Deep Learning Basics: An Overview

Ashray Jain

October 4, 2024

Abstract

This paper provides an overview of fundamental deep learning concepts, with a focus on neural networks. Key topics include neural network architecture, data preprocessing, training methodologies, and common challenges. The implementation of a neural network using Python, Keras, and TensorFlow is discussed, specifically for classifying handwritten digits from the MNIST dataset. Additionally, the network's performance is evaluated on a similar self-made dataset. This discussion is based on a presentation by Ashray Jain at the Institute for Theoretical Physics, Leipzig University.

1 Introduction

Deep learning is a subset of machine learning. It uses multi-layered neural networks to perform tasks such as pattern recognition, classification, and regression among many others. The term "deep" refers to the number of layers in the network. Typically when a neural network has 3 or more hidden layers, it qualifies to be called deep. In the following section neural networks and their constituents are discussed in a more detailed manner.

2 Neural Networks

2.1 Basic Concepts

A neural network is a machine learning system that operates similarly to the human brain. It uses processes that emulate the functioning of biological neurons, working together to recognise patterns, evaluate choices, and make decisions.

A neural network is composed of layers of nodes, also known as artificial neurons. These layers include an input layer, one or more hidden layers, and an output layer (see figure 1). Each node is interconnected and has its own weight and bias. It also has a threshold. When a node's output exceeds its threshold, it gets activated and transmits data to the next layer. If it doesn't, no data is forwarded to the subsequent layer.

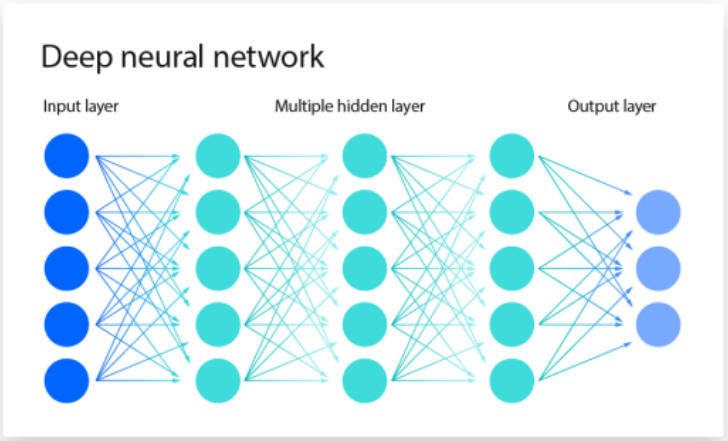


Figure 1: Structure of a Deep Neural Network, illustrating the input layer, 3 hidden layers, and an output layer. Each node in the network represents a neuron, connected by synapses that carry the data forward. It can be thought that whenever data passes through a connection it is scaled by the weight associated with the synapse and whenever data reaches a node it is offset by addition (subtraction in case of negative bias) of the bias associated with that node.[4]

3 Mathematical Foundations

3.1 Linear and Non-Liner Mapping

Now to begin understanding how exactly these networks work, we need to begin by understanding something called "Linear Mapping". In a neural network input values are given at the input layer. Imagine input as a vector $\vec{x}$, with it having dimensions that are equal in number to the number of nodes in the input layer. The input is given by passing each component of this vector to the corresponding input node. In

neural networks, *linear mapping* transforms input values using weighted sums and biases, described by affine transformations. For example, with inputs $\vec{x} = (x_1, x_2)^T$ and weights W, the output $\vec{y}$ is given by $\vec{y} = W\vec{x}+\vec{b}$. This can be further transformed to $z = W'^T\vec{x} + d'$ with additional weights W' and bias d. This is shown in figure 2.

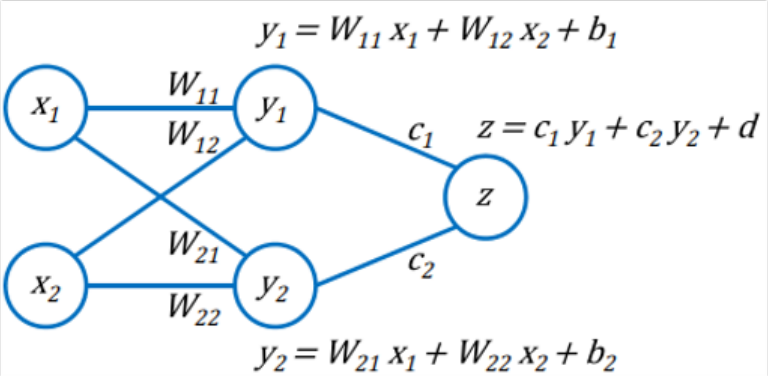


Figure 2: A neural network with two input nodes (x_1 and x_2) undergoing a linear transformation with weights W_{ij} and biases b_i to produce outputs y_1 and y_2. These outputs are then linearly combined using weights c_i and an additional bias d to produce the final output [1]

Now, all this is good and well, but the problem is that with just linear mapping, the network can only approximate linear functions. This is a severe restriction on the diversity of problems that a neural network can handle. To deal with this issue we use non-linear mapping. *Nonlinear mapping* involves applying activation functions like ReLU, sigmoid, or tanh (See figure 3) to the outputs at each node to capture more complex patterns. This process enables neural networks to model a wider range of phenomena.

3.2 Universal Approximation Theorem

This theorem states that a neural network with a single hidden layer can approximate any continuous, multi-input/multi-output function with arbitrary accuracy.

The basic idea is that the hidden neurons allow generation of augmented activation functions with arbitrary gradients and y-axis offsets. These can then be added

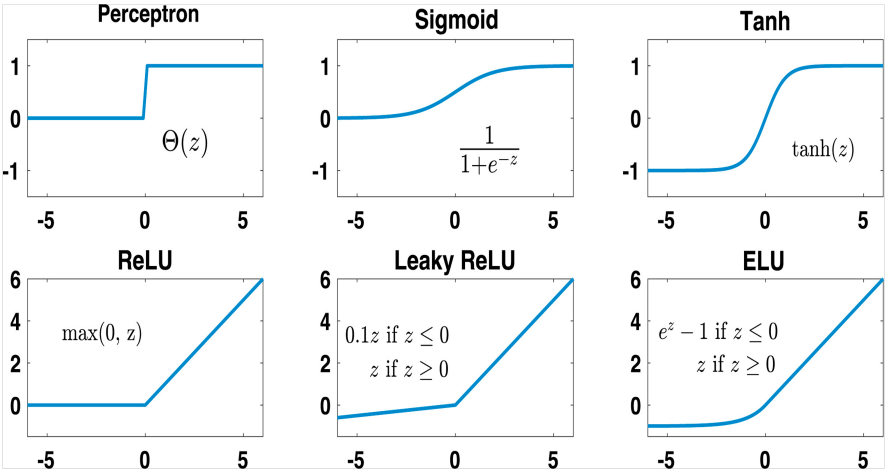


Figure 3: Different examples of activation functions. In modern times, activation functions like Tanh are avoided because of the vanishing gradient problem (discussed in challenges section) and those like ReLU are used instead. [1]

together to approximate any arbitrary function. For a nice proof of this theorem, refer to chapter 4 of Nielsen's free online book (Nielsen, 2015) [2].

Modern neural networks generally contain multiple hidden layers to learn more complex features.

4 Training Neural Networks

4.1 Data Preprocessing

Numerical stability is vital for successfully constructing a neural network model. When the input data $\vec{x}$ have a wide range of magnitudes and vary significantly among the individual components x_i, it complicates the adjustment of network parameters. Therefore, meticulous preparation and pre-processing of the input data steps are crucial for effective model development. Techniques include zero-centering, normalisation, and logarithmic transformation among others. Refer to table 1.

4.2 Epochs and Minibatches

Network training constitutes providing the input data to the network and checking its prediction against expected result. If you have a set of training data, it usually

Table 1: Data Pre-Processing

Type of Processing	**Data Type**	**Formula**
Zero-centering	Non-zero mean	$x_i - \langle x_i \rangle$
Order of magnitude	Different magnitudes	$x_i' = \frac{x_i - \langle x_i \rangle}{\sigma_i}$
Logarithm	Large fluctuations	$x_i' = \log x_i$

constitutes the input vectors $\vec{x_i}$ and a label associated with each of them y_i. The label carries the expected result. It is important to note that the labels are never fed into the network, only the vectors $\vec{x_i}$. Network training is conducted in an iterative manner, utilising the available training data repeatedly and as efficiently as possible. Two key terms are essential to describe this process: epoch and batch.

4.2.1 Epoch

An epoch in network training refers to the complete use of all training data once. Typically, network training involves many epochs, meaning the entire training data set is reused multiple times.

4.2.2 Batch (or Minibatch)

Using all training data in each iteration becomes inefficient. The solution is to use a randomly selected subset, called a 'minibatch' or 'batch'. This is also the reason for a relatively high accuracy of the network model ($\sim$90%) after only the 1st epoch of training as seen later in the results section. Batch sizes are often powers of 2, such as 32.

4.3 Parameter Initialisation

To start with the training process we need the network to be able to generate predictions as already discussed in the previous section. There needs to be some choice of parameters set in the network for this. At this point, we are not interested in them being optimal values, absolutely any choice would do. When selecting these initial values the main goal is to break the symmetry of the weight parameters. This prevents different network nodes from learning identical mappings instead of diverse ones.

Symmetry breaking for the initial weights W is done using random numbers from a uniform $[-s, s]$ or a Gaussian distribution with a mean of 0 and a standard

deviation σ. The bias parameters $\vec{b}$ are typically set to zero.

4.4 Cost Function

The cost function (also called loss function), hereon referred to as L, is central to the optimisation process and network modeling. It provides a global measure of the network's prediction quality. During training, thousands or millions or more parameters need to be correctly adjusted. Successful optimisation involves reaching or closely approaching a good local minimum on the parameter hypersurface (this is the cost function plotted against all the, possibly millions of, parameters). Similar to a multi-parameter fit, the cost function provides a single scalar measure per iteration of how well the parameters have been adjusted. The minimum of this function indicates a sufficiently good local minimum has been achieved, which must be verified by evaluating the network's predictions in each project. Two common cost functions are "Cross Entropy" and "Mean Squared Error" also called "MSE". Let's try to understand cost functions better by considering example of the MSE function.

4.4.1 Mean Squared Error (MSE) Cost Function

The MSE measures the average of the squares of the errors, i.e., the differences between the predicted values $f(x)$ and the expected target values $y(x)$.

Mathematically, for a set of input values x, network predictions $f(x)$, and target values $y(x)$, the MSE is defined as:

$$\text{MSE} = \mathbb{E}\left[(f - y)^2\right] \tag{1}$$

The objective is to find the optimum setting of the network parameters such that the MSE becomes minimal, indicating that the predictions closely match the target values.

4.5 Backpropagation

Backpropagation is a method used to calculate the gradients of the cost function with respect to the network parameters. It works by propagating the error backward through the network.

This involves calculating the partial derivatives of L with respect to W and b using the chain rule:

$$\frac{\partial L}{\partial W} = \frac{\partial L}{\partial z_3} \cdot \frac{\partial z_3}{\partial z_2} \cdot \frac{\partial z_2}{\partial z_1} \cdot \frac{\partial z_1}{\partial W} \tag{2}$$

$$\frac{\partial L}{\partial b} = \frac{\partial L}{\partial z_3} \cdot \frac{\partial z_3}{\partial z_2} \cdot \frac{\partial z_2}{\partial b} \tag{3}$$

Here z values represent the weighted sums of inputs before applying the activation function. For a given layer l, $z^{(l)}$ is computed as:

$$z^{(l)} = W^{(l)} a^{(l-1)} + b^{(l)}$$

where $W^{(l)}$ is the weight matrix, $a^{(l-1)}$ are the activations from the previous layer, and $b^{(l)}$ is the bias vector. The z values are then passed through an activation function to produce the activations for the current layer.

Since the calculation of these derivatives is performed in the opposite direction of the forward pass, this procedure is known as backpropagation.

4.6 Gradient Descent

Gradient descent is an optimisation method used to minimise the cost function L. It involves adjusting the network parameters in the direction opposite to the gradient of L with respect to those parameters.

Stochastic Gradient Descent (SGD) is a variant where a minibatch of training data is used in each optimisation step, which introduces a higher variance in the resulting gradients but allows for multiple parameter updates per epoch. This can sometimes help escape from local minima (the goal is to find a good global minima).

The backpropagation algorithm computes the gradient of the cost function, which is used by gradient descent to update the network's parameters iteratively. The learning rate determines the step size in this optimisation process.

4.7 Learning Rate

In the process of finding cost function minima and thereby optimising the network, we get the direction of parameter adjustment by the opposite gradient but the step size (α) used for going from iteration t to t+1 is called the learning rate.

$$W_{t+1} = W_t - \alpha \mathbb{E}\left[\frac{\partial L}{\partial W}\right]_t \tag{4}$$

$$b_{t+1} = b_t - \alpha \mathbb{E}\left[\frac{\partial L}{\partial b}\right]_t \tag{5}$$

4.8 Learning Strategies

The learning rate significantly affects whether and how quickly a good local minimum of the parameter hypersurface is reached. If the learning rate is too low, the minimum may never be reached, while if it is too high, a good local minimum might be missed. See figure 4.

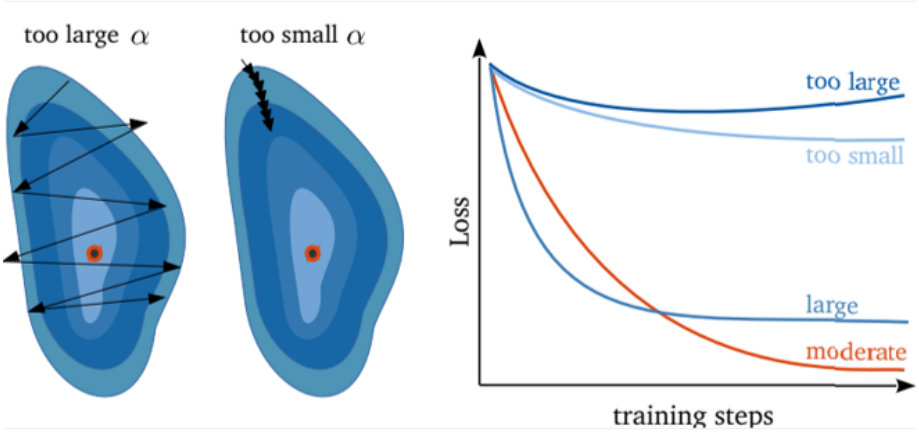


Figure 4: On the left is the contour plot of the cost function (blue) with the minimum marked in red, illustrating the role of learning rate in the process of reaching the minimum. On the right is a plot of the cost function against training steps for various learning rates. [1]

For this reason learning rates are often adjusted during network training. Initially, larger step sizes are used to explore the starting region on the parameter hypersurface. In later stages of training, the learning rate is reduced to allow for finer examination of the parameter hypersurface. These setups for learning rate adjustment are called "Learning Strategies". There are several different of these, each offering their own set of advantages and disadvantages. Few of these are Adagrad, RMSprop, Momentum and Adam.

5 Application: MNIST Dataset

With everything covered so far, it is finally time to stitch it all together and make our own neural network. A popular choice for problem to solve comes from the MNIST

dataset.

The MNIST dataset is a large collection of handwritten digits commonly used for training various image processing systems. It contains 60,000 training images and 10,000 testing images, each representing a digit from 0 to 9. The images are grayscale and size-normalised to 28x28 pixels, pixel values range from 0 to 255, implying 256 gray nuances making it a standard benchmark for evaluating machine learning algorithms. An example from this dataset is shown in figure 5

The goal is to find a model which recognises and distinguishes between the ten handwritten digits (0-9).

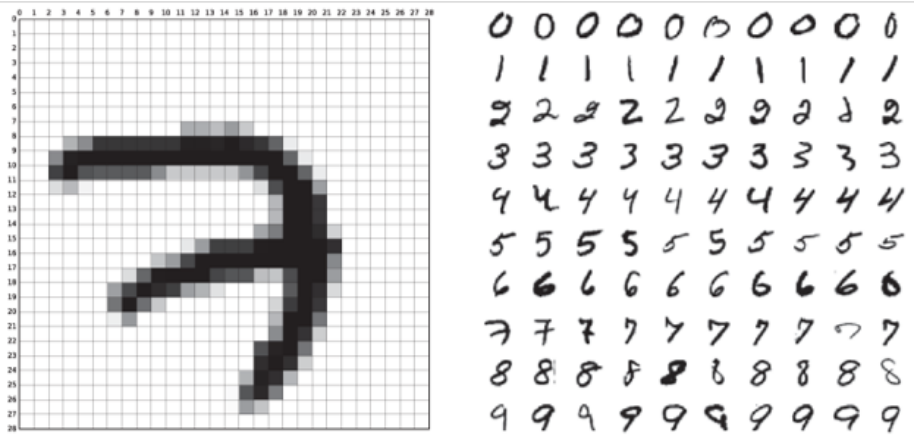


Figure 5: Example data from the MNIST dataset [5].

5.1 Python Code

The python code presented here is an unabridged version with slight modifications by the author of this write-up and is based on the code in the supplementary notebook available with reference [3]. The modifications mostly pertain to changing how the results are presented and plotted.

The code is accompanied with appropriate commentary to aid the comprehension of purpose of the different parts.

```
from __future__ import print_function
import keras,sklearn
# suppress tensorflow compilation warnings
import os
os.environ['KMP_DUPLICATE_LIB_OK']='True'
```

```
import tensorflow as tf
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'
import numpy as np
seed=0
np.random.seed(seed) # fix random seed
#tf.set_random_seed(seed) #Broken due to update, new usage in next line
tf.random.set_seed(seed)
import matplotlib.pyplot as plt
#Using TensorFlow backend.




#!!! STEP 1: Load and process the data!!!
from keras.datasets import mnist
print('Data import complete!')

# input image dimensions
num_classes = 10  # 10 digits

img_rows, img_cols = 28, 28  # number of pixels

# the data, shuffled and split between train and test sets
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
# X_train here would become a 3D numpy array with lengths as follows = (num. samples * img rows * img cols) 60000*28*28

# reshape data, depending on Keras backend
X_train = X_train.reshape(X_train.shape[0], img_rows * img_cols)
X_test = X_test.reshape(X_test.shape[0], img_rows * img_cols)

# cast floats to single precesion
X_train = X_train.astype('float32')
X_test = X_test.astype('float32')

# rescale data in interval [0,1]
X_train /= 255
X_test /= 255
print('Data values rescaling in range [0,1] successful')


# look at an example of data point
print('an example of a data point with label', Y_train[20])
plt.matshow(X_train[20, :].reshape(28, 28), cmap='binary')
```

```
plt.show()

# convert class vectors to binary class matrices
Y_train = keras.utils.to_categorical(Y_train, num_classes)
Y_test = keras.utils.to_categorical(Y_test, num_classes)
# This converts each label, for example 7 into a one hot encoding compatible numpy array, for example for 7 it
# would be [0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]

print('X_train shape:', X_train.shape)
print('Y_train shape:', Y_train.shape)
print()
print(X_train.shape[0], 'train samples')
print(X_test.shape[0], 'test samples')
print('Data in place. Ready to create the NN.')



#!!! STEP 2: Define the model and its architecture!!!
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D


def create_DNN():
    # instantiate model
    model = Sequential()
    # add a dense all-to-all relu layer
    model.add(Dense(400, input_shape=(img_rows * img_cols,), activation='relu'))
    # add a dense all-to-all relu layer
    model.add(Dense(100, activation='relu'))
    # apply dropout with rate 0.5
    model.add(Dropout(0.5))
    # soft-max layer
    model.add(Dense(num_classes, activation='softmax'))

    return model
print('Network nodes activation choice set to ReLU')
print('Model architecture created successfully! Choose Cost function now.')



#!!!Step 3: Choose the Optimizer and the Cost Function!!!
```

```
93  def compile_model(optimizer=keras.optimizers.Adam()):
94      # create the mode
95      model=create_DNN()
96      # compile the model
97      model.compile(loss=keras.losses.categorical_crossentropy,
98                    optimizer=optimizer,
99                    metrics=['accuracy'])
100     return model
101 print('Cost function choice set to Cross-Entropy')
102 print('Model compiled successfully and ready to be trained.')
103
104
105
106
107 #!!!Step 4: Train the model!!!
108 # training parameters
109 batch_size = 64
110 epochs = 10
111 print('batch_size = 64\nepochs = 10')
112
113 # create the deep neural net
114 model_DNN=compile_model()
115
116 # train DNN and store training info in history
117 history=model_DNN.fit(X_train, Y_train,
118           batch_size=batch_size,
119           epochs=epochs,
120           verbose=1,
121           validation_data=(X_test, Y_test))
122 print('Training complete! NN ready to be tested.')
123
124
125
126 #!!!Step 5: Evaluate the Model Performance on the Unseen Test Data
        !!!
127 # evaluate model
128 score = model_DNN.evaluate(X_test, Y_test, verbose=1)
129
130 # print performance
131 print()
132 print('Test loss:', score[0])
133 print('Test accuracy:', score[1])
134
135 # look into training history
136
137 # Create x-axis labels starting from 1
```

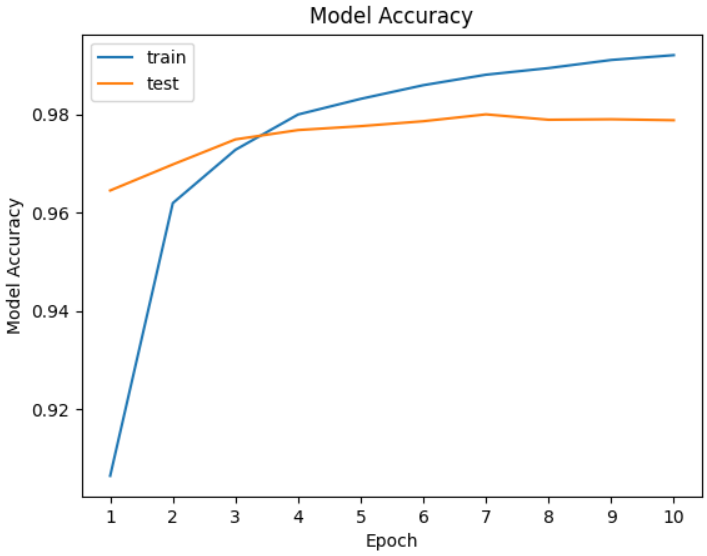
```
x_labels = list(range(1, len(history.history['accuracy']) + 1))

# summarize history for accuracy
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.xticks(ticks=range(len(history.history['accuracy'])), labels=
    x_labels)
plt.ylabel('Model Accuracy')
plt.xlabel('Epoch')
plt.legend(['train', 'test'], loc='best')
plt.title('Model Accuracy')
plt.show()

# summarize history for loss
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.xticks(ticks=range(len(history.history['loss'])), labels=
    x_labels)
plt.ylabel('Model Loss')
plt.xlabel('Epoch')
plt.legend(['train', 'test'], loc='best')
plt.title('Model Loss')
plt.show()
```

5.2 Results

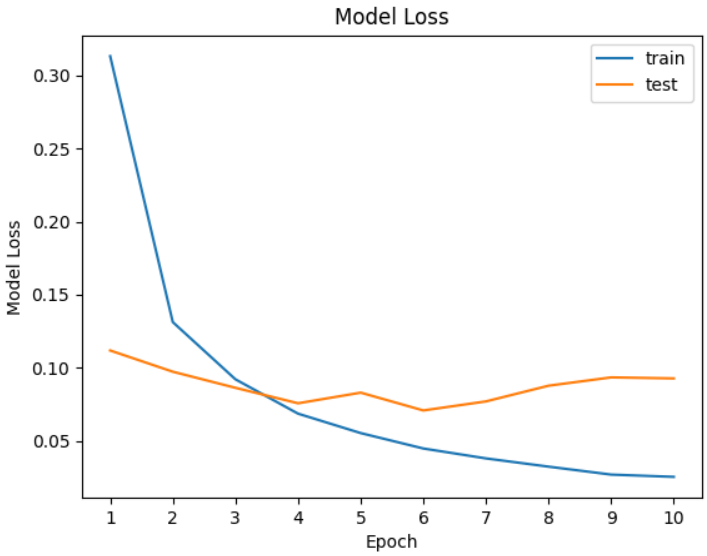
5.2.1 MNIST dataset

The performance of the model is evaluated based on the accuracy of classified images and the cost function after each epoch. The following images (figure 6) presents a compilation of these results.

The model is also tested against test data prepared by the author of this write-up, motivated by a general curiosity. These results are compiled in the next section.



(a) Model Accuracy vs Epoch



(b) Model Loss vs Epoch

Figure 6: (a) Shows fraction of correctly classified images with each subsequent epoch of training. This is for MNIST training and test dataset. It can be seen that the accuracy improves as the training progresses. (b) Shows the same but for the loss function value. This is for MNIST training and test dataset. It can be seen that loss decreases as the training progresses.

5.2.2 My dataset

The performance of the model is evaluated based on the same criteria as the last section. The data prepared by the author was obtained as described in the next paragraph.

To prepare this dataset, images of 5 digits handwritten with a sharpie on a piece of blank paper were clicked with a smartphone. These images were then cropped in a square aspect ratio and magnified to roughly match the size of the digits in the MNIST dataset. A gray-scale filter was applied to make the pictures monochrome. These images can be seen in figure 7.

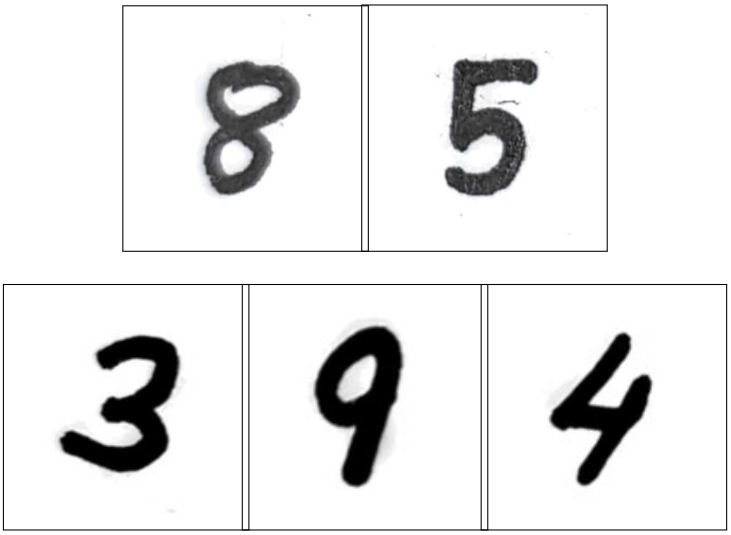


Figure 7: Handwritten digit images used to create a custom dataset for model evaluation. The digits were written with a sharpie, captured using a smartphone, and processed to make them monochrome.

These images were then passed to ChatGPT-4o [6] with the following prompt:

*"I want you to take these pictures and put a uniform 28×28 grid over them. Now treat each grid box as a pixel and associate with it a number between and including 0 to 255. 0 corresponds to white and 255 to black. Once you do this give me the data as a numpy array of shape 28*28"*

The data collected this way can be checked at reference [7]. To visualise the data in a way similar to figure 5, the following python code can be used. The resulting images are shown in figure 8.

```python
import matplotlib.pyplot as plt
import pandas as pd

# Read the CSV file into a DataFrame
df = pd.read_csv('Path\to\Pixel_Data.csv', header=None)

# Convert the DataFrame to a numpy array
my_number = df.to_numpy()

# Ensure the shape is 28x28
my_number = my_number.reshape((28, 28))

print(my_number)

plt.matshow(my_number.reshape(28, 28), cmap='binary')
plt.show()
```

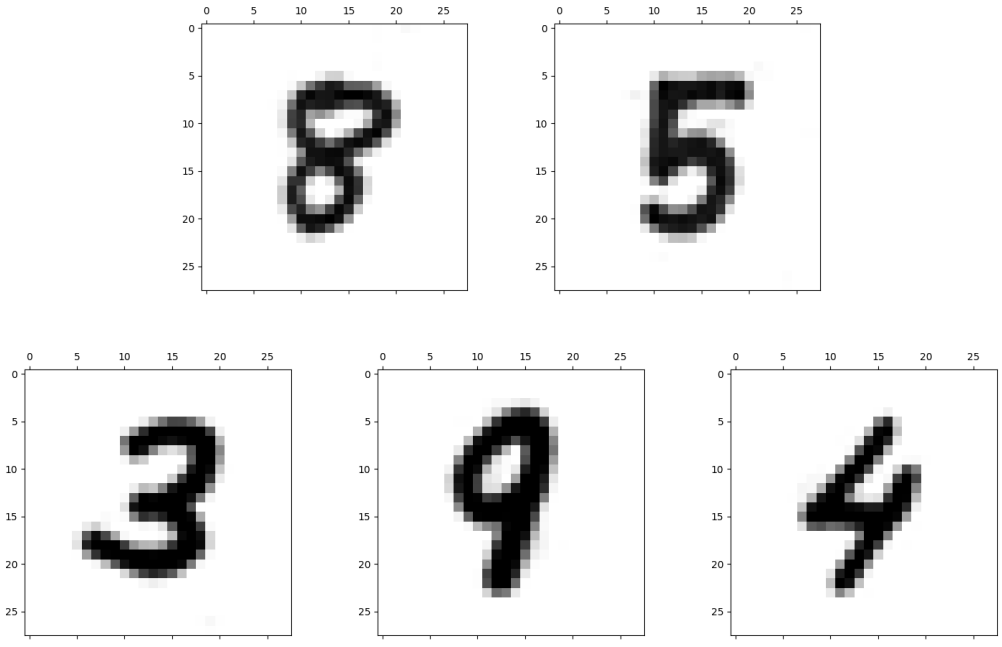


Figure 8: Handwritten digit images from figure 7 processed to match the MNIST dataset specifications.

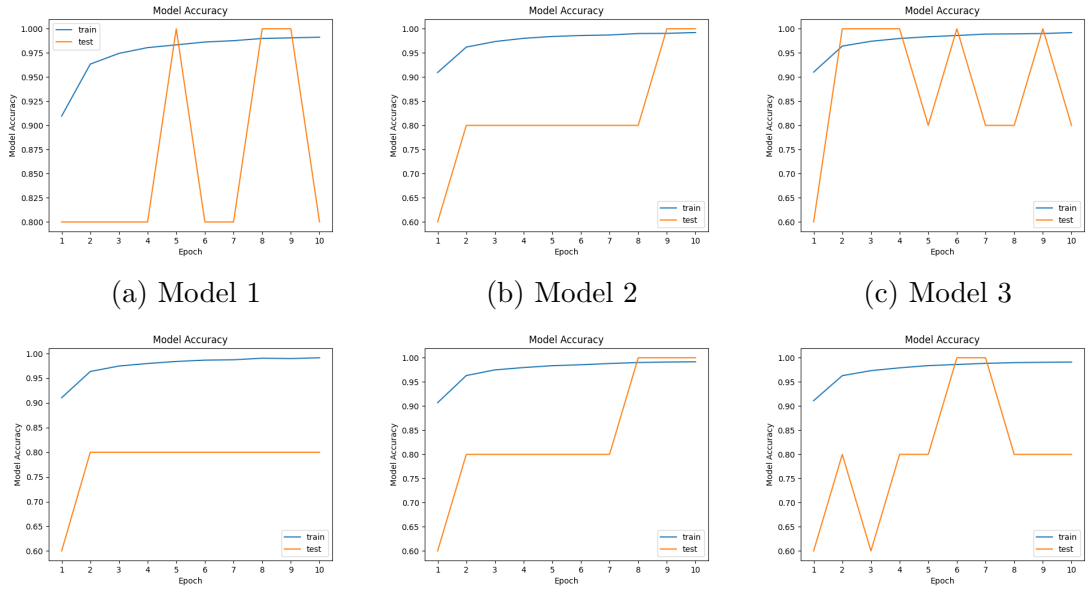
To test the data, the same Python script is used except a single difference. A

substitution in variable definition is made which causes the MNIST test dataset to be replaced by the author's test dataset. The portion of code modified to achieve this is shown below.

```
# Preceding code remains the same
#!!! STEP 1: Load and process the data!!!
import pandas as pd
from keras.datasets import mnist

# List of paths to the CSV files
csv_paths = [
    'Path\to\Pixel_Data1.csv',
    'Path\to\Pixel_Data2.csv',
    'Path\to\Pixel_Data3.csv',
    'Path\to\Pixel_Data4.csv',
    'Path\to\Pixel_Data5.csv'
]

# Function to read a CSV file and convert it to a 28x28 numpy array
def read_csv_to_array(file_path):
    df = pd.read_csv(file_path, header=None)
    return df.to_numpy().reshape(28, 28)

# Read each CSV file and store the resulting arrays in a list
arrays = [read_csv_to_array(path) for path in csv_paths]

# Stack the arrays into a single numpy array of shape (5, 28, 28)
X_my_numbers = np.stack(arrays)

# Verify the shape
print(X_my_numbers.shape)  # Should print (5, 28, 28)

Y_my_numbers = np.array([3, 8, 9, 5, 4])
print('Data import complete!')



# input image dimensions
num_classes = 10  # 10 digits

img_rows, img_cols = 28, 28  # number of pixels

# the data, shuffled and split between train and test sets
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
# X_train here would become a 3D numpy array with lengths as follows
    = (num. samples * img rows * img cols) 60000*28*28
```

```
42
43 X_test = X_my_numbers
44 Y_test = Y_my_numbers
45
46 # reshape data, depending on Keras backend
47 X_train = X_train.reshape(X_train.shape[0], img_rows * img_cols)
48 X_test = X_test.reshape(X_test.shape[0], img_rows * img_cols)
49
50 # Following code remains the same
```

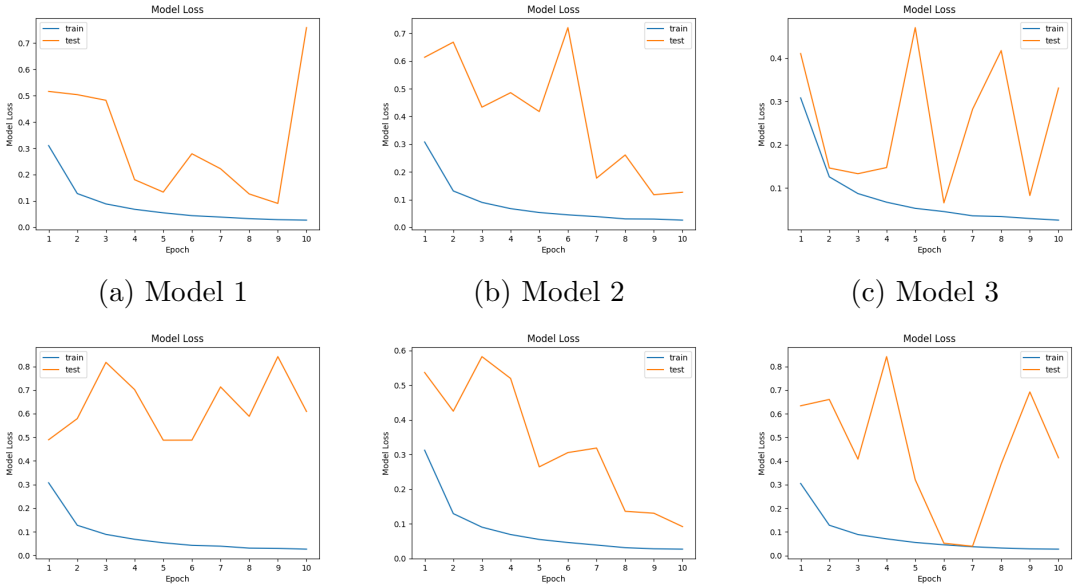
The dataset was tested on and during six independent network training instances (here called Model 1, 2, 3, 4, 5 & 6) and the following images (figures 9 & 10) present a compilation of the results.



(a) Model 1 (b) Model 2 (c) Model 3

(d) Model 4 (e) Model 5 (f) Model 6

Figure 9: Shows accuracy vs epoch number for 6 different instances of trained network model. A general trend of improving accuracy with epoch number can be observed.



(a) Model 1 (b) Model 2 (c) Model 3

(d) Model 4 (e) Model 5 (f) Model 6

Figure 10: Shows loss vs epoch number for 6 different instances of trained network model. A general trend of decreasing loss with epoch number can be observed.

The results indicate an accuracy that is reasonably acceptable provided that the dataset has not been processed in the same standardised way as the MNIST dataset and that it has not been denoised or processed to remove random ink marks on the page. The error can be explained by random statistical flukes provided that dataset is constituted by a very small number, only five images. It could also be that the model is consistent in inaccurately categorising the same one or two images in all the independent runs as these images may have some feature that makes them susceptible to an ambiguous interpretation. Further investigation towards validity of any of these hypothesis lies outside the scope of this paper.

6 Challenges in Neural Networks

Neural networks, despite their remarkable success in various applications, face several significant challenges. Few of them are discussed below.

- The vanishing and exploding gradient problems occur during backpropagation in neural networks. The vanishing gradient problem arises when the gradients

of the loss function with respect to the parameters become exceedingly small, particularly in deep networks, causing minimal updates to the earlier layers and impeding learning. Conversely, the exploding gradient problem occurs when these gradients grow exponentially, leading to very large updates and instability in the network. Both issues are exacerbated by activation functions like sigmoid and tanh, which can compress input values to a limited range. Mitigation strategies include using Rectified Linear Unit (ReLU) activation functions, batch normalisation, and careful weight initialisation methods such as Xavier or He initialisation to maintain gradient magnitudes within a manageable range and ensure stable training.

- Overfitting occurs when a neural network model learns not only the underlying patterns in the training data but also the noise and random fluctuations. This leads to excellent performance on the training dataset but poor generalisation to new, unseen data. Overfitting is a significant challenge in neural network training, particularly when the model is complex or the dataset is small. Mitigation strategies include using techniques such as regularisation (e.g., L1 or L2 regularisation), dropout, data augmentation, and early stopping. Additionally, cross-validation can help in selecting the optimal model complexity and hyperparameters to ensure better generalisation.

7 Conclusion

Deep learning and neural networks offer powerful tools for solving complex problems in various fields. However, careful consideration of network architecture, training methodologies, and data preprocessing is essential for achieving optimal performance.

References

[1] Martin Erdmann, Jonas Glombitza, Gregor Kasieczka, and Uwe Klemradt, *Deep Learning for Physics Research*, RWTH Aachen University, University of Hamburg, July 2021. doi: 10.1142/12294

[2] Nielsen, M. (2015). *Neural Networks and Deep Learning*. Retrieved from `http://neuralnetworksanddeeplearning.com/chap4.html`

[3] P. Mehta, M. Bukov, C.-H. Wang, A.G.R. Day, C. Richardson, C.K. Fisher, D.J. Schwab, *A high-bias, low-variance introduction to Machine Learning for*

physicists, Physics Reports, Vol. 810, pp. 1-124, 2019, ISSN 0370-1573, https://doi.org/10.1016/j.physrep.2019.03.001.

[4] IBM. (n.d.). Neural Networks. Retrieved from https://www.ibm.com/topics/neural-networks

[5] MNIST example data image retrieved from https://www.mdpi.com/2076-3417/9/15/3169

[6] OpenAI. *ChatGPT: Chatbot Language Model*, https://www.openai.com/chatgpt, (accessed August 2024).

[7] Ashray Jain. *Image Pixel Data*, DNN_MyData_HandwritingRecognition, https://drive.google.com/drive/folders/1yurlxJCtEr7Y6hPDZ9SrmLkAzl93izi0?usp=drive_link, (accessed August 2024).