Machine Learning Engineer Nanodegree
Capstone Project

Charles Costello
November 2nd, 2016

I. Definition

Project Overview

In the quest for artificial intelligence, one of the greatest hurdles is machine perception. Demonstrating that a machine is capable of receiving, processing, and, in a sense, understanding sensory input in a manner similar to humans would be a major step towards building a genuinely intelligent machine.

One of the major problem domains in machine perception is vision, and more specifically image classification. Inspired by Hubel and Wiesel's work on visual perception in cats, many scientists and engineers have created successful image classification machine learning models using deep neural networks. A number of these models were published in papers that have had great influence on the field, particularly Lecun et al. 1998, Krizhevsky et al. 2012, Simonyan et al. 2014, and, most directly connected with our project, Goodfellow et al. 2013.

In this project, we will attempt to solve the problem of image classification using a similar approach to those listed above with the Street View House Numbers dataset (SVHN) collected by Google Street View. We hope that by successfully training a learning model on this data, we will gain insight into how best to solve problems of image classification using deep learning.

Problem Statement

In this image classification project, we are trying to train a machine learning model to correctly recognize digit sequences in images. Given an image with a sequence of digits in it, such as a picture of a house number, we want our model to be able to tell us what the sequence of numbers is. This is a supervised machine learning task, and like any such task, we will be training our model on a series of labeled data points, and subsequently testing it on a number of unlabeled data points. In this case the data points are RGB images in a png format.

Teaching a machine to recognize a sequence of digits from an image is a difficult problem. In order to build an image classification model, we need to collect and preprocess our data, develop a deep learning model, train the model, and measure its performance. We will then iteratively refine the model as needed to increase performance.

Our problem of image classification is solved by creating a model that can correctly recognize digit sequences in new images. This means that the model will correctly classify each individual digit in the image, as well as report them in the correct order with the correct number of digits. We can measure our model's ability to do this by obtaining its accuracy on a test dataset which consists of different images than we used to train the model. We will then define a performance benchmark, and once we obtain an accuracy that exceeds this benchmark, we can consider our problem solved.

Metrics

Image classification is the task of training a model to correctly identify images. As such, we can measure the performance of our model by determining the percentage of images that the model correctly classifies. This is the model's accuracy. In order to ensure the integrity of this accuracy, we will ensure that the images used for testing performance are not the same images as those used to train the model.

As SVHN contains images with a variable number of digits, the determination of whether an image is classified correctly or not needs to be precisely defined. We will consider an image correctly classified if and only if the model outputs the correct number of digits, and each digit is correct, and they are in the correct order.

However, as both the percentage of each digit (0-9) and the number of digits per image (1-5) are not perfectly balanced across SVHN sets, we may gain a more complete picture of the model's ability to generalize by taking not just accuracy, but also precision and recall into account. Therefore we will also use the F1 score during cross-validation in order to determine the best hyperparameter values. The F1 score is calculated using precision and recall. However, we will still use accuracy as our primary metric to report our benchmark and final performances, as it is the most intuitive and widely used metric when reporting image classification performance. We can derive the formulas for accuracy and F1 using the below equations:

$$accuracy = \frac{true\ positive + true\ negative}{true\ positive + true\ negative + false\ positive + false\ negative}$$

$$F1 = 2 \times \frac{precision \times recall}{precision + recall} \text{ with } precision = \frac{true\ positive}{true\ positive + true\ negative} \text{ and } recall = \frac{true\ positive}{true\ positive + false\ negative}$$

II. Analysis

Data Exploration

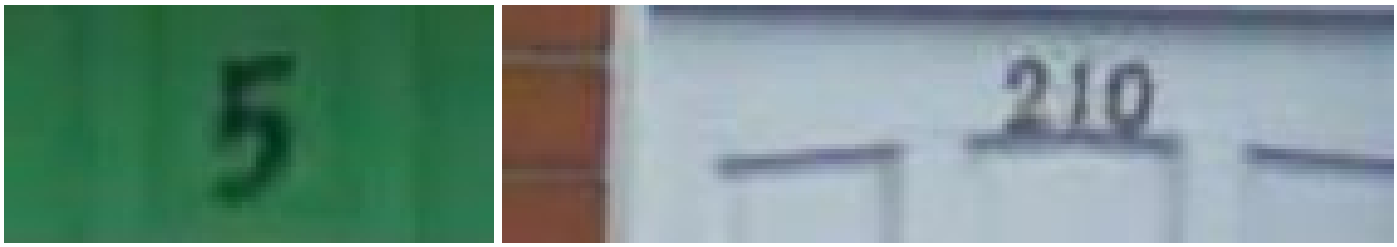

Figure 1: Two images from the SVHN test set.

In order to train and test our model, we will use the Street View House Number (SVHN) dataset collected by Google Street View. It is available in two forms, as the original images with bounding boxes, and as cropped single digit images. We will use the former in our project, as it represent the real world problem (see figure 1). The dataset is divided into three sets: train, test, and extra. The train set contains 33,402 images with 73,527 digits, the test set contains 16,068 images with 26,032 digits, and the extra set contains 202,353 images with 531,131 digits. Each set contains the images as variable-size png files, as well as a digitStruct file that contains the position, size, and label for each bounding box.

	0s (%)	1s (%)	2s (%)	3s (%)	4s (%)	5s (%)	6s (%)	7s (%)	8s (%)	9s (%)
Train	6.75	18.92	14.45	11.60	10.18	9.39	7.82	7.64	6.89	6.36
Test	6.70	19.59	15.94	11.07	9.69	9.16	7.59	7.76	6.38	6.13
Extra	8.58	17.05	14.07	11.44	9.53	10.07	7.83	8.28	6.66	6.49

Figure 2: Percentage of each digit in each set in SVHN.

	1 digit (%)	2 digits (%)	3 digits (%)	4 digits (%)	5 digits (%)
Train	15.38	54.28	26.02	4.29	0.03
Test	19.00	63.94	15.92	1.12	0.02
Extra	4.64	35.45	52.77	7.09	0.06

Figure 3: Percentage of the number of digits per image in each set in SVHN.

Looking at figures 2 and 3, we can see both the percentage of each digit and the percentage of digits per image are not balanced across SVHN sets. Additionally, exactly one out of 251,823 total images has more than five digits. We will treat this image as an outlier in our dataset.

Exploratory Visualization

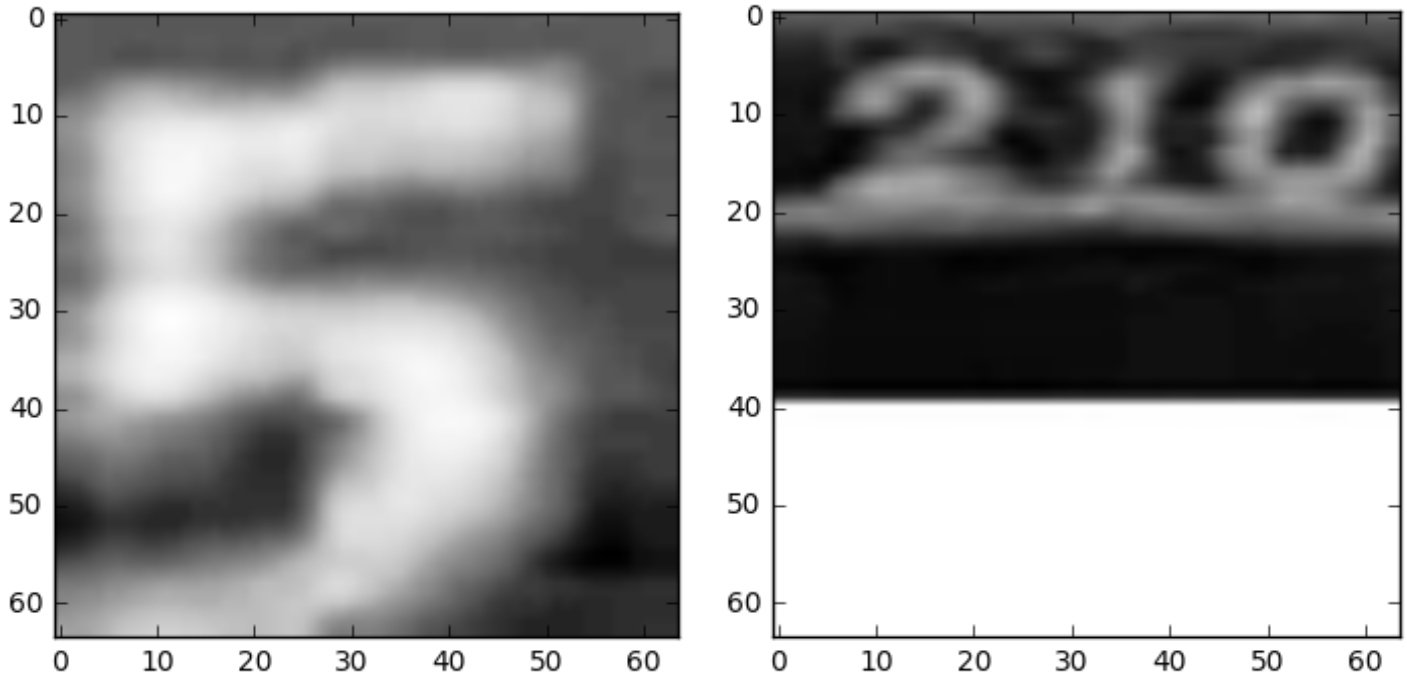

Figure 4: Same two images from the test set, after preprocessing.

In figure 4 we see the same two sample images as in figure 1. However, these images differ from the images in figure 1 because they have been through preprocessing. They have been converted to grayscale, cropped to only show the digits as far as possible, and resized so that each image is 64 pixels by 64 pixels. This visualization is relevant because it shows us the images as they will be inputted into the model for training and testing. Most

importantly, we can see that we have cropped a very large part of these images away in order to focus on what the model is trying to classify: the digits.

Algorithms and Techniques

Teaching a computer to classify images is a difficult task. Attempts to explicitly program a computer to classify images have been shown to be ungeneralizable. Therefore, we need to implicitly train a model to recognize images. This is done through machine learning. Machine learning, and even the subtopic of classification, is a broad field that offers many algorithms to train models with. We could choose support vector machines, nearest neighbor, naive bayes, or dozens of other algorithms. Most of these algorithms, however, do not perform well on problems of image classification.

Images are very high dimensional data, and those dimensions (pixel densities) are not obviously connected to the content of the image. Therefore, in order to train a model well, we need an algorithm that allows us to extract relevant features from this high dimensional input. This in turn necessitates a more complex model, one that allows for multiple nonlinear transformations to perform this feature extraction. Neural networks, with their arbitrarily deep architecture and ability to use many functions to introduce nonlinearity, are consequently an excellent choice for image classification, and are what we will use in this project.

As the structure of neural networks is less well defined than other machine learning algorithms, we will take an exploratory approach when implementing our model. We will create multiple models of increasing complexity, starting with a simple linear neural network, then adding multiple non-linear transformations, and finally adding convolutions and pooling. This approach will provide us with insight into both how best to choose the optimal neural network architecture, and how model performance changes as a function of model complexity. Before actually implementing our models, however, it is important to understand the theory behind neural networks to ensure that our implementation is correct and efficient.

Neural networks are graph structures that are used in a variety of machine learning contexts. As we will use them to implement supervised classification, we will describe them in this context. In general, we can think of neural network training in two discrete steps: forward propagation and backpropagation. We will look at each in turn.

Forward propagation is the process of generating a prediction from an input, and quantifying the error between that prediction and that input's true label. The input, as well as the prediction, take the form of multidimensional arrays, or tensors. The quantified error is generally called the loss. In order to calculate the loss, the model needs to generate a prediction from an input. The model first transforms the input through a series of one or more layers, each one implementing some function, into an output tensor that can be interpreted as a prediction. In order for these predictions to be accurate, the input data must be transformed in an intelligent way. The model's "intelligence" is stored in each layer in other tensors, generally called weights and biases, which interact with the input of each function. The actual loss can be then be calculated from the output of the model's final layer. The loss can be calculated many different ways, one of the most popular ways being softmax cross-entropy. The final output is fed to the software cross-entropy functions to obtain a scalar loss. Softmax cross-entropy can be calculated by the following equations:

$$softmax(x_i) = \frac{exp(x_i)}{\Sigma exp(x)}$$

$$cross-entropy(x) = -\Sigma y log(x) \text{ with } y = \text{true labels and } x = \text{model predictions after softmax}$$

After calculating the loss, the model then performs backpropagation to update the weights and biases in order to minimize that loss. This update is accomplished using the chain rule to propagate the gradient through the

model, starting with the loss and finally calculating the partial derivatives of the all weights and biases. These partial derivatives can then be used by gradient descent to update the weights and biases by some magnitude determined by a learning rate hyperparameter. There are many optimization algorithms that implement this process of gradient descent slightly differently, but all update the weights and biases by taking steps towards a minimum in the gradient space. Also, it is important to note that as backpropagation requires the entire neural network graph to be differentiated, it is imperative that each function used in forward propagation be differentiable.

Forward propagation and backpropagation, when taken as a whole, constitute a single step in the training of a neural network. Neural networks are trained for many steps that make successive updates to the weights and biases in order build up the "intelligence" of the model. Much of the work of implementing neural networks is determining which functions most correctly and efficiently train the model, the collection of which constitute the architecture of the model. In this project, we explore three different architectures; we will look at the general architecture of each here, while describing specific implementation details later.

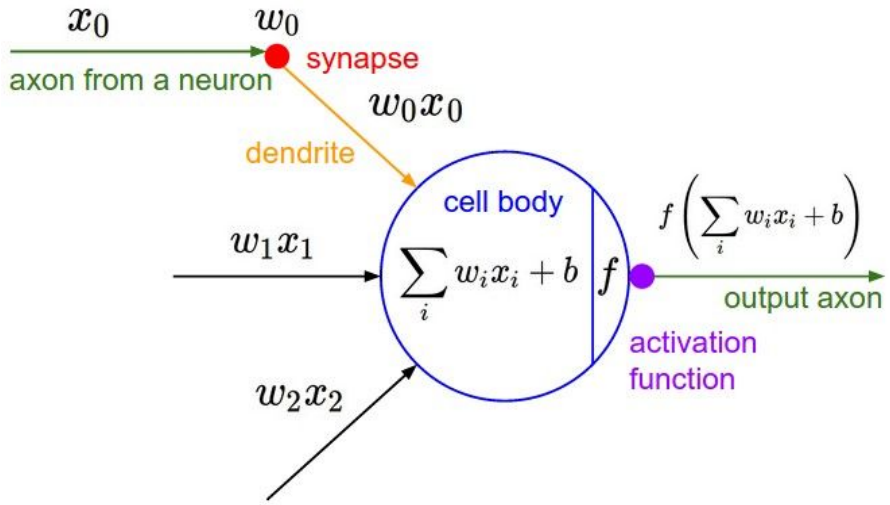

Figure 5: Representation of a linear neural network architecture, from Stanford's course on Convolutional Neural Networks for Visual Recognition

Our first model, a linear neural network uses the simplest possible function to transform the input: a matrix multiplication between the input and weights then addition of the biases (figure 5). This essentially implements logistic regression, which, when vectorized, takes the same form as the equation for a line:

$$f(x) = wx + b$$

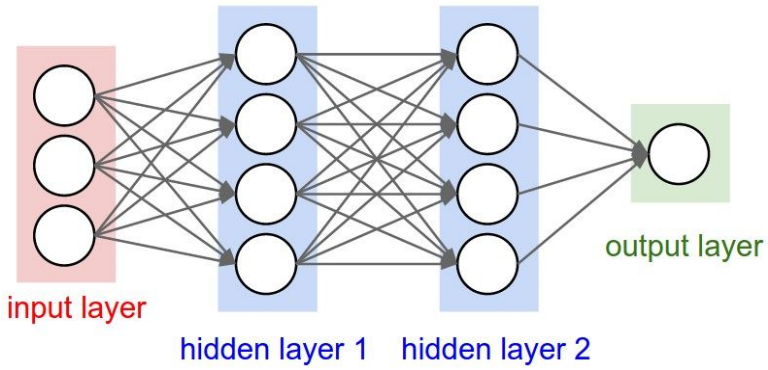

Figure 6: Representation of a deep neural network architecture, from Stanford's course on Convolutional Neural Networks for Visual Recognition

Our second model, a deep neural network, uses multiple rectified linear unit (ReLU) functions to introduce nonlinearity (figure 6). The ReLU function is itself simple, essentially placing a lower bound of zero on the input tensor:

$$ReLU(x) = max(x, 0)$$

Adding this nonlinearity allows the model to develop a much more rich interpretation of the input data than simple linear transformations allow, giving it the ability to train more efficiently.

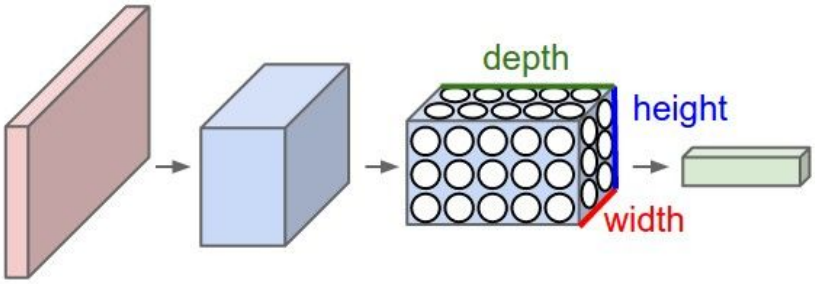

Figure 7: Representation of a convolutional neural network architecture, from Stanford's course on Convolutional Neural Networks for Visual Recognition

Our final model is a convolutional neural network. Convolutional neural networks differ from fully connected deep neural networks in that instead of representing network layers as flat matrices, they represent layers as volumes, and iteratively "stretch" an image as in figure 7. As a result of this "stretching," the model extracts relevant features from the image while also limiting the number of parameters used. The actual convolutions themselves are functions that compute a matrix multiplication between the weights and only a small window of the input, generally controlled by hyperparameters that define the size and stride of that window. Each convolution performs this matrix multiplication multiple times with different weights, stacking together the resulting tensors. Determining the length of a convolution's stride and the number of times it runs defines the output's dimensions.

We also utilize pooling functions in our last model. Pooling is the process of further reducing the dimensionality of an input by reducing data in a small window into a scalar value. This reduction can be calculated through various means, most often by taking the average or max value in the window. The window's size and stride are generally controlled by hyperparameters.

Beyond the basic workings of neural networks and their various architectures, there is one more important technique to consider: regularization. Regularization is used to combat overfitting and generally create a more robust model that generalizes well to new data. There are two forms of regularization we will be using in our models: L2 regularization and dropout. L2 regularization dampens the effect of weights by reducing their value via the following equation:

$$L2(x) = \frac{1}{2}\lambda x^2$$

With λ being a hyperparameter constant called the regularization constant. We generally only regularize the final layer's weights. Also, squaring the weights tensor works to dampen weights with a greater magnitude more. Dropout is the process of setting some proportion of the values in a weights tensor to zero, generally 50%. This adds robustness to the model by forcing it to define classes with redundant features. Similar to L2 regularization, we generally only perform dropout on the last layer of the model. We will use this theoretical backdrop to implement our models.

Benchmark

When developing a learning model, it is useful to define a performance benchmark to compare the model's performance against. In our case, we will use the performance of our second model, the deep neural network, as a benchmark for our subsequent models. As the deep neural network represents a fairly simple way to train our model, using it as a benchmark will provide meaningful contrast with our later, more complex model, allowing us to see precisely how we can increase performance by introducing convolutions into our model.

We use accuracy as our performance benchmark metric, as opposed to the F1 score, because we want our benchmark and final performance to be as straightforward as possible to interpret, and the simplest and most widely accepted way to express a model's ability to perform classification is to measure its accuracy. The final performance benchmark is the accuracy that this deep neural network achieves on the test set: 25.70%.

III. Methodology

Data Preprocessing

The SVHN dataset is comprised of png images and digitStruct files. In order to input this data into our model for training, it must first be preprocessed. For the digitStruct files, this just entails extracting the image metadata (bounding box dimensions and labels) into a Python dictionary for easy retrieval.

Converting the images into training-ready data takes multiple steps, however. First we convert the images to grayscale (from three channels to one). During the training process we want our model to extract the most relevant features from the images in order to recognize digits. The color of the image does not add semantic information to the digits, but only serves to add overhead to the training process. Consequently, we removed color from the images.

After converting to grayscale, we want to crop the images. As we saw earlier when we explored that data, each image contains information beyond the digits in it, which is extraneous to our object of recognizing the digits. By cropping the image around the bounding boxes already present in the metadata, we significantly decrease this extraneous information, making it much easier for our model to identify the digits without getting confused by the other information in the images.

Next, we standardize the size of the images. During image classification training, the model treats each pixel as a feature. And in order to train the model, all images need to have the same number of features. Consequently, we resize all images to be 64x64 pixels, thus ensuring a uniform number of features across all images. Also, we normalized the feature values, transforming the range from [0, 225] to [-0.5, 0.5]. We do this in order to center the mean of the data, allowing for easier calculations during training. Performing these preprocessing steps ensures that our models will be able to use the data as efficiently as possible and therefore reach the best performance possible.

SVHN has three data sets: train, test, and extra. For our model, we want to create the traditional three machine learning sets: train, validation, and test. To create our test set, we will simply copy the SVHN test set, which ensures the final accuracy we obtain from our models is comparable to published accuracies. However, we want the train and validation sets to be as balanced as possible for cross validation to be effective, so we take images from the SVHN train and extra sets to create both sets. We also take care to keep the same ratio of total images between our train and validation sets as exists between the SVHN train and extra sets, roughly one to six. This

results in the sets being very well balanced for both percentage of each digits and percentage of number of digits per image, as figures 8 and 9 show.

	0s (%)	1s (%)	2s (%)	3s (%)	4s (%)	5s (%)	6s (%)	7s (%)	8s (%)	9s (%)
Train	8.34	17.26	14.13	11.46	9.61	9.99	7.83	8.21	6.69	6.48
Valid	8.53	17.54	13.89	11.44	9.61	10.01	7.82	8.13	6.68	6.34
Test	6.70	19.59	15.94	11.07	9.69	9.16	7.59	7.76	6.38	6.13

Figure 8: Percentage of each digit in train, validation, and test sets.

	1 digit (%)	2 digits (%)	3 digits (%)	4 digits (%)	5 digits (%)
Train	6.17	38.12	48.98	6.67	0.05
Validation	5.95	37.99	48.96	7.05	0.05
Test	19.00	63.94	15.92	1.12	0.02

Figure 9: Percentage of the number of digits per image in train, validation, and test sets.

Also, when creating these sets we need to determine how to represent digit sequences of varying lengths in individual images, as we need to feed them into the model in a standard format. As we noted above, all but one image has less than six digits. Due to this, we can create five different vectors for each image, one for each possible digit. These vectors will have a length of 11, with zero through nine representing their respective digit and ten representing no digit. These vectors are then one hot encoded, which allows us to apply the softmax function to them during training. Finally, we remove the singular outlier image.

Implementation and Refinement

When actually implementing our model, we use iterative refinement to build a sequence of models with increasing complexity that achieve increasing accuracy. This implementation is done in Python 2.7, using with a variety of libraries, most notably TensorFlow. TensorFlow is a high-level library for developing and training deep learning models, and using it allows us to develop our specific models in a clean, efficient manner that is easily understood.

The first model is a one layer linear neural network (LNN), which we create in order to ensure that we understand how to successfully define and run a neural network in TensorFlow. To implement this model, we follow basic TensorFlow structure. First we create a graph with data variables, weights, and biases. We then create a model that defines the neural network, in this case just one layer that is implemented with a simple matrix multiplication to create the output. However, as we are classifying up to five digits per image, we need to output five different values, one representing each digit space in the image. Consequently, we make five sets of weights, biases, and outputs. Next we define the loss function, which calculates the cross-entropy of the output after it has been passed through a softmax function. Then we define the optimizer; here we choose the Adam optimizer, which implements gradient descent as well as automatically calculates the learning rate for different parameters. We decide to use Adam based on informal experimentation with different optimizers that TensorFlow offers. Finally, we define the predictions for each set, which are the model's classification of each digit space in the image. As we are purposely trying to create a simple model here, we do not perform any cross

validation with the LNN model. We do perform L2 regularization on the weights however, and include this regularization in the loss function.

After defining our graph, we need to actually populate it with data and obtain predictions from it. In TensorFlow, this is done in a session. We use stochastic gradient descent by feeding the data into the model in batches for training. We train the model for ten epochs, recording the loss, train accuracy, and validation accuracy every epoch. Finally, after all training is complete, we obtain the final test accuracy: 4.69%. We can see that this performance is atrocious, which means that our model lacks the complexity to perform any real training on the data, and that consequently we need to create a more complex model before we can even obtain a reasonable performance benchmark.

Our next model is a more complex deep neural network (DNN) that introduces nonlinearity via ReLU functions. It has two hidden layers, the first with 1024 nodes and the second with 128. We also introduce dropout in the DNN model. Beyond these changes, we train the deep neural network identically to the linear neural network, and obtain the same performance metrics. The deep neural network test set accuracy is 25.70%. This performance is still terrible, but as this DNN model has a relatively standard neural network architecture, we will use it for our performance benchmark. However, we will have to create a significantly more complex model if we hope to reach reasonable performance.

Layer	Process	Height x Width x Depth
1	Input	64x64x1
2	Convolution	64x64x16
3	Convolution	64x64x16
4	Pooling	32x32x16
5	Convolution	32x32x32
6	Convolution	32x32x32
7	Pooling	16x16x32
8	Convolution	16x16x64
9	Convolution	16x16x64
10	Pooling	8x8x64
11	Convolution	8x8x128
12	Convolution	8x8x128
13	Pooling	4x4x128
14	Fully Connected	128
15	Output	5x11

Figure 10: CNN model architecture

Our final, most complex model is a convolutional neural network (CNN). We can see from figure 10 that our CNN model architecture, including input and output, has 15 layers. Also, we can see the change in dimensionality for each image, going from 64x64x1 in the input to 4x4x128 after all the convolutions and pooling layers, and finally to a 5x11 matrix of digit predictions.

We also perform cross-validation for this model. Though this model has many hyperparameters that could be used in cross-validation, including the number of layers, the optimizer used, and the depth of each convolution, we decided, due to limited time and computational resources, to focus on the learning rate and regularization constant. However, our choice of the Adam optimizer negates the need to cross-validate over the learning rate, as it calculates it dynamically. Consequently, we focused on obtaining the optimal regularization constant. As TensorFlow does not have a built-in function to perform cross-validation, we simply trained our model for one epoch on a series of regularization constant values to obtain the optimal value, in this case 0.01. We use this value when training our final CNN model. This final CNN model takes around 10 hours to train on a single GPU on an Amazon Web Services instance. We are pleasantly surprised to reach 87.38% test set accuracy.

Developing an end-to-end deep learning solution is not a trivial task, and many difficulties presented themselves during the development of our solution. The first was determining the optimal way to create our train, validation, and test sets in a balanced manner, as the three SVHN sets do not obviously have a one-to-one correspondence to these sets. This necessitated a thorough exploration of the data to obtain information both about the distribution of individual digits, and the distribution of the lengths of the digit sequences in the images. After obtaining this information, we were able to logically determine an optimal way to construct our sets, and actually create them as described above. This difficulty highlights the necessity of thoroughly exploring and numerically describing any data being inputted to a deep learning model.

Another main complication of this project lay in constructing the CNN model. Having a 15 layer model that involves both convolutions and pooling makes keeping track of tensor shapes very difficult. This presented a problem, as we need to know what the tensor shape should be at each layer in order to correctly define the shape of the weights and biases, as well as to generally understand how the model is working. The solution to this problem lay in developing helper functions that calculated the shapes of convolution and pooling layers. These functions proved to be immensely useful when constructing the CNN model, and were used to generate the dimensions in figure 10.

One final complication, which is common to all deep learning models with relatively high complexity, was long training times. The CNN model in particular took a very long time to train. Our solution to this was to use a Amazon Web Services instance that had excellent GPU support, allowing our TensorFlow code to execute much faster than it would have on even a multicore CPU. Running our project in this manner demonstrates the growing importance of both cloud computing and GPU utilization when training deep learning models.

IV. Results

Model Evaluation and Validation

As we can see in figure 11, our linear, deep, and convolutional neural network models reached 4.69%, 25.70%, and 87.38% test set accuracies, respectively. As it obtained the greatest test set accuracy, the CNN model is the optimal model. This aligns with the published results for SVHN, almost all of which use convolutional neural networks. More importantly, we can see a clear trend of achieving better performance with deeper networks that utilize convolutions.

	1	2	3	4	5	6	7	8	9	10	Acc %
SVHN	5	210	6	1	9	1	183	65	144	16	100.00
LNN	1	24	4	2	1	6	21	1	114	24	4.69
DNN	1	20	1	9	9	1	105	5	140	10	25.70
CNN	5	210	6	3	9	1	183	65	144	16	87.38

Figure 11: Comparison of first ten test images' labels with each model's predictions, and each model's total test set accuracy score.

Additionally, we know that this model is robust enough to give a comparable level of accuracy when introduced to new data. Firstly, we can make this claim because SVHN contains real-world images, which means there is a large variance in the data, and consequently that small perturbations in data would not noticeably affect the outcome. Also, we can compare the final validation and test accuracies against the final train accuracy to assess robustness. Our final train, validation, and test accuracies were 81.25%, 88.62%, and 87.38%, respectively. The fact that our final validation and test accuracies, which are calculated on images that the model has not seen before, are greater than our final train accuracy means that our model generalizes very well, which in turn lends confidence to our claim that the model is robust.

Justification

The problem of image classification is fairly straightforward to assess, as we ultimately only care about whether all digits in an image are correctly classified. Therefore, we can compare our CNN model's performance with the benchmark DNN model's performance simply by comparing their accuracies. Having already used the F1 score to obtain the best hyperparameter values during cross-validation, we refrain from also using it here in order to keep the comparison as straightforward and concrete as possible.

We created our DNN model to serve as a benchmark, and it achieved 25.70% accuracy on the test set. Our final CNN model reached 87.38% accuracy on the test set. The fact that our model so vastly improved the accuracy, an increase of almost 250%, shows that our CNN model is vastly superior to our benchmark DNN model. Therefore, we can say that the CNN model has solved our problem.

V. Conclusion

Free-Form Visualization

Figure 12 shows learning curves for each model we created in this project. Each individual graph show the loss value and the training and validation accuracy percentages as a function of training time, measured in epochs. Each graph is interesting in its own right, showing steady decreases in loss and increases in accuracy, but these graphs are most interesting when taken together. We can clearly see that as we increase model complexity, the accuracies are comparatively greater and losses comparatively smaller at each time step. This is clear evidence that adding depth and convolution to a neural network increases performance on image classification tasks.

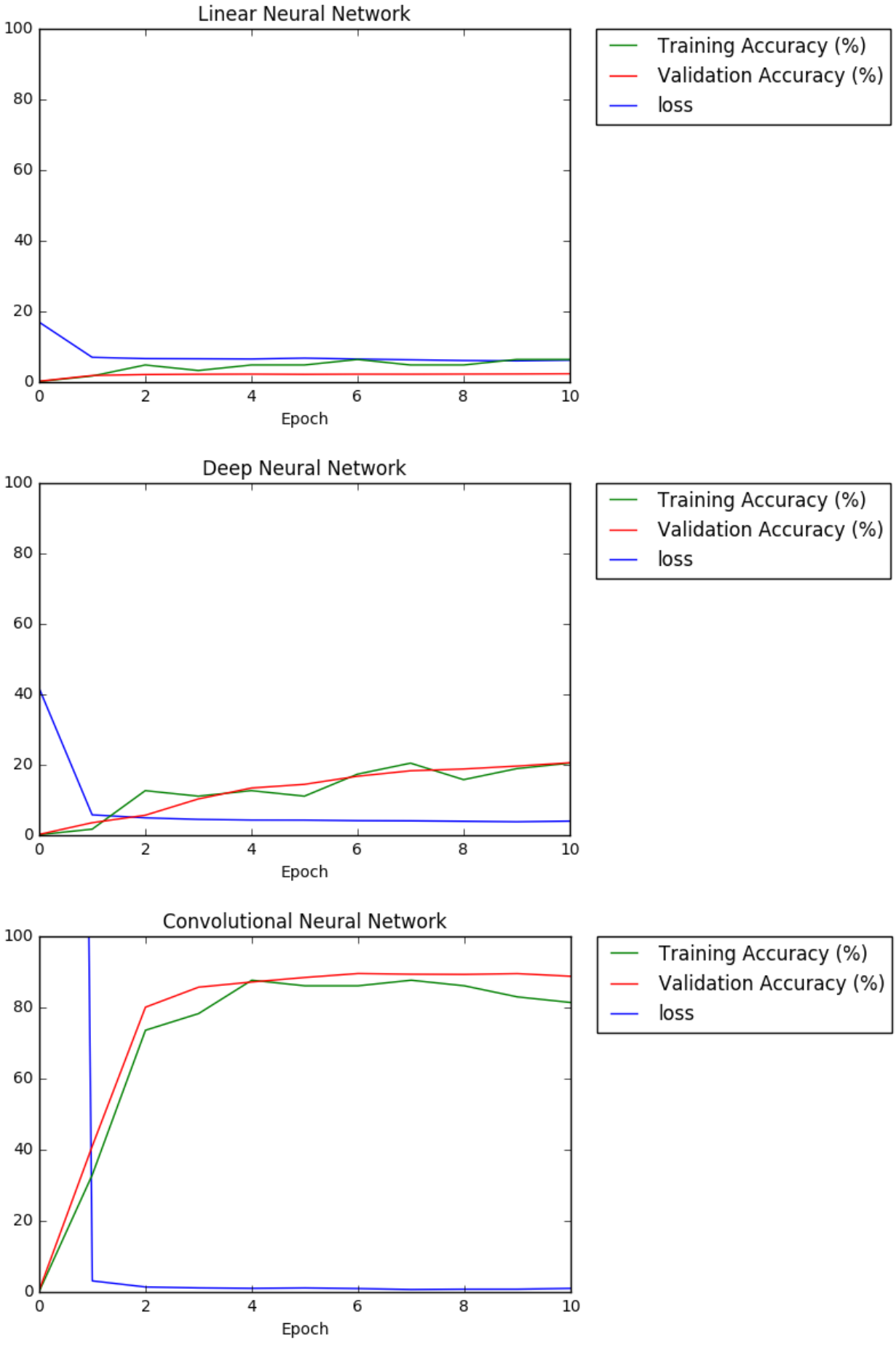

Figure 12: Learning curves for the LNN, DNN, and CNN models.

Reflection

In this project, we utilized deep learning techniques to create a successful image classification model. We began by exploring and preprocessing our data to transform it into a form appropriate to be fed into a neural network. Next we balanced the data to ensure that cross-validation was as efficient as possible. After that we removed a data outlier and defined our performance metrics, accuracy and F1 score. With that initial work accomplished, we implemented three deep learning models, which included a simple linear neural network, a deep neural network that served as a benchmark, and a convolutional neural network that served as a final model. We performed cross-validation and obtained a final performance of 87.38% test set accuracy.

By reflecting on the entire project, we can obtain many insights into the nature of machine learning, and more specifically deep learning in the context of image classification tasks. First, we can see the overwhelming importance of creating a model that is deep and complex enough to adequately extract features from images during training. Both the LNN and DNN models plateaued at very low accuracies; it wasn't until we created a 15 layer CNN that we achieved respectable performance.

Another key insight is the importance of having an adequate amount of data. As we have seen, the SVHN data is divided into train, test, and extra sets. This might tempt one to use just the train set for training and the test set for testing, but this configuration simply does not provide sufficient data to train the model, which ends up overfitting. In order to reach acceptable performance without overfitting we need to utilize all available data, including the extra set.

One final insight is the importance of preprocessing data. The original SVHN images have varying resolutions, and the placement of the digits within the image also varies greatly. Simply resizing all images to the same size and feeding them into the model does not give the performance we are looking for; the model gets confused with all the extra information in the images. Instead, we need to take advantage of the fact that the digits have already been localized and given bounding boxes. This allows us to crop the images, removing most of the information that does not relate to what we are trying to recognize: the digits.

One interesting, and perhaps less obvious, takeaway from this project lies in how the model learns. If we take a close look at figure 11, we can see an interesting pattern: even the LNN model, which achieves atrocious test set accuracy, is fairly close to the SVHN labels when predicting the length of the digit sequence in the images. In most of the ten example images, the length of the LNN model prediction is the same as the SVHN label, and no prediction deviates from the label more than one digit place. This suggests that the model can very quickly ascertain the number of digits in the images, and that most training time is spent classifying each individual digit. This may have ramifications on further work that more generally deals with both identifying the number of objects in an image and classifying those objects.

Ultimately, working through this image classification project has given us tremendous insight into the nature of machine learning and the practical considerations of implementing a deep learning model for image classification. We learned not only how to preprocess data, build deep learning models, perform cross-validation, and measure performance, but also specific insights that helped us achieve excellent performance.

Improvement

Despite the success of our model, it could be improved upon. Things that we could have done to increase our model's performance includes making the CNN even deeper, cross-validating on more hyperparameters, and adding batch normalization. Performing these improvements would certainly allow us to exceed the 87.38% accuracy we obtained.

Beyond specific improvements to the CNN model we created, we could also make our model more general by removing the limitation of classifying exactly five digits per image. One possible way to do this would be to use a recurrent neural network to sequentially process subsections of the image and output the digits one by one. Another worthwhile improvement would be to remove the model's reliance on bounding boxes. This would mean adding a localization task to our model, though, which would require significant changes to our project.

Using deep learning for image classification is an extremely active and interesting field of research. New theories and techniques are published regularly, and there is always the promise of reaching yet higher levels of performance. This project has highlighted many of the fundamental aspects of using deep learning for image classification, and we are excited to keep improving the model and to use it as a sandbox for experimenting with new technologies.

References

I. J. Goodfellow, Y. Bulatov, J. Ibarz, S. Arnoud, and V. D. Shet. 2013. "Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks." *arXiv 1312.6082*

A. Krizhevsky, I. Sutskever, and G. E. Hinton. 2012. "ImageNet Classification with Deep Convolutional Neural Networks." *Advances in Neural Information Processing Systems 25 (NIPS 2012)*

Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. "Gradient-Based Learning Applied to Document Recognition." *Proceedings of the IEEE* 86 (11): 2278–2324.

Y. Netzer, T. Wang, A. Coates, A. Bissacco, B. Wu, and A. Y. Ng. 2011. "Reading Digits in Natural Images with Unsupervised Feature Learning." *NIPS Workshop on Deep Learning and Unsupervised Feature Learning 2011*

K. Simonyan and A Zisserman. 2014. "Very Deep Convolutional Networks for Large-Scale Image Recognition." *arXiv 1409.1556*