TX1 came flashed with JetPack.

Install dependencies:

$ sudo apt-get install git

$ sudo apt-get install cmake

Clone repository:

$ git clone <https://github.com/dusty-nv/jetson-inference>

Build, configure, compile (this step can take up to 30 minutes):

$ mkdir build

$ cd build

$ cmake ../

$ make

Test to make sure it works:

$ cd jetson-inference/build/aarch64/bin

$ ./imagenet-camera

After typing the last command, a camera image appears on the screen and is able to classify images according to ImageNet dataset and GoogleNet architecture.

**Step 2: Set up server**

A typical workflow would require designing, building, training and testing a deep neural network on a host PC and then deploying the trained model to a distributed machine for inference. So, in order to better demonstrate a typical workflow, I set up an AWS server to use as the host PC. I worked with an AWS server with Ubuntu 16.04, CUDA toolkit, and DIGITS pre-installed[[5]](#footnote-4). A p2.Xlarge EC2 instance type was selected to take advantage of the GPU performance[[6]](#footnote-5).

Ssh into server:

$ chmod 400 key.pem

$ ssh -i ~/Desktop/key.pem ubuntu@publicDNS

**Step 3: Get MNIST data onto server**

Next, the project needed to get the MNIST data onto the host PC. The data is available through the NVIDIA/DIGITS github repo[[7]](#footnote-6).

Install dependencies:

$ sudo apt-get update && sudo apt-get -y upgrade

$ sudo apt-get install python-pip

$ sudo apt install python-minimal

$ pip install pillow

$ pip install numpy

Download DIGITS git repo and run a provided script to create the mnist dataset in the mnist folder:

$ git clone <https://github.com/NVIDIA/DIGITS.git>

$ cd DIGITS

$ python -m digits.download\_data mnist ~/mnist

**Step 4: Upload data into DIGITS**

DIGITS is web-based application that allows for rapid prototyping and visualizing of neural networks. In order to use DIGITS, it is necessary to upload the MNIST data to the DIGITS server. It is important to note that proper permissions are necessary and the MNIST data must be in the proper location on the host PC for DIGITS to read it[[8]](#footnote-7). When uploading data into DIGITS, the names of the folders are the classifications. In this example, we have 0,1, 2, 3, 4, 5, 6, 7, 8, and 9. Uploading this dataset takes approximately 5 minutes.

In terminal on host PC

Get permissions right to move data to home/digits/data and moe:

$ sudo chown ubuntu:ubuntu /home/digits/data

$ mv /home/ubuntu/mnist /home/digits/data

Get permissions right to allow DIGITS to read files[[9]](#footnote-8):

$ chmod -R +r /home/digits/data

After these steps, navigate to the host PC public IP address port 5000. Here the project accesses the DIGITS graphic user interface. Certain parameters can be changed, for example:

* Image size: 28 x 28
* color/grayscale (grayscale)
* trainingImages: /data/mnist/train
* Test Images: /data/mnist/test
* % of training set for validation:20

**Step 5: Train LeNet model with MNIST data**

Next, with the data loaded into DIGITS, the project needed to train a model[[10]](#footnote-9). DIGITS works through a graphic user interface accessed by going to the server's public IP address followed by :5000. Here a variety of tuning parameters are available, most notably:

* Solver type: stochastic gradient descent
* Training rate: .01
* Batch size: 1
* Epochs: 30
* Subtract mean: images
* Data Augmentations: none

**Step 6: Deploy trained model to Jetson TX1**

From the TX1, we can access the DIGITS server using the AWS server’s public IP address followed by :5000. Once we are on the DIGITS server, we can download the trained model onto the TX1. The model is downloaded as a tar file and needs to be extracted[[11]](#footnote-10).

Extract model (use name of file):

$ tar -xzvf 20171231-035440-39cd\_epoch\_30.0.tar.gz

Then put each file into a new folder, rename the folder LeNet1, and place into the same directory as the imagenet-camera file on the TX1. This is likely /jetson-inference/build/aarch64/bin. Then cd to this directory and type:

$ NET=LeNet

$ ./image-camera \

--prototxt=$NET/deploy.prototxt \

--model=$NET/snapshot\_iter\_22500.caffemodel \

--labels=$NET/labels.txt \

--input\_blob=data \

--output\_blob=softmax

This will execute the LeNet model trained with MNIST data. The camera will open on the screen and the TX1 will be able to correctly classify handwritten digits.

