Tweet Emoji Classification

Oriya Sigawy
ID: 214984932

Chaim Averbach
ID: 207486473

Adi Peisach
ID: 326627635

January 2025

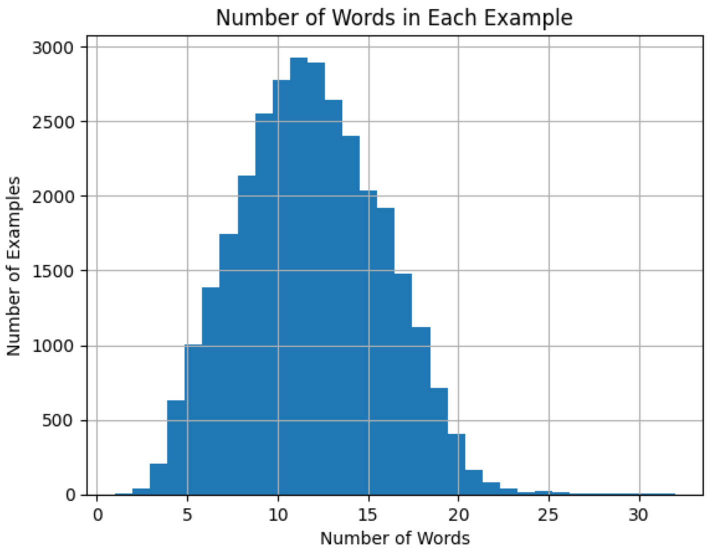
1 Introduction

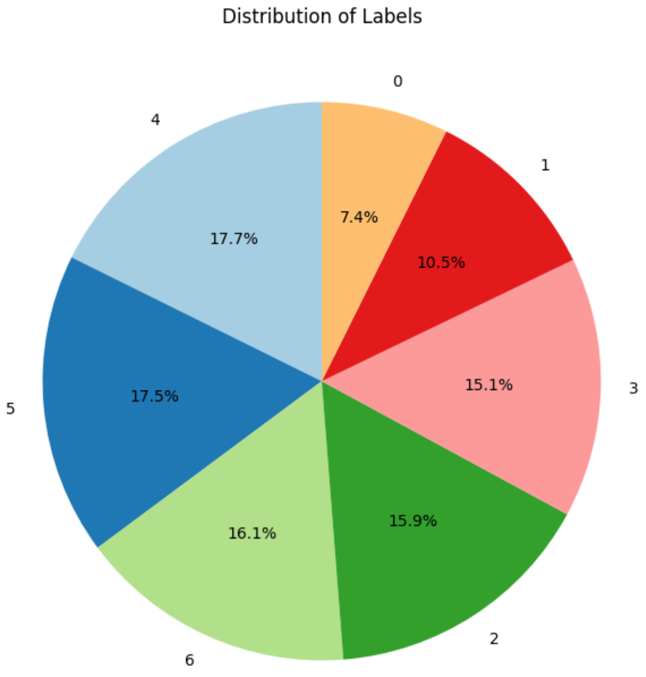
1.1 Goal

Our goal in the project is to classify collected tweets from Twitter according to a number of emojis from the dataset.

1.2 Dataset

The dataset consists of 44852 tweets with associated emojis (originally labeled as numbers, each associated with an emoji), each tweet is roughly between 5 and 20 words:





0: ✨, 1: 📸, 2: 😍, 3: 😂, 4: 😁, 5: 💕, 6: 🔥

A few example rows:

Text	Label
Heat Nation @ AmericanAirlines Arena	🔥
And my morning starts @ Church Of The Reedemer	😁
The face you make when you want to thank everyone who partied with you @user by…	📸
Baylor James, you are so loved sweet boy. @ Capital Regional Medical Center	💕
Ice coffee (@ Dunkin Donuts in Philadelphia, PA)	😍

As part of the preprocessing, we removed extra white spaces, changed all letters to lower case, tokenized the text and stemmed

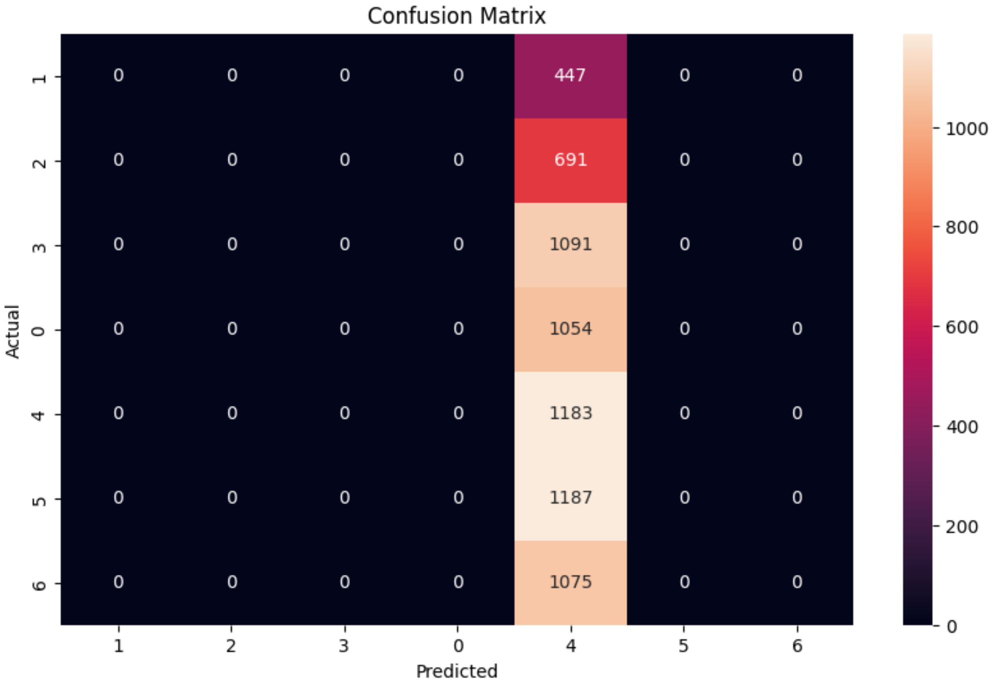
it. Additionally, for each model we added the relevant columns such as bag-of-words counts or embedded vectors.
We used a different train-test-validation split for each model, mostly for convenience reasons, and each will be discussed in the paragraph for the appropriate model.

2 Baseline Model

As a baseline model, we used a simple "most frequent" classifier, using the scikit-learn `DummyClassifier` class with the `strategy='most_frequent'` parameter.
For this model we combined the train and validation sets, as no validation is required, this gives us a final split of 0.85-0.15 for train and test respectively.
Model evaluation is as expected with low metrics and a centered confusion matrix:

Accuracy	0.1758
Precision (weighed)	0.0309
Recall (weighted)	0.1758
F1 Score (weighted)	0.0526

Now is a good time to discuss the metrics: accuracy is calculated as expected, simply as the proportion of examples that were correctly labeled, for this model this is simply equal to the proportion of examples (in the test set) that had the label 4. All other scores are calculated separately with each class being considered the 'positive' label and then the scores are averaged, weighted by the classes' proportion.



Baseline Model Confusion Matrix

The one good thing that can be said about this model is that for label 4 it achieved a recall score of 1.0...

3 Softmax Model

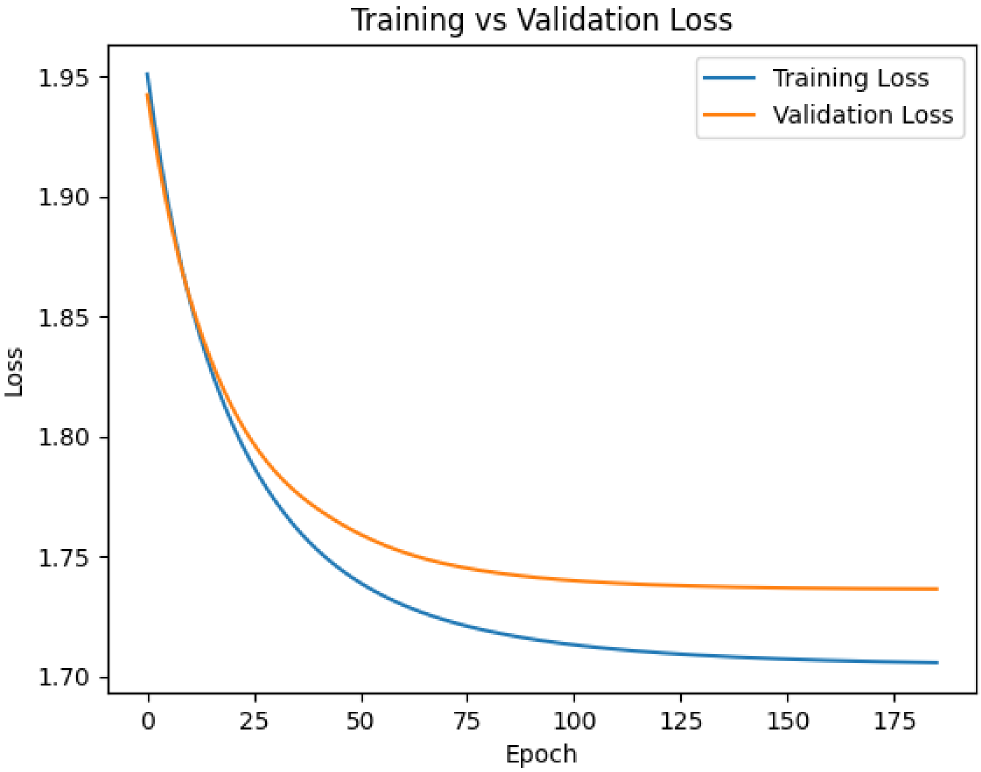
For the next model, we first applied the aforementioned preprocessing - extra white space removal, lowercase, tokenization, and stemming, then we sampled the 500 most common words (excluding stop words) from the training dataset for our bag-of-words representations of the inputs, we then added for each row in each of the datasets a column for each of the popular words representing the number of times the word appears in the text:

```
def add_bag_of_words(df, column, bag_of_word):
    new_df = df.copy()
    for word in bag_of_word:
        new_df[f'word:{word}'] = new_df[f'{column}'].apply(lambda x: int(x.count(word)))
        new_df[f'word:{word}'] = new_df[f'word:{word}'].astype('float32')
        new_df = new_df.copy()
    return new_df
```

After adding the bag-of-words features to our datasets we defined the Softmax logistic regression model as follows:

```
class SoftmaxModel(nn.Module):
    def __init__(self, input_size, output_size):
        super(SoftmaxModel, self).__init__()
        self.linear = nn.Linear(input_size, output_size)
    def forward(self, x):
        return self.linear(x) # Softmax is included in the loss function
# Model configuration
num_classes = len(emoji_mapping)
input_dim = softmax_x_train.shape[1]
softmax_model = SoftmaxModel(input_dim, num_classes)
# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(softmax_model.parameters(), lr=0.01)
```

For this model we limited the bag of words features to only the 200 most common words after observing that beyond this threshold single words have next to no effect on the prediction.
We used cross entropy as our loss function, note that it takes in the raw logits and such our model is a very simple single linear layer.
We ran the model through a simple gradient descent training loop with early stopping configured according to the loss on the validation set, for this model we used a 0.7-0.15-0.15 train-validation-test split.

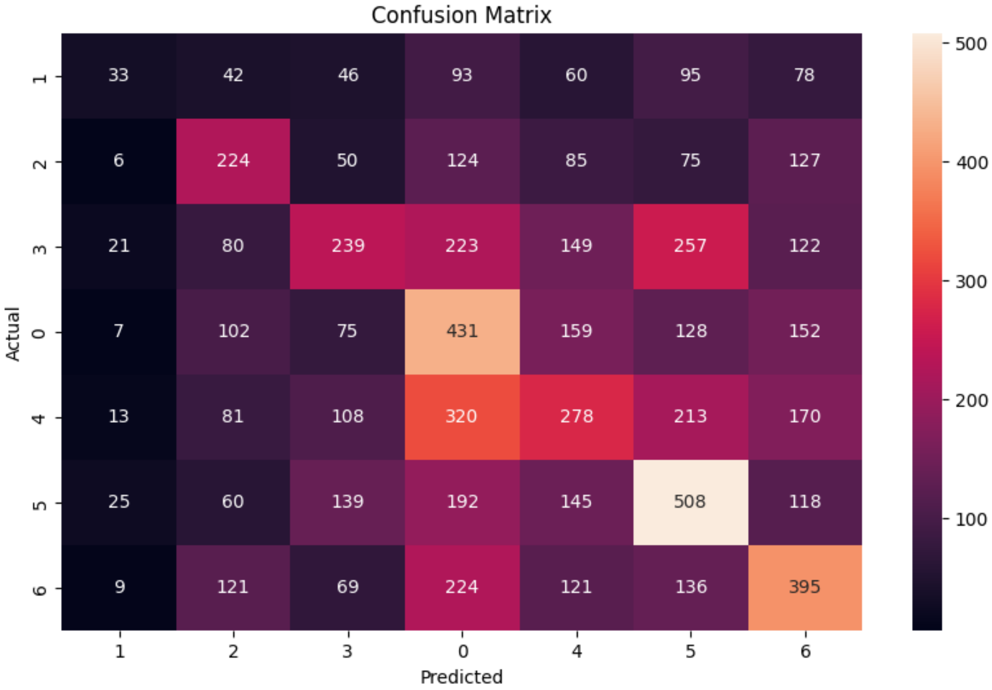


Softmax Model Losses by epoch

The early stopping triggered after approximately 180 epochs (as can be seen in the figure by the validation loss flat-lining) and yielded the following results:

Accuracy	0.3133
Precision (weighed)	0.3139
Recall (weighted)	0.3133
F1 Score (weighted)	0.3043

With the following confusion matrix:



Softmax Model Confusion Matrix

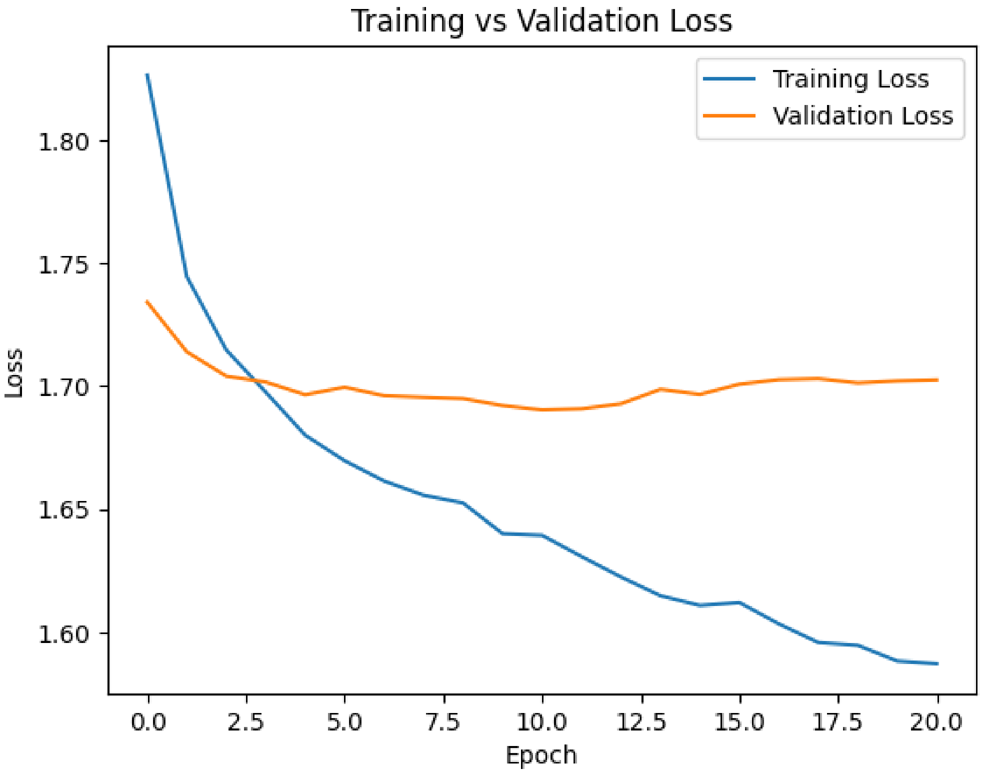
As we can see by the high values on the diagonal, the model performs decently (certainly better than the baseline), although there is still room for improvement.

4 Simple Neural Network Model

Next, we wanted to add to the model the ability to learn a little more about the relations between words, in order to do so we added a few layers to the neural network, additionally, we expanded the bag-of-words to include the 500 more common words in order to allow the model to learn more about the relations between them.
The model was constructed as follows:

```
class SimpleNN(nn.Module):
    def __init__(self, input_size, output_size):
        super(SimpleNN, self).__init__()
        self.network = nn.Sequential(
            nn.Linear(input_size, 128),
            nn.BatchNorm1d(128),
            nn.ReLU(),
            nn.Dropout(p=0.6),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(p=0.3),
            nn.Linear(64, output_size)  # No activation, raw logits for CrossEntropyLoss
        )

    def forward(self, x):
        return self.network(x)
```

As can be seen, we added a few additional layers with batch normalization and a ReLU activation, note that some pretty aggressive dropout was added to the layers after seeing the network overfitting too early in the training process.
Using a similar training loop as the softmax model with early stopping, although with far fewer epoch. As expected, the deeper network converged on its optimal solution far faster, despite using a lower learning rate (reduced to 0.001 from 0.01):

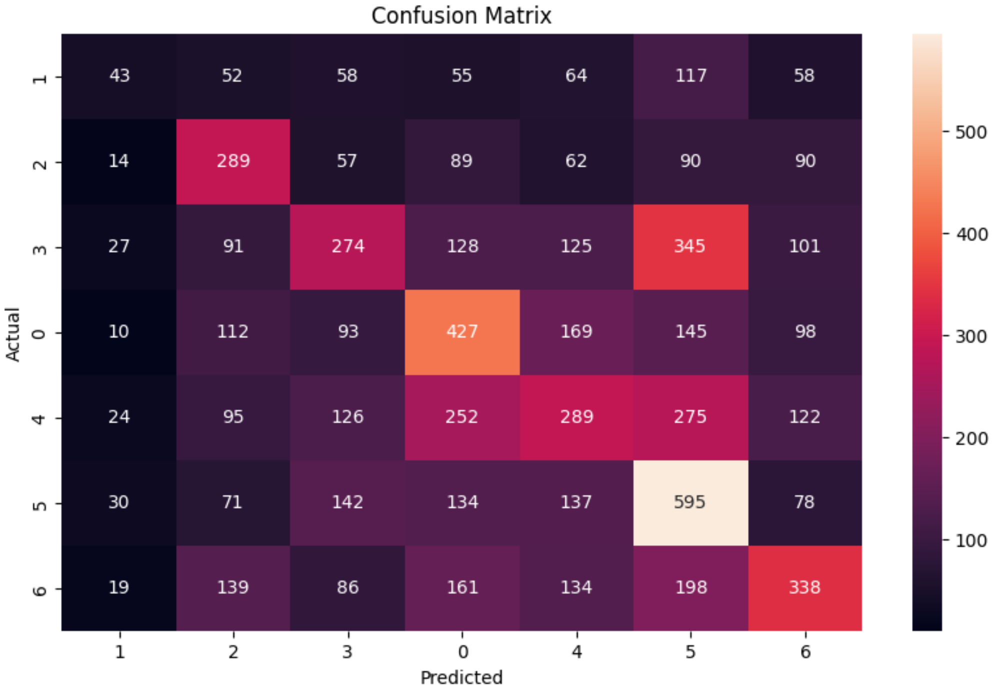


Simple Neural Network Losses by Epoch

It can be clearly seen that after around epoch 10 the network started to overfit to the training set and the validation loss started to rise.
After training the model achieved the following metrics:

Accuracy	0.3352
Precision (weighed)	0.3313
Recall (weighted)	0.3352
F1 Score (weighted)	0.3254

We can see a minor improvement over the softmax model by the newer model's ability to learn more about the relations between words, despite still only using a bag-of-words set of features. The confusion matrix is as follows:



	1	2	3	0	4	5	6
1	43	52	58	55	64	117	58
2	14	289	57	89	62	90	90
3	27	91	274	128	125	345	101
0	10	112	93	427	169	145	98
4	24	95	126	252	289	275	122
5	30	71	142	134	137	595	78
6	19	139	86	161	134	198	338

Simple Neural Network Confusion Matrix

The matrix still shows a lot of mislabeled examples, but with a strong diagonal.

5 Bidirectional LSTM Models

5.1 Simple Single Layer Model

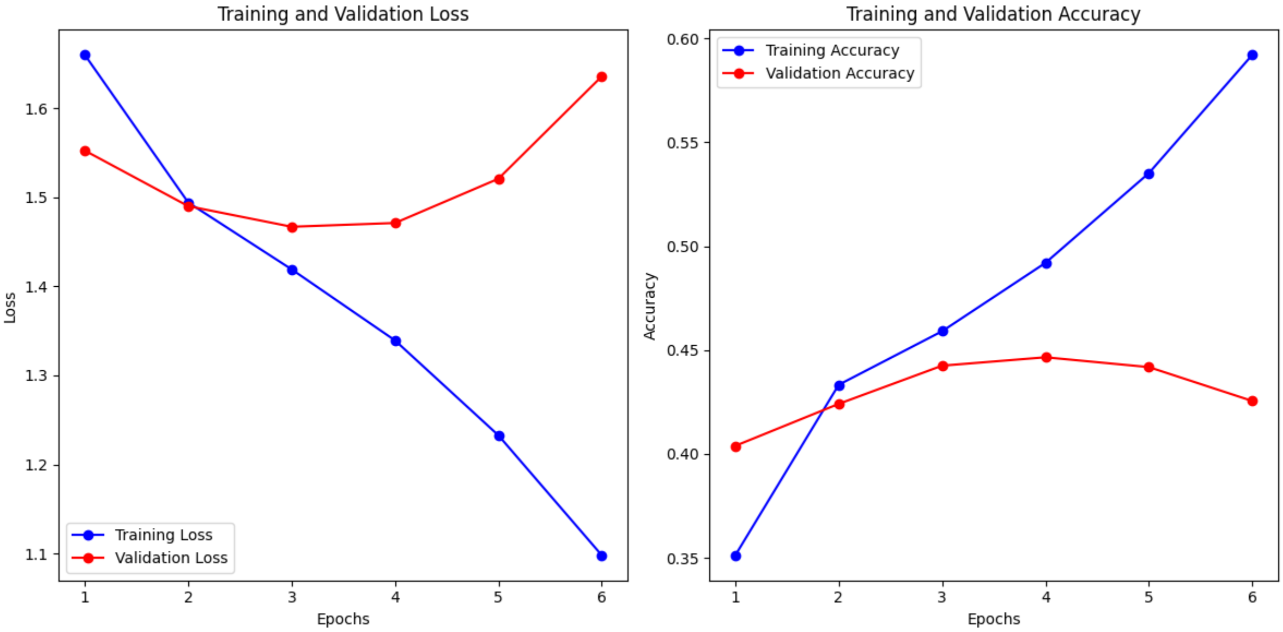
The next step in the development of the model was to use embeddings for the words in the text, if we decide to cut off each tweet at 20 words (almost all have fewer than that), if we use a decently dimensioned embedding of 200 dimensions we will reach 4000 numerical features, since this is a much too large number to use in a fully connected neural network, we decided to go with a recurrent network using LSTM cells.
First, as the features to use, we truncate each text to 20 words (after simple processing and tokenization, notably no stemming or removal of stop words) then embed the words using a pretrained GloVe embedding trained on 2 billion tweets with 27 billion tokens, this can be found in the GloVe project website, we used the version with 200 dimensions.
After embedding the tweets and adding 0-padding, each example is now a 20 by 200 tensor. In order to reduce the number of parameters of the model (as a 4000-sized input is far too large) and to better learn the relations between words with respect to locality within the text we decided to use a bidirectional LSTM layer as the first part of our model (note that we moved from a PyTorch based framework to a Keras based one, as the PyTorch LSTM model took too long to train):

```
model = Sequential()
model.add(Input(shape=(max_sequence_len, embedding_dim)))
model.add(Bidirectional(LSTM(units=512)))
model.add(Dropout(0.3))
model.add(Dense(units=128, activation='relu'))
model.add(Dense(units=64, activation='relu'))
model.add(Dense(units=32, activation='relu'))
model.add(Dense(units=len(emoji_mapping), activation='softmax'))
model.summary()

model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

hist=model.fit(x_train,y_train,validation_split=0.2,shuffle=True,batch_size=64, epochs=10,
                    callbacks=[EarlyStopping(patience=3, restore_best_weights=True)])
```

This is the summary of the network (N being the size of the batch):

Layer	Output Shape	Param #
Bidirection LSTM	(N, 1024)	2,920,448
Dropout	(N, 1024)	0
Dense #1	(N, 128)	131,200
Dense #2	(N, 64)	8,256
Dense #3	(N, 32)	2,080
Dense #4 (Logit output)	(N, 7)	231

With a total of 3,062,215 parameters to learn. Note that we no longer use batch normalization layers, as they have been observed to not add anything to this model's ability to learn, we still use a single dropout layer after the LSTM layer, this will later be changed with a better mechanism.
This model showed the ability to learn at a very quick rate, converging after only 3 epochs (later starting to overfit):

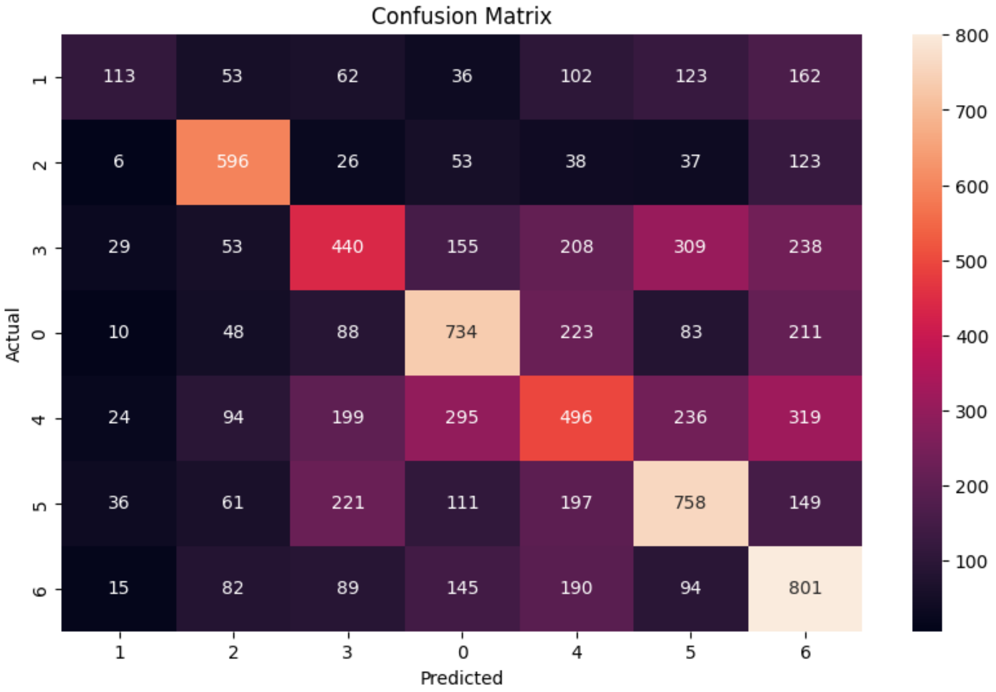


Recurrent Neural Network Losses by Epoch

We can see the validation loss being minimized at epoch 3 and then starting to rise, even though the training loss continues to go down.
This network achieves much better results than the previous ones, justifying its complexity:

Accuracy	0.4390
Precision (weighed)	0.4369
Recall (weighted)	0.4390
F1 Score (weighted)	0.4288

With the final confusion matrix showing the improvement:



Actual \ Predicted	1	2	3	0	4	5	6
1	113	53	62	36	102	123	162
2	6	596	26	53	38	37	123
3	29	53	440	155	208	309	238
0	10	48	88	734	223	83	211
4	24	94	199	295	496	236	319
5	36	61	221	111	197	758	149
6	15	82	89	145	190	94	801

Recurrent Neural Network Confusion Matrix

5.2 Adding Self-Attention

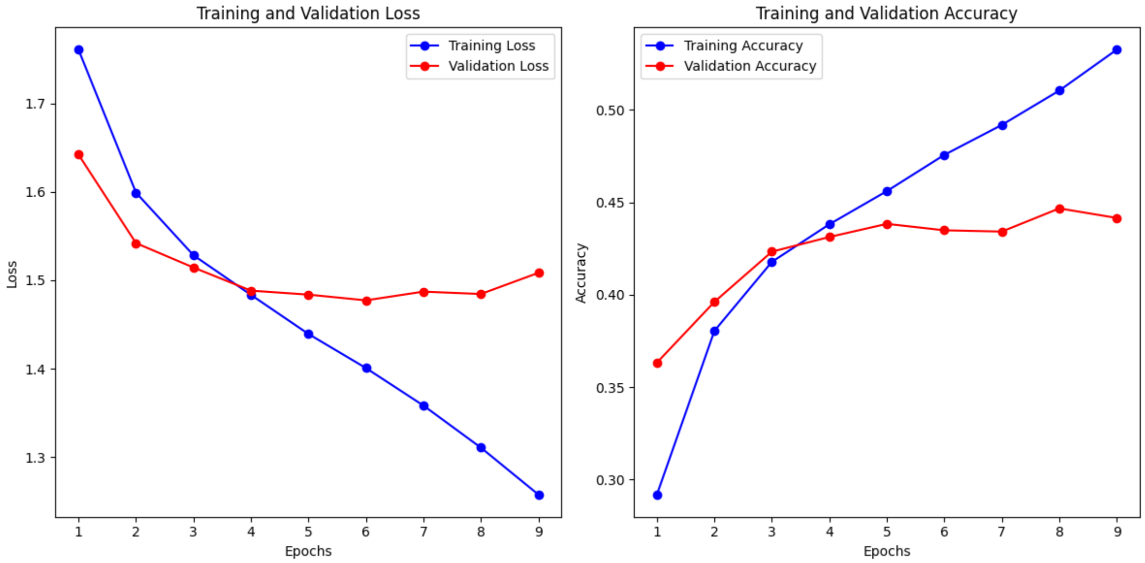
As an attempt to improve the LSTM model's ability to identify important parts of the tweet, we decided to add a self-attention layers and an additional LSTM layer:

```
model = Sequential()
model.add(Input(shape=(max_sequence_len, embedding_dim)))
model.add(Bidirectional(LSTM(units=512, recurrent_dropout=0.3, return_sequences=True)))
model.add(SeqSelfAttention())
model.add(Bidirectional(LSTM(units=256, recurrent_dropout=0.3, return_sequences=True)))
model.add(Attention())
model.add(Dense(units=128, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(units=64, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(units=32, activation='relu'))
model.add(Dense(units=len(emoji_mapping), activation='softmax'))
model.summary()
```

As can be seen, we added a sequential self attention layer after the first LSTM layer, and another self attention layer after the second one, additionally we added recurrent dropout in hopes of delaying the model's overfitting. All of which gives us the following new summary:

Layer	Output Shape	Param #
Bidirection LSTM #1	(N, 20, 1024)	2,920,448
Sequential Self Attention	(N, 20, 1024)	65,601
Bidirection LSTM #2	(N, 20, 512)	2,623,488
Self Attention	(N, 512	513
Dense #1	(N, 128)	65,664
Dropout	(N, 128)	0
Dense #2	(N, 64)	8,256
Dropout	(N, 64)	0
Dense #3	(N, 32)	2,080
Dropout	(N, 32)	0
Dense #4 (Logit output)	(N, 7)	231

For a total of 5,686,281 parameters. Using the same training parameters as before, we get the following charts:

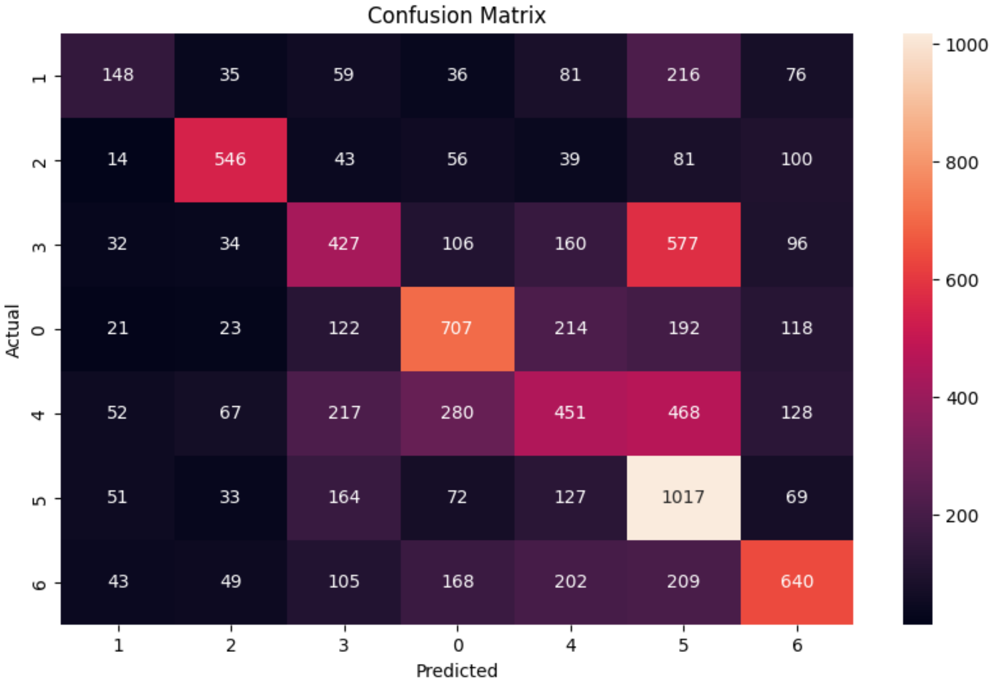


Recurrent Neural Network with Attention Losses by Epoch

As we can see, we did succeed in the goal of delaying the overfit of the model, as to results, they are debatably marignally better:

Accuracy	0.4387
Precision (weighed)	0.4459
Recall (weighted)	0.4387
F1 Score (weighted)	0.4309

With the following confusion matrix:



Recurrent Neural Network with Attention Confusion Matrix

We can see that in some regards this network performs better than the previous, while in some others it performs worse, we think that this models is open to future improvement and tweaking.

6 Conclusion

In conclusion, during our work we explored different models for classification of short texts, we experimented with different methods of data preparation, feature extraction, model design, and model training methods.
The final breakdown of model performance is as follows:

Metric	Baseline	Softmax	FCNN	LSTