Group Assignment 2 - CS 985/CS 988

Group AH - Peter Janik (201979128), Inno Mvula (201973944), Thom Reynard (201977555)

April 8th 2020

I. Kannada-MNIST Classification Problem

The *Kannada-MNIST Classification Problem* is a widely renowned machine learning problem focused on determining values from handwritten digits. Developed in 1999 by the Modified National Institute of Standards and Technology, it became the foundation for many of the concepts used in today's machine learning. The Kannada-MNIST dataset uses Kannada digits, a language spoken predominantly by people of Karnataka in southwestern India, as opposed to the arabic digits used in the original dataset, these commonly used around the world.

In this assignment, we were challenged to build and train a machine learning model using Keras to predict the value of each handwritten digit. Each image is produced on a 28x28 grid resulting in 784 pixels for each image, with values in the individual pixel columns ranging from 1 to 255 indicating the darkness of the particular pixel, with higher values indicating darker pixels. [1] [2] [3]

I.I Approach Used

The approach used to complete the classification of the Kannada-MNIST handwritten digits applied a class of deep learning neural networks called Convolutional Neural Networks (CNN). CNNs are powerful deep learning networks and are the preferred deep learning method for image classification. Exploring historical Kaggle scores for the non-Kannada MNIST dataset provides significant indication that the aforementioned CNN model can be considered most likely to be appropriate for the dataset. The combination of data augmentation with convolutional, pooling and dense layers form the foundation of a strong, efficient and accurate machine learning model for the Kannada-MNIST dataset (see Figure 1 and 2) [4]

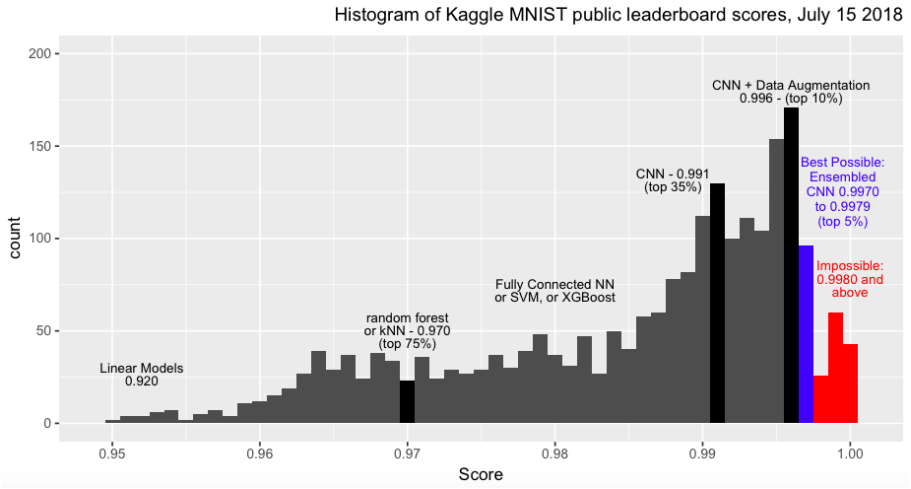

Figure 1: Historical Breakdown of Kaggle Scores for MNIST data [4]

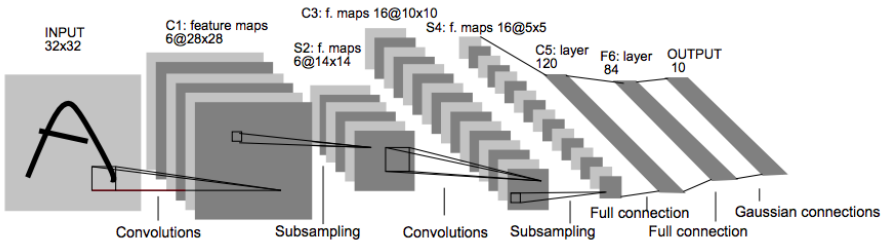

Figure 2: CNN Model Breakdown [4]

II Code Breakdown

II.I Packages

Multiple machine learning models were employed to determine which one provides the most statistically significant results. Ultimately, the packages used for the model included: NumPy, Pandas, Matplotlib, as well as seaborn and sklearn. In addition to the basic packages, Keras and TensorFlow were included to provide the base for the machine learning aspect of the code.

```
#Loading GPU for faster training time
%tensorflow_version 2.x
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))


#import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from matplotlib.pyplot import figure
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow import keras
from google.colab import files
import io
from keras.preprocessing.image import ImageDataGenerator


from numpy.random import seed
seed(2)
tf.random.set_seed(2)
```