## References

<https://en.wikipedia.org/wiki/Convolutional_neural_network#Fully_connected>

Andrew Ng. <https://www.coursera.org/specializations/deep-learning>

<http://web.engr.illinois.edu/~slazebni/spring17/lec01_cnn_architectures.pdf>

<https://hacktilldawn.com/2016/09/25/inception-modules-explained-and-implemented/>

## Appendix A: LeNet Caffe Code

Note: GoogleNet code is not provided because it is so long - about 50 pages)

name: "LeNet"

layer {

name: "train-data"

type: "Data"

top: "data"

top: "label"

data\_param {

batch\_size: 64

}

include { stage: "train" }

}

layer {

name: "val-data"

type: "Data"

top: "data"

top: "label"

data\_param {

batch\_size: 32

}

include { stage: "val" }

}

layer {

# Use Power layer for input scaling

name: "scale"

bottom: "data"

top: "scaled"

type: "Power"

power\_param {

# 1/(standard deviation on MNIST dataset)

scale: 0.0125

}

}

layer {

name: "conv1"

type: "Convolution"

bottom: "scaled"

top: "conv1"

param {

lr\_mult: 1

}

param {

lr\_mult: 2

}

convolution\_param {

num\_output: 20

kernel\_size: 5

stride: 1

weight\_filler {

type: "xavier"

}

bias\_filler {

type: "constant"

}

}

}

layer {

name: "pool1"

type: "Pooling"

bottom: "conv1"

top: "pool1"

pooling\_param {

pool: MAX

kernel\_size: 2

stride: 2

}

}

layer {

name: "conv2"

type: "Convolution"

bottom: "pool1"

top: "conv2"

param {

lr\_mult: 1

}

param {

lr\_mult: 2

}

convolution\_param {

num\_output: 50

kernel\_size: 5

stride: 1

weight\_filler {

type: "xavier"

}

bias\_filler {

type: "constant"

}

}

}

layer {

name: "pool2"

type: "Pooling"

bottom: "conv2"

top: "pool2"

pooling\_param {

pool: MAX

kernel\_size: 2

stride: 2

}

}

layer {

name: "ip1"

type: "InnerProduct"

bottom: "pool2"

top: "ip1"

param {

lr\_mult: 1

}

param {

lr\_mult: 2

}

inner\_product\_param {

num\_output: 500

weight\_filler {

type: "xavier"

}

bias\_filler {

type: "constant"

}

}

}

layer {

name: "relu1"

type: "ReLU"

bottom: "ip1"

top: "ip1"

}

layer {

name: "ip2"

type: "InnerProduct"

bottom: "ip1"

top: "ip2"

param {

lr\_mult: 1

}

param {

lr\_mult: 2

}

inner\_product\_param {

# Since num\_output is unset, DIGITS will automatically set it to the

# number of classes in your dataset.

# Uncomment this line to set it explicitly:

#num\_output: 10

weight\_filler {

type: "xavier"

}

bias\_filler {

type: "constant"

}

}

}

layer {

name: "accuracy"

type: "Accuracy"

bottom: "ip2"

bottom: "label"

top: "accuracy"

include { stage: "val" }

}

layer {

name: "loss"

type: "SoftmaxWithLoss"

bottom: "ip2"

bottom: "label"

top: "loss"

exclude { stage: "deploy" }

}

layer {

name: "softmax"

type: "Softmax"

bottom: "ip2"

