MNIST Handwritten Digit Classification

Dorian Knight

Created: July 1st 2025
Updated: July 1st 2025

Contents

1 Project Overview

This project was an introductory investigation into how machine learning works, what it's good for and how to use it. Using TensorFlow, I was able to create a feedforward artificial neural network model that classifies handwritten digits from the MNIST (Modified National Institute of Standards and Technology) database. GitHub repo.

2 Purpose

The project's purpose was to start understanding how neural networks are made, trained and evaluated. By experimenting with TensorFlow's model libraries, I was able to get a taste of what machine learning is capable of.

3 Outcomes

By using TensorFlow's libraries in Python, I was able to create an artificial neural network that can classify handwritten digits from 0-9 with up to 95.84% accuracy. I also experimented with varying the number of epochs used in model training and found that more epochs aren't always better. While an increase in training epochs resulted in plateauing accuracy, loss continued to increase seemingly exponentially. This suggests excessive epochs lead to model overfitting as the model learns the significant differences but also learns the training dataset noise, which is undesirable for model generalizability.

4 Skills Used

(a) Python programming language icon

(b) Tensorflow library icon

Python was used as the base programming language to work in. The TensorFlow library is used to create, train and evaluate the model. The dataset was sourced from Keras, which was imported from the TensorFlow library.

5 Model Construction

The model I built was simple. Each MNIST image was a 28 by 28 grayscale pixel image. Because each image was 28 by 28, I made the input layer 28 by 28 nodes such that each node would map directly to one pixel from the image.

The input layer fed directly into a dense hidden layer where the data was mapped out across 128 dimensions. The dimensionality of my hidden layer dictates the number of attributes that the model could use to differentiate between two handwritten digits.

A quick, intuitive example would be using car type (SUV, hatchback, coupe) and colour to distinguish between different car models. With only two dimensions to compare between, we lack sufficient ability to differentiate between two black SUVs even though one could be a Honda Pilot and the other could be a Lincoln Navigator. Increasing the dimensionality of your space is an essential element for improving model specificity but more dimensions are not always better as this article suggests: [link]. Higher dimensionality often comes at the cost of data density meaning that as you increase the number of dimensions, you'll need exponentially more data to give your model enough information to learn from.

The ReLU (Rectified Linear Unit) was used as the activation function in the hidden layer. A ReLU is a unit ramp function where y=x for all x values above zero, else y=0. Using ReLU introduces non-linearity to my model. Introducing this nonlinearity helps to mitigate the "vanishing gradient problem" which describes when the model doesn't know how to adjust its weights and biases during backpropagation due to other approaches (sigmoid and tanh) saturating when the positive or negative input becomes sufficiently large. The ReLU activation function always produces a derivative of 1 (when the value is positive) which alleviates the problem of saturation ensuring that the gradient never vanishes.

The output layer was composed of ten nodes where each node maps to a digit from zero through nine. The SoftMax function was used as the output layer's activation function as the SoftMax activation function works better in multi-class classification tasks compared to the sigmoid function which does better at binary classification as suggested in this article [link]. In our case, we have ten classes to distinguish between, so the SoftMax operator is the better choice for our output layer.

```
model = keras.Sequential([
    keras.layers.Flatten(input_shape=(28,28)), # Input data is 28x28 pixels
    keras.layers.Dense(128, activation=tf.nn.relu), # Rectified linear unit activation
    keras.layers.Dense(10, activation=tf.nn.softmax) # Output layer picks the "highest probability outcome"
])
```

Figure 2: Model construction layer by layer

6 Model Optimization and Loss

The optimizer and loss functions work together to adjust the node's weights and biases. The 'Adam' optimizer utilizes two key concepts in optimization as suggested by this article [link]:

- Momentum
- Root Mean Square Propagation

The 'Adam' optimization helps to avoid oscillations in gradient descent and reduces early-stage instability. Sparse categorical accuracy is used to quantify the model's loss as it compares how often the model's output matches the training/testing data's label.

7 Model Results

The model was trained using the parameters mentioned above over 1-30 epochs. Using zero epochs was neglected as that would be the equivalent of using an untrained model. An epoch denotes how many times a model gets to look at the training data where one epoch restricts the data to seeing each training image once, while 30 epochs let the model look at each image 30 times.

By varying the number of epochs, we can see how it leads to varying levels of model performance below:

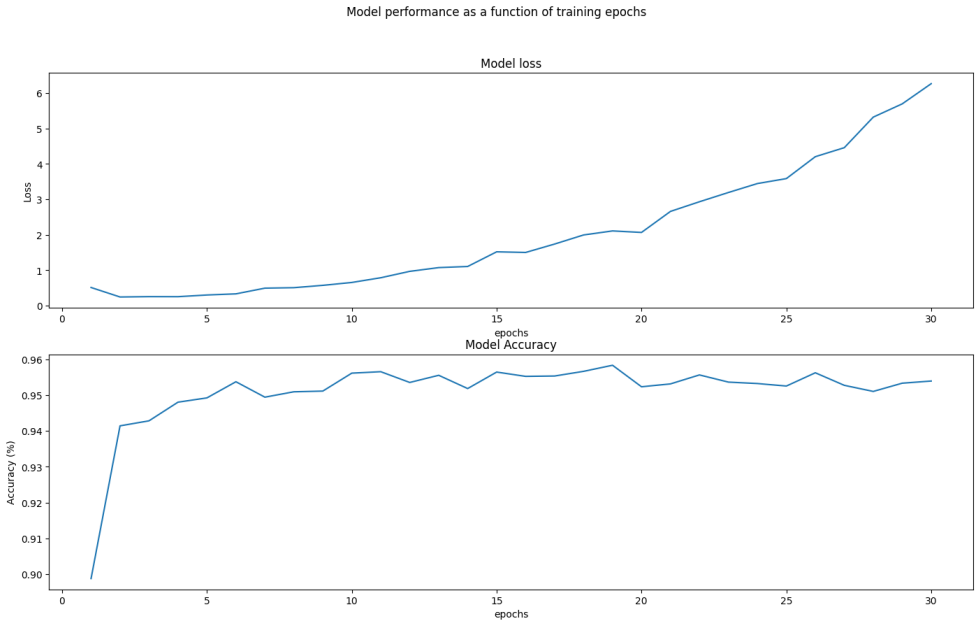

Figure 3: Model performance vs training epochs

Loss is the difference between the true value and the actual value. The model can accumulate loss while also getting the right answer.

Accuracy is the more intuitive metric on this graph. It tells us how often the model gets the right answer where 0% accuracy means the model never gets the right answer while 100% accuracy means the model always gets the right answer. In this series of tests, the model accuracy maxes out at 95.84%, showing that if the model were given 1000 handwritten digits, it would identify 958 correctly and misclassify 42.

As we increase the number of epochs in training, we can see that the model accuracy increases and eventually plateaus at about 95% accuracy. The model loss shows a different trend. Interestingly, as the epochs increase, the loss seems to increase as well exponentially. At first glance, this was confusing. How can the model rack up more losses if it gets to train for more iterations? Shouldn't the model loss decrease with training time? Well, increases in loss as epochs increase could be a sign of overfitting. As the model learns to match the training data too much, it loses its generalizability and when evaluated on other datasets it hasn't seen in training, it can deviate from the expected answer. While a well-trained model without overfitting may assign a handwritten '5' digit with a 0.94 on the node denoting the number 5, an overfit model may assign that same digit a 0.74 due to the digit not resembling its training data perfectly. While 0.74 was likely the largest number across the ten output nodes, the difference between 1 and 0.94 is much less than the difference between 1 and 0.74 and thus the overfit model will register a larger loss even though both models have the same accuracy.

For the model parameters used in this report, six epochs seem to be the sweet spot. Model accuracy is high at 95.38% while model losses are low at around 0.34 ensuring high accuracy while also retaining high levels of generalizability.

8 Project Reflection

While I'm personally happy with the model's performance given my low level of expertise, I do realize that 95.38% accuracy is too low to be used in most commercial applications. Imagine if you deposited a cheque at the bank and one of the digits was interpreted incorrectly, you wouldn't be very pleased. For customer-facing applications I would aim for 99% accuracy as most OCR (optical character recognition) software has an accuracy of 98-99% as suggested by this article: [link].

I've learned a lot by going through this simple TensorFlow exercise, but I want to know more. I want to specifically know more about how models are trained; how gradient descent is performed and what the limitations of artificial neural networks are compared to other types of neural networks such as spiking neural networks. I want to know more about the hardware used to train and run these neural networks, why GPUs are so important and how specialized neuromorphic hardware can accelerate the rate at which we develop and deploy new AI systems. All in due time, I guess.