Following the import of packages, the data was imported using the readcsv function within Pandas to Jupyter for data analysis and to Google Colab to allow for the use of Google's server capabilities when working with TensorFlow, allowing the code to run faster with the added technical resources. All in all, two files were uploaded to both systems, the [60000, 786] sized training file, and the [10000,785] sized test file which was used for evaluation.

```
import os
os.environ['PYTHONHASHSEED'] = str(2)

#Uploading training Data
uploaded = files.upload()
#Saving training data for use in analysis
train_data = pd.read_csv(io.BytesIO(uploaded['training.csv']))
#Uploading Test data
uploaded = files.upload()
#Saving test data to test our model and make predictions
test_data = pd.read_csv(io.BytesIO(uploaded['test.csv']))
#Observation of shape and size of training data set
```

```
train_data.shape
#Observation of Training Dataset
train_data.info()
```

II.II Data Analysis

Both datasets were analysed for missing values that may cause interpretation difficulties; none were found, concluding the data is clean and ready for analysis. Splitting the data into target and predictors and further distribution analysis revealed an equal 6000 split amongst the ten digits present in the dataset. As the Keras conv2D function works best with three-dimensional data, the data is reshaped using the reshape function within Python.

```
train_data.isnull().sum().sum()
train_data.keys()
train_data.iloc[:, 1]

#Splitting Target and Predictors
X, Y = train_data.iloc[:, 2:].values/255, train_data.iloc[:, 1].
    values
#Reshaping dataset as CNN works best with higher dimensions
x = X.reshape(X.shape[0],28,28,1)
#Distribution of each class
import collections
print(collections.Counter(Y))
```

After the data is reshaped and split into a target and its predictors, the train-test split function is run that splits the training data further into a new set of training and test data, that will be used to determine model efficiency and accuracy. In this example, 10 percent of the train data now becomes a test data and following a distribution check, the data remains evenly distributed amongst the target digits with each digit being represented by 5400 independent rows. A further 10 percent split is taken to create additional validation data with now 48600 values being present in the validation training set, while 5400 values make up the validation test set, which will be used in Keras for model completion.

```
#Creating the Training and Test set from data
X_train_full, X_test, Y_train_full, Y_test
    = train_test_split(x, Y, test_size = 0.10, random_state = 42,
    stratify = Y)
#Splitting training data into a train set and validation set
X_train, X_valid, Y_train, Y_valid
    = train_test_split(X_train_full, Y_train_full, test_size =
    0.10, random_state = 42, stratify = Y_train_full)
#Exploring the number of observations/rows in each Split
len(X_train_full), len(Y_train_full), len(X_train), len(Y_train),
    len(X_valid), len(Y_valid)

#observation of one image in the MNIST data array
some_digit = X_train[1]
some_digit_image = some_digit.reshape(28, 28)
plt.imshow(some_digit_image, cmap ='binary')
plt.axis('off')
plt.show
```

II.III Model Creation

Using the Keras Sequential API, we construct the machine learning model which will be firstly tested on our split training set to determine efficiency and accuracy; following this on the test data to produce final results which will be compared to actual figures.

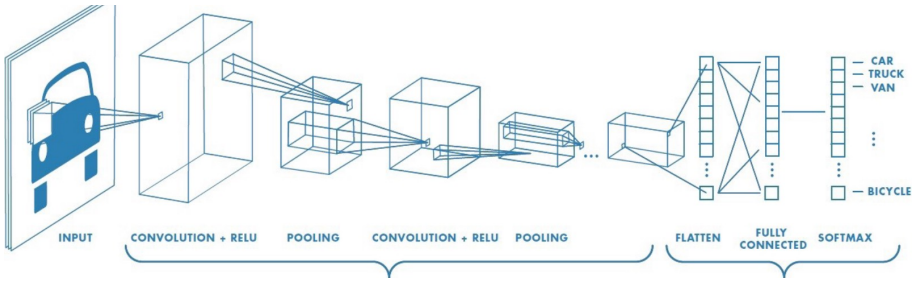

Figure 3: Visual Breakdown of Convolution Neural Network [5]

This specific model is built using convolutional layers stacked upon one another, increasing model accuracy. The idea of convolution is common is image differentiation and used widely in computer vision, specifically with problems such as the one posed by the Kannada-MNIST dataset. According to Jason Brownlee at MachineLearningMastery, "a convolution is the simple application of a filter to an input that results in an activation, [where] repeated application of the same filter to an input results in a map of activation's called a feature map, indicating the locations and strength of a detected feature in an input, such as an image" [6]. There are several activation techniques available to users. The use of these different techniques allows users to sometimes significantly lower the number of parameters the model has to estimate, leading to faster processing and more efficient learning.

```
#Creating/Building the model using the sequential API
model = keras.Sequential()

#We then build the Convolutional layers. Flatten layers role is to convert each input into a 1D array
model.add(keras.layers.Conv2D(64, 3, activation = "relu", padding = "same", input_shape = X_train[0].shape))
model.add(keras.layers.MaxPooling2D(2))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Conv2D(128, 3, activation = "relu", padding = "same"))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Conv2D(128, 3, activation = "relu", padding = "same"))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.MaxPooling2D(2))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Conv2D(256, 3, activation = "relu", padding = "same"))
```

```
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Conv2D(256, 3, activation = "relu", padding = "same"))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.MaxPooling2D(2))
model.add(keras.layers.Flatten())

#we then add a dense hidden layer with 300 neurons, using the relu activation function
model.add(keras.layers.Dropout(rate = 0.5, seed = 2))
model.add(keras.layers.Dense(900, activation = "relu"))

#finally we add a dense output layer with 10 neurons, one for class
model.add(keras.layers.Dropout(rate = 0.5, seed = 2))
model.add(keras.layers.Dense(10, activation = "softmax"))
```

For the use of this project, we determined that the use of the Rectified Linear Unit as an activation technique was most appropriate. The ReLU function "is a piece-wise linear function that will output the input directly if is positive, otherwise, it will output zero [having] become the default activation function for many types of neural networks because a model that uses it is easier to train and often achieves better performance" [7]. The convolutional layers are built alongside layers of max pooling which are used to reduce the dimensionality of the product of the convolutional layer. Using the summary function, we can examine the model's layers, their names, their output shapes and number of parameters of each sequential layer that has been added on. With the model created and each of the convolutional layers all sequentially added, it is necessary to compile the model. This is done using the compile feature within Keras, focusing on maximising accuracy.

```
#Compiling the model
model.compile(loss = "sparse_categorical_crossentropy",
              optimizer = keras.optimizers.Adamax(learning_rate = 0.005, beta_1 = 0.9, beta_2 = 0.999),
              metrics=["accuracy"])
```

Now that all the data has been introduced and the model has been compiled, there are several ways that the model can be additionally trained to improve efficiency. One of the most common ways to do this is image augmentation, which creates new images by manipulating and augmenting the images provided within the dataset. This can be done usually through rotating, flipping or rescaling current images to create new previously unseen examples the algorithm can use to more efficiently determine patterns within the given dataset.

```
#Using ImageDataGenerator to additionally train the data
train_gen = ImageDataGenerator(rotation_range = 10, width_shift_range = 0.01, height_shift_range = 0.04)
batches = train_gen.flow(X_train, Y_train, batch_size = 32)
val_batches = train_gen.flow(X_valid, Y_valid, batch_size = 32)

#Training and evaluating the model
history = model.fit(batches, steps_per_epoch=X_train.shape[0]//32, epochs = 30, validation_data = val_batches, validation_steps = X_valid.shape[0]//32)
```

```
#Creating a graph to indicate key values across epochs
pd.DataFrame(history.history).plot(figsize = (8, 5))
plt.grid(True)
plt.gca().set_ylim(0, 1)
plt.show()

#Getting Final Evaluation Scores for the model
model.evaluate(X_test, Y_test)

#Predicting values using model
Y_pred = model.predict_classes(X_test)
Y_pred
```

II.IV Model Optimisation

Following model training, optimisation as a tool can further improve a model's accuracy through trial and error and individual parameter tweaking. Hyper-parameters are individual settings within a function/model that can be adjusted to improve a model's performance. These tweaks can include changing the learning rate of the model, which if set too high or too low can negatively affect the performance of the learning tool. Other parameters that can be tweaked are the batch sizes used to determine "the number of samples to work through before updating the internal model parameters" [8], as well as epoch sizes which defines the number times that the learning algorithm will work through the entire training dataset" [8]. Typically, the higher the number of epochs, the more accurate the model as the number of iterations grows and the machine becomes much more capable of determining patterns within the data. In the end, batch normalization was used to increase the speed of learning by the CNN, while the Dropout function and its hyperparameter, dropout rate, was tuned to find an optimal value to avoid overfitting through regularization.

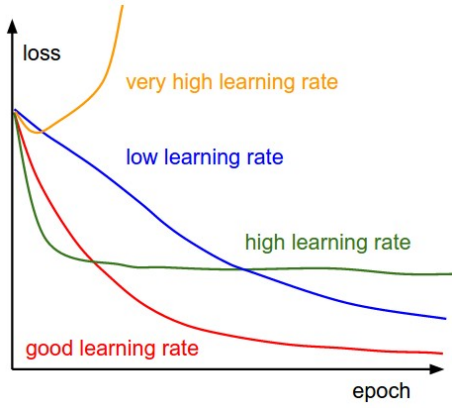

Figure 4: Learning Rate vs. Loss Factor Dependencies [9]

Having initially set the epoch value too low, we often found that the accuracy

value of the model remained very low (seen in the graph below). It is only by the time the epoch value was set to 30 that we were able to find that there is minimal change in the accuracy values of the model, and that the additional technical capacity needed to run through the dataset more often was not necessary, with a case of diminishing returns.

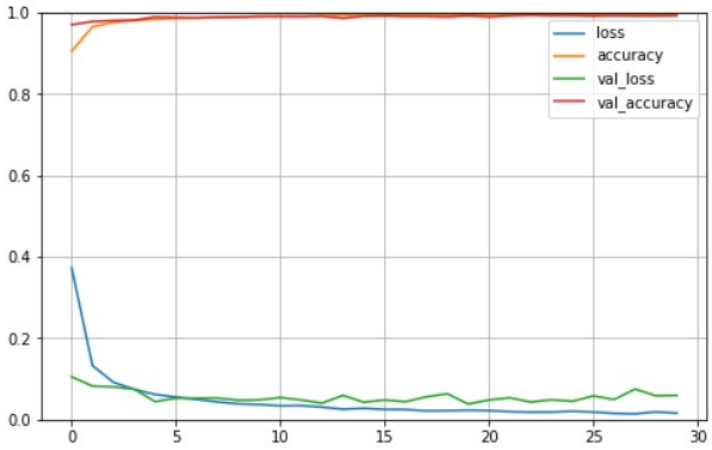

Figure 5: Visual Breakdown of Loss/Accuracy Values across Epochs

II.V Model Evaluation

Final model evaluation produced a loss value of 0.0287 and an accuracy score of 0.9952, a very positive score considering the dataset. Aside from the simple metrics of loss and accuracy, other values such as precision, recall and F1-score were considered. As evident in the below figure, across the different classes and overall, all three metrics performed exceptionally well. The extremely high precision values ranging from 0.99 to 1.00 indicated a low likelihood of the Type-I error whereby a true positive would be mislabelled. Similarly, the high recall and F1-scores provide an indication as to the, in this case, highly unlikely chance that a true negative would be mislabelled as a positive. While there is often a balancing issue between the recall and precision values, especially in imbalanced datasets, the high F1-score given with this model shows that there is a good balance between the two metrics, implying good health of the model.

```
#Classification report depicting the precision, recall, and f1-
    score of t=the different classes and overall model
from sklearn.metrics import classification_report
class_rep_rf = classification_report(Y_test, Y_pred)
print(class_rep_rf)
```

	precision	recall	f1-score	support
0	0.99	0.99	0.99	600
1	0.99	0.99	0.99	600
2	1.00	1.00	1.00	600
3	0.99	1.00	0.99	600
4	1.00	1.00	1.00	600
5	1.00	1.00	1.00	600
6	0.99	0.99	0.99	600
7	0.99	0.99	0.99	600
8	1.00	1.00	1.00	600
9	1.00	0.99	0.99	600
accuracy			1.00	6000
macro avg	1.00	1.00	1.00	6000
weighted avg	1.00	1.00	1.00	6000

Figure 6: Visual Breakdown of Precision/Recall/F1-scores

II.VI Final Steps and Evaluation

Once the model compilation, training and evaluation is complete, the data is reshaped back to original format and extracted to a csv file for use in actual evaluation on Kaggle.

The final model produces an accuracy score of **0.97966** [1]

III. Twitter Sentiment Analysis Problem

Much like the Kannada MNIST problem, the Twitter sentiment analysis problem presented is a widely renowned machine learning exercise focusing on the aspects of text analysis to determine patterns in human behaviour and interaction that may indicate a person's stance on a product or service.

Twitter is often time used by people to voice their opinions on various topics. Due to the wide-scale nature of Twitter, analysis like the one being conducted in this study can provide invaluable information to a company in the public's stance on a certain issue or even product. As the user is no longer questioned individually in a specific setting where he may not feel comfortable to tell the truth, this can also provide much more accurate results regarding the public's true view.

Sentiment Analysis in general focuses on the aspects of picking out specific key words or phrases that may give a better indication of an individual's opinion on something. These can be classed as 'good' or 'bad' words, which can direct a user or in this case algorithm to determine sentiment orientation, whether this be positive, neutral or negative. The approach of using opinion words (the lexicon) to determine opinion orientations is called the lexicon-based approach to sentiment analysis, developed and analysed further by Ding et al. in 2008 and Taboada et al. in 2010. [10] There are several issues that need to be considered with sentiment analysis including the use of emojis, which are commonplace on the Internet today, to convey emotion, as well as sarcasm, which a machine learning algorithm may not be able to pick up on easily.

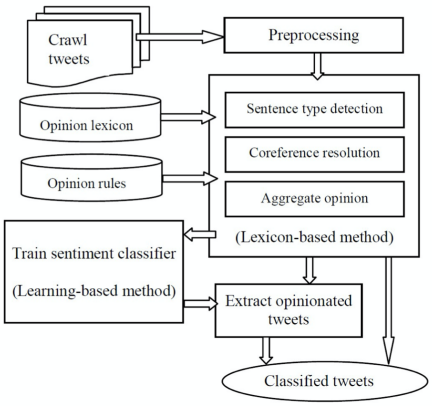

Figure 7: Visual Breakdown of Sentiment Analysis Approach [10]

III.I Approach Used

The approach used to complete the sentiment analysis was the combination of a Convolutional Neural Network and Long-Short Term Memory (LSTM) with word embeddings, encoding and decoding. The idea for this model is not new and it is "through the use of a multi-layered CNN and character level word embeddings [that] the Facebook team was able to classify the polarity of longer form text - Yelp and Amazon reviews - with 95.72 percent accuracy" [11]. It is also through the strong previous performance of Tweet2Vec, a method developed at MIT for tweet representation, that relied on "learning tweet embeddings using [a] character-level CNN-LSTM encoder decoder" [12] that formed the base of our model.

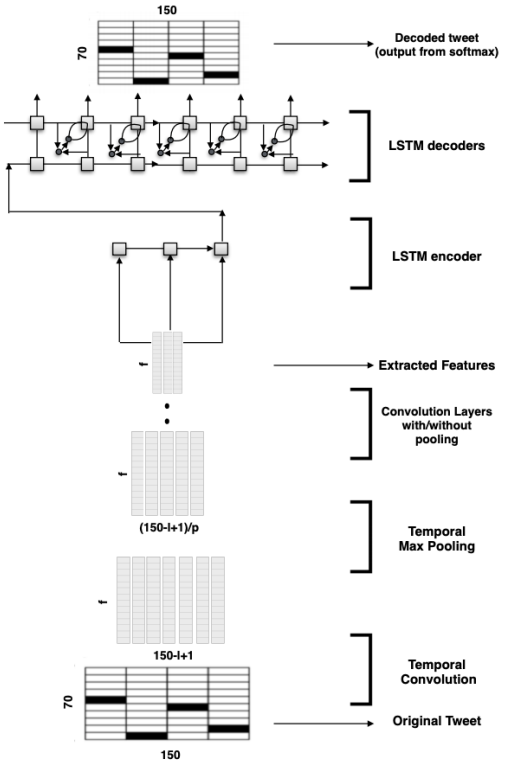

Figure 8: Visual Breakdown of CNN-LSTM Encoder-Decoder Approach [12]

IV. Code Breakdown

IV.I Packages

Similarly, to the MNIST problem, the packages used included NumPy and Pandas for basic processing, as well as TensorFlow and Keras to assist with building the machine learning algorithm. From the TensorFlow back-end, the Sequential API was imported to sequentially build the convolutional layers; Embedding, LSTM, Dense and Dropout packages were imported as layer types from which the CNN would be built.

```
import numpy as np
import pandas as pd
%tensorflow_version 1.x
import tensorflow as tf from tensorflow import keras
from keras.preprocessing import sequence
from keras import Sequential
from keras.layers import Embedding, LSTM, Dense, Dropout
from google.colab import files
```

IV.II Data Analysis

Following package import, the training and test data were uploaded to Jupyter and Google Colab, in order to once again take advantage of the additional processing performance available with the service. Simple data analysis was complete using the shape and head functions, indicating the size of the training set to be a [1000000, 5] matrix, while the test set is a [399890, 4] matrix. Both sets of data contain an id value assigned to each tweet, the date of the tweet, the user that tweeted and the text within the tweet. The training data has an additional column for the target value of the tweet determining its true sentiment. While the description of the data provides information that neutral tweets occur (where target = 2) within the dataset, further investigation however reveals the dataset only contains negative and positive tweets (where target = 0 or target = 4). Following this, all columns within the dataset aside from the target value and the text of the tweet were removed, as only these were deemed necessary for the model's function. [13] [2]

```
#Training and Test Files uploaded
file = files.upload()
train = pd.read_csv("training.csv")
test = pd.read_csv("test.csv")

train.shape
test.shape
train.head(6)
test.head(6)

count = train["target"].value_counts()
print(count)
```

IV.III Parameters

Similarly, to the Kannada MNIST algorithm, there are several parameters that can be adjusted to improve the efficiency of model prediction. The parameters that we have chosen to optimise include the vocab size, which corresponds to the total number of distinct phrases within the set of tweets. This can be used to exclude phrases that may not provide significant insight into the sentiment of the tweet; in our case, we found the lack of a limit to provide the best results. The max words parameter, as expected, removes any tweets that are above a certain number of words, with previous research suggesting that "any practical sentiment analysis algorithm generalizations should be taking tweet length into account" [14]. The previously described parameters of batch size and epochs were similarly adjusted within this model and given the training set has 1,000,000 entries, a batch size of 5000 was decided as most appropriate, producing the most significant results.

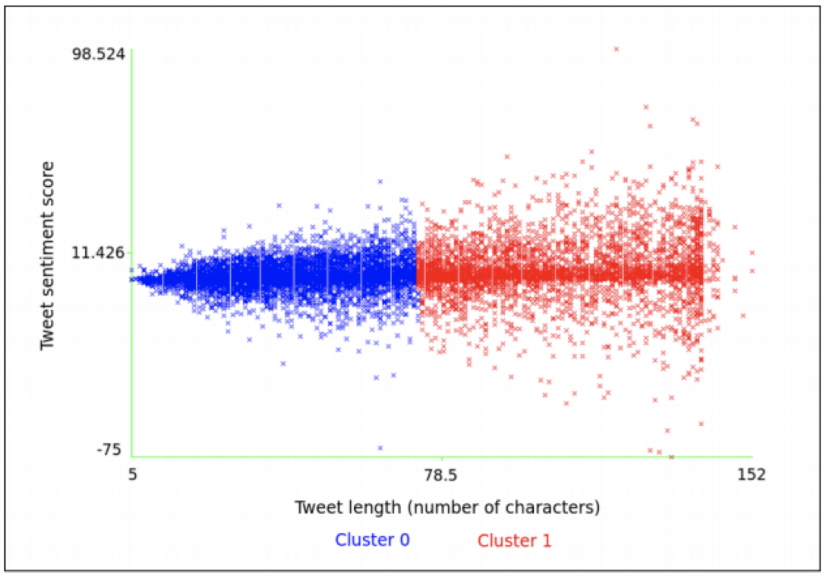

Figure 9: Visual Breakdown of Tweet Length vs. Sentiment [14]

```
#The important parameters have been collected here for fine-tuning.

vocab_size = 50000
#Set as None for no limit
max_words = 20
#Longest tweet after coding depends on the vocab size but is
    roughly 45.
batch_size = 5000
#train set has 1,000,000 entries
num_epochs = 10
embedding_size = 32
```

IV.IV Text Encoding and Padding

Each tweet must be converted to a sequence of integers. The vocabulary size has been limited to avoid overfitting with words only used in one tweet. Only the most common words are used, the rest are omitted when converting to integer list form. Constructing the tokeniser, we are considering words as our basic unit, not characters, so char level has been set to False. Applying the tokeniser to the text column creates a coded column, with all words now represented by individual integers.

```
train_raw = train[["target", "text"]]
train_raw.head(6)
test_raw = test[["text"]]
test_raw.head(6)

tokenizer = keras.preprocessing.text.Tokenizer(num_words =
    vocab_size, lower = True, char_level = False)
tokenizer.fit_on_texts(train_raw["text"])
tokenizer.fit_on_texts(test_raw["text"])
```

As all input tweets must have the same length longer tweets are shortened to the maximum number of words, while shorter tweets are padded with zeroes which do not count as words. Further analysis finds that now, the maximum tweet length and the minimum tweet length are equal to the maximum number of words specified earlier.

```
train_raw["coded"] = tokenizer.texts_to_sequences(train_raw["text"
    ])
test_raw["coded"] = tokenizer.texts_to_sequences(test_raw["text"])

print('Maximum tweet length: {}'.format(max(len(max(train_raw["
    coded"], key=len)), len(max(test_raw["coded"], key=len)))))
print('Minimum tweet length: {}'.format(min(len(min(train_raw["
    coded"], key=len)), len(min(test_raw["coded"], key=len)))))

train_raw["padded"] = train_raw["coded"]
test_raw["padded"] = test_raw["coded"]

train_raw["padded"] = sequence.pad_sequences(train_raw["coded"],
    maxlen=max_words, padding = "post", value=0).tolist()
test_raw["padded"] = sequence.pad_sequences(test_raw["coded"],
    maxlen=max_words, padding = "post", value=0).tolist()
train_raw.head(3)

print('Maximum tweet length: {}'.format(max(len(max(train_raw["
    padded"],key=len)), len(max(test_raw["padded"], key=len)))))
print('Minimum tweet length: {}'.format(min(len(min(train_raw["
    padded"],key=len)), len(min(test_raw["padded"], key=len)))))
```

IV.V Building the Model

Now that the data has been uploaded, analysed and adjusted for specific parameters, it is essential to build the model. Using the Sequential API from Keras, embedding and LSTM layers are stacked with the final layer using 'softmax' as the activation since our targets are 5 mutually exclusive classes (the numbers 0-4). A model summary reveals the layers that make up the network, as well as their output shape and number of parameters. Similar to the methods used in Roy et. al (2016), the use of a Convolution Neural Network with Long-Short Term Memory relies "on the intuition that the sequence of features (e.g. character and word n-grams) extracted from CNN can be encoded into a vector representation using LSTM that can embed the meaning of the whole tweet" [12], providing us with a complete picture as to the sentiment behind the tweet.

```
model=Sequential()
model.add(Embedding(vocab_size, embedding_size, input_length=
    max_words)) model.add(LSTM(20))
model.add(Dense(5, activation="softmax"))
print(model.summary())
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_6 (Embedding)      (None, 20, 32)            1600000
_________________________________________________________________
lstm_6 (LSTM)                (None, 100)               53200
_________________________________________________________________
dense_6 (Dense)              (None, 5)                 505
=================================================================
Total params: 1,653,705
Trainable params: 1,653,705
Non-trainable params: 0
_________________________________________________________________
None
```

Figure 10: Visual Breakdown of Keras Sequential Model Summary

The model is compiled using sparse categorical cross entropy as the loss factor, with the Adam optimizer, measuring for the highest degree of accuracy. The use of sparse categorical cross entropy is an extremely common method of calculating loss in Keras, particularly with the dataset at hand where the values are not one-hot encoded. The Adam optimizer used is similarly extremely common when training deep neural networks and "is an adaptive learning rate method [computing] individual learning rates for different parameters. Its derived from adaptive moment estimation [using] estimations of first and second moments of gradient to adapt the learning rate for each weight of the neural network" [15]. Training for increased model accuracy is also optimal as it is the easiest metric to understand and determine model value.

```
model.compile(loss="sparse_categorical_crossentropy", optimizer='
    Adam', metrics=['accuracy'])
```

In order to train the model, the training data is further split into a validation set, that will be used to determine model accuracy and a new training set on

which the model training will be performed. The sets are then prepared for training using the tolist() and asarray() commands splitting the tables into separate columns of values which can be fed directly to the fit() command. Given the batch size of 5000, the model is training on the remaining 995000 samples and validated against a batch of 5000, while the epoch value defines that the model is run 10 times to increase model accuracy.

```
validation = train_raw[:batch_size]
new_train = train_raw[batch_size:]
validation.head(3)
new_train.head(3)

X_valid = validation["padded"].values.tolist() X_valid = np.asarray
    (X_valid)
y_valid = validation["target"].values.tolist() y_valid = np.asarray
    (y_valid)
X_train = new_train["padded"].values.tolist() X_train = np.asarray(
    X_train)
y_train = new_train["target"].values.tolist() y_train = np.asarray(
    y_train)

model.fit(X_train, y_train, validation_data=(X_valid, y_valid),
    batch_size=batch_size, epochs=num_epochs)
```

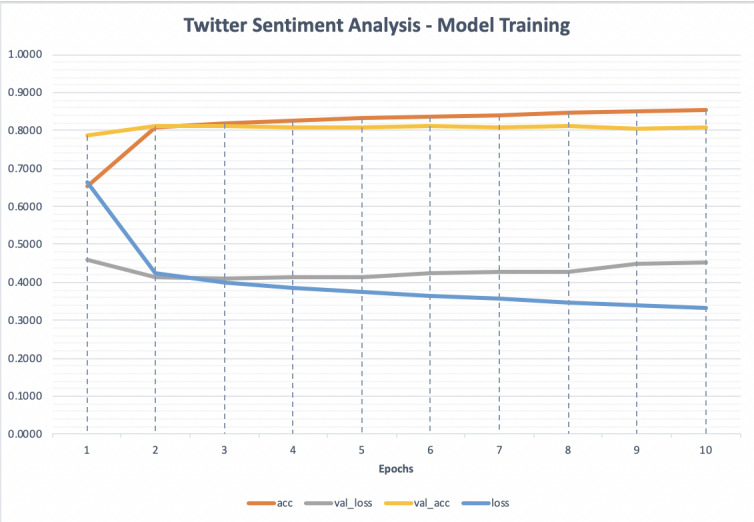

Figure 11: Visual Breakdown of Loss/Accuracy Values across Epochs

IV.VI Evaluation and Predictions

Given the model has now been built and trained, the final result of the training after 10 epochs is a loss value of 0.3341 and an accuracy score of 0.8525. The model is used on the original test data to predict the classes of the 399890 tweets and exported to a csv file for submission. We found after 10 epochs the value of the accuracy to begin to converge, thus 10 epochs were used in an attempt to avoid the issue of over-fitting the model. The accuracy score of 0.8525 provides indication that based on the validation training data around 85.25 percent of the values were assigned to the correct class.

```
to_test = test_raw["padded"].values.tolist()
to_test = np.asarray(to_test)

predictions = model.predict_classes(to_test)
test["target"] = list(predictions)

submission = test[["id","target"]]
submission.to_csv("prediction.csv",index=False) files.download("
    prediction.csv")
```

The final model produces an accuracy score of **0.81140 - Tom Reynard** [13]

References

[1] Vinay Uday Prabhu. CS98X Kannada MNIST. https://www.kaggle.com/c/cs98x-kannada-mnist/overview, 2020. [Online; accessed 3-March-2020].

[2] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[3] Chubu. Kannada Mnist Classification: A Kaggle Case Study. https://medium.com/datadriveninvestor/kannada-mnist-classification-a-kaggle-case-study-fa1b3f72b54, 2020. [Online; accessed 16-March-2020].

[4] Chris Deotte. How to score 97, 98, 99, and 100 percent. https://www.kaggle.com/c/digit-recognizer/discussion/61480, 2018. [Online; accessed 7-March-2020].

[5] Sumit Saha. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53, 2018. [Online; accessed 18-March-2020].

[6] Jason Brownlee. How Do Convolutional Layers Work in Deep Learning Neural Networks? https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/, 2019. [Online; accessed 9-March-2020].

[7] Jason Brownlee. A Gentle Introduction to the Rectified Linear Unit (ReLU). https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/, 2019. [Online; accessed 11-March-2020].

[8] Jason Brownlee. Difference Between a Batch and an Epoch in a Neural Network. https://machinelearningmastery.com/difference-between-a-batch-and-an-epoch/, 2018. [Online; accessed 10-March-2020].

[9] Hafidz Zulkifli. Understanding Learning Rates and How It Improves Performance in Deep Learning. https://towardsdatascience.com/understanding-learning-rates-and-how-it-improves-performance-in-deep-learning-d0d4059c1c10, 2018. [Online; accessed 16-March-2020].

[10] Lei Zhang, Riddhiman Ghosh, Mohamed Dekhil, Meichun Hsu, Bing Liu. Combining Lexicon-based and Learning-based Methods for Twitter Sentiment Analysis. https://www.hpl.hp.com/techreports/2011/HPL-2011-89.pdf, 2011. [Online; accessed 20-March-2020].

[11] Michael Cai. Sentiment Analysis of Tweets using Deep Neural Architectures.

https://web.stanford.edu/class/archive/cs/cs224n/cs224n.1194/reports/custom/15786247.pdf, 2018. [Online; accessed 18-March-2020].

[12] Soroush Vosoughi, Prashanth Vijayaraghavan, Deb Roy. Tweet2Vec: Learning Tweet Embeddings Using Character-level CNN-LSTM Encoder-Decoder. https://lsm.media.mit.edu/papers/tweet2vec$_v vr.pdf$, 2018. [$Online$; $accessed 18-March-2020$].

[13] R Bhayani. CS98X Twitter Sentiment Classification. https://www.kaggle.com/c/cs98x-twitter-sentiment, 2020. [Online; accessed 3-March-2020].

[14] Matthew Mayo. A Clustering Analysis of Tweet Length and its Relation to Sentiment. https://arxiv.org/pdf/1406.3287.pdf, 2015. [Online; accessed 18-March-2020].

[15] Vitaly Bushaev. Adam — latest trends in deep learning optimization. https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c, 2018. [Online; accessed 22-March-2020].