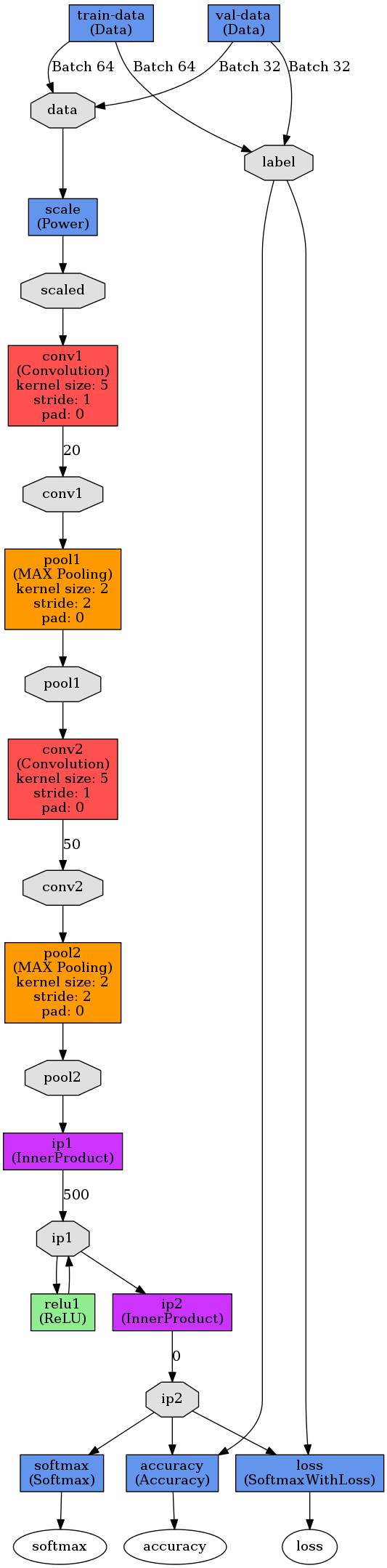
top: "softmax"

include { stage: "deploy" }

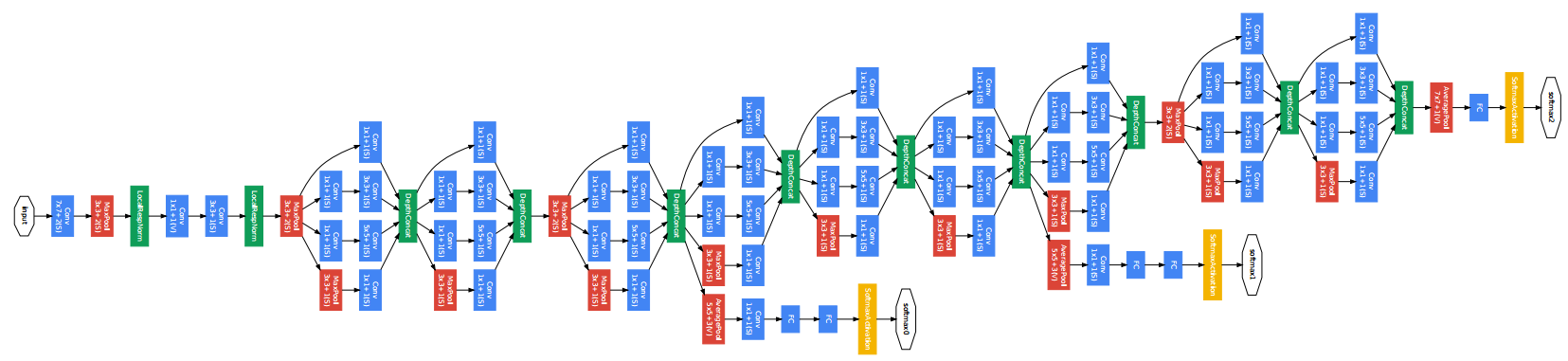
}

## Appendix B: Visualizations

LeNet Architecture:



GoogleNet architecture:



1. Y. LeCun,et al. Gradient-based learning applied to document recognition. 1998. [↑](#footnote-ref-0)
2. Szegedy, et al. Going Deeper with Convolutions. 2014. [↑](#footnote-ref-1)
3. https://developer.nvidia.com/digits [↑](#footnote-ref-2)
4. <https://github.com/dusty-nv/jetson-inference> [↑](#footnote-ref-3)
5. <https://aws.amazon.com/marketplace/pp/B076DHKCZJ> [↑](#footnote-ref-4)
6. <https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AccessingInstancesLinux.html> [↑](#footnote-ref-5)
7. <https://github.com/NVIDIA/DIGITS/blob/master/examples/s3/README.md> [↑](#footnote-ref-6)
8. <https://devtalk.nvidia.com/default/topic/1026757/-quot-folder-does-not-exist-or-is-not-reachable-quot-on-digits-ami/> [↑](#footnote-ref-7)
9. <https://github.com/NVIDIA/DIGITS/issues/1026> [↑](#footnote-ref-8)
10. <https://github.com/NVIDIA/DIGITS/blob/master/docs/GettingStarted.md> [↑](#footnote-ref-9)
11. <https://github.com/dusty-nv/jetson-inference> [↑](#footnote-ref-10)