The effect of hidden layers in a Neural Network when trained

CandNo: 246619

24/02/2023

Abstract

Neural networks are powerful machine learning algorithms which take their inspiration from the human brain. They are able to competently approximate many different kinds of functions which gives them use in many different applications, one such being image recognition. The accuracy of a neural network can vastly outperform traditional programming techniques, but they are difficult to design and train. In this report I will be looking at the effects of using a different amount of hidden layers within a neural network to be able to predict handwritten digits found within the MNIST dataset. Here we show the surprising results that the neural network performed the best when containing no hidden layers and that the accuracy of the classifier degraded with every addition hidden layer which was added.

1 Introduction

MNIST is a popular dataset in the field of machine learning which consists of 60,000 training images and 10,000 testing images of hand written numbers normalised to 28 by 28 pixels. The black and white images have often found as a good first test to many kinds of networks due to the simpleness of the data and many approaches have found near to human level of performance in classification.

In this study I will be looking to find the effects of hidden layers in a Feed Forward Neural Network when predicting the hand written digits in the images from the MNIST dataset. A neural network is an implementation of machine learning that takes inspiration from the workings of the brain. It is comprised of nodes which are modelled to work similarly to how a synapse would fire within a brain. These nodes take inputs which can be modified by the weights and biases associated with the connection between the node and its previous. The resulting product is then passed through to an activation function – this provides an output which is mapped through a specific function from the input. The resulting output will then be fed into the next layer of nodes. It is important to state that each node is connected to each and every node of the previous and next layer.

We can think of the weights and the biases within the network as acting as a long term storage within the network (Russell & Norvig (2010)). By adjusting these values we can effectively teach the network and eventually approximate the relationship between the inputs and outputs to the network.

Neural networks are seen as a black box system – where it can not be entirely known how the hidden layers are working. To find a suitable number of layers is an important part of the process of constructing a neural network and requires a certain level of experimentation. Although there are programmatic methods to figure this out.

My hypothesis is that a network with 2 hidden layers will perform the best. This is based on my research on the dataset and draws from other neural networks that have been built to classify these images (*The mnist database* (n.d.))

2 Methods

The architecture of the network always begins with an input layer of 784 (the amount of pixels within the image flattened) and end with an output layer of 10 (representing each of the possible digits). The amount of hidden layers within the network is then incremented from 0 hidden layers to a maximum of 4 – all consisting of 64 nodes.

I chose to use the ReLU activation function to conduct this experiment as with limited runs I found it to have the best accuracy. The network is trained using Stochastic Gradient Descent with a learning rate set to 0.05. The images are trained in batches of size 64 and each training/testing cycle was carried out over 20 epochs.

When classifying an image the network will produce a number associated with each of the 10 nodes of the output layer, each representing a digit. The max of the probabilities is then taken, which is used to predict the class associated with the image.

3 Results & Discussion

After training over 20 epochs I found that the model with the highest accuracy was the network with no hidden layers. The accuracy converged to 92.25%, but at all times including the very first epoch scored higher than all other models. As you can see in Figure 1, each addition hidden layer added to the network negatively impacted the accuracy, which was a surprising result.

The network with no hidden layers gained the least amount of accuracy over the 20 epochs, it began with an initial accuracy of 89.94%. The network with 4 hidden layers saw the greatest improvement in accuracy and saw a comparably more steady increase over the 20 epochs.

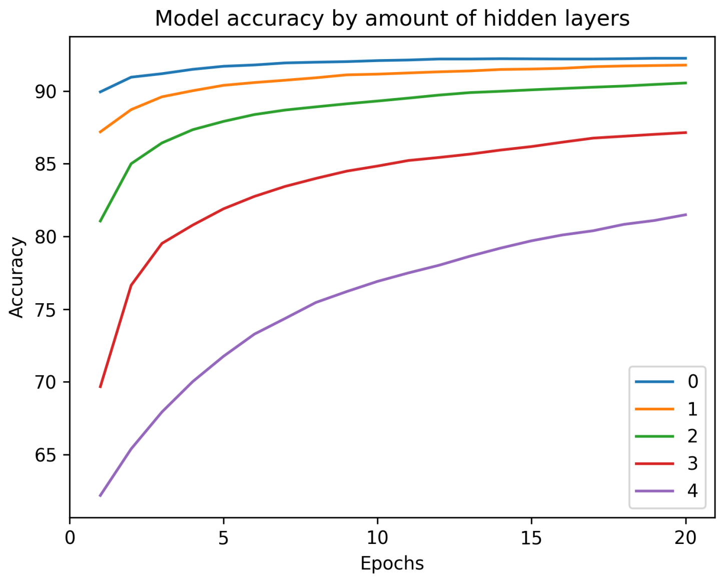

Figure 1

The results of the experiment surprised me and ultimately leads me to believe I may have gone wrong somewhere when constructing the neural network. Common belief is that neural networks with 1 – 2 hidden layers are the most accurate when classifying images in the MNIST dataset.

It is my understanding that the Pytorch nn.Linear function sets up the weights and biases to be initially random but I may be overlooking an important aspect which would explain the performance with no hidden layer.

A more important point would be that is it even possible to classify MNIST accurately with no hidden layer? This would depend whether the dataset was linearly separable. There seems to be conflicting thoughts on this, but my intuition would tell me that the data is not linearly separable. I could not imagine it to be possible to draw a line separating the a '3' and an '8' in the data space for example. With this assumption in mind it should not be possible for a neural network with no hidden layer to predict so accurately using data with is not separable.

References

Russell, S. J. & Norvig, P. (2010), '19'.

The mnist database (n.d.).
URL: *http://yann.lecun.com/exdb/mnist/*

```
import nn_classifier
import numpy as np
import matplotlib as plt

epoch = 20
max_layers = 5
results = []
layers = [784, 10]
hidden_nodes = 32

for x in range(max_layers):
    nn = nn_classifier.nnClassifier(layers, epoch)
    results.append(nn.train_and_test())

    layers.insert(1, hidden_nodes)

print(results)

file = open('result.txt', 'w')
for item in results:
    file.write(str(item) + '\n')
file.close()

from typing import *

import torch
import torch.nn as nn
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import torch.optim as optim


class nnClassifier:

    def __init__(self, layers: List[int], epochs: int):
        self.layers = layers
        self.batch_size = 64
        self.epochs = epochs
        self.linear_layers = []
        self.accuracy = []

        for i in range(1, len(layers)):
            self.linear_layers.append(nn.Linear(layers[i - 1], layers[i]))

    def predict(self, x, linear_layers):
        for layer in linear_layers:
            x = layer(x)
        return x

    def train_and_test(self):
        # dataset class contains data within
        transform = transforms.ToTensor()
        train_dataset = datasets.MNIST('../data', train=True, download=True, tra
        test_dataset = datasets.MNIST('../data', train=False, download=True, tra

        train_loader = DataLoader(train_dataset, batch_size=self.batch_size)
        test_loader = DataLoader(test_dataset, batch_size=self.batch_size)

        parameters = [list(x.parameters()) for x in self.linear_layers]
        optimizer = optim.SGD(parameters[0], lr=0.05)

        loss_fn = nn.CrossEntropyLoss()

        for epoch in range(self.epochs):
            for batch_id, (x, y) in enumerate(train_loader):
                # convert data to a vector - but still with a size of 64 (batch
                x = x.view(-1, 784).float()

                # zero gradients
                optimizer.zero_grad()

                # predict output
                y_hat = self.predict(x, self.linear_layers)

                # calc error
                loss = loss_fn(y_hat, y)
                print(loss)

                # calculate the gradients
                loss.backward()

                # update the parameters as a result of backprop
                optimizer.step()

                # print status
                #if batch_id % 200 == 0:
                    #print(f"{batch_id} / {len(train_loader)}")

            # test epoch
            correct = 0
            total_count = 0
        # don't need any gradients during testing
        # this is purely to test the classifier so far
        # no back prop is needed
        with torch.no_grad():
            for x, y in test_loader:
                # convert data to a vector
                x = x.view(-1, 784).float()

                # predict output
                y_hat = self.predict(x, self.linear_layers)

                # check correctness
                _, pred_label = torch.max(y_hat.data, 1)
# torch.max results the max value and the index of the max
                total_count += x.data.size()[0]  # add the amount of the bat
                correct += (pred_label == y.data).sum()
# for each

        accuracy = correct / total_count * 100
        self.accuracy.append(accuracy.item())

    return self.accuracy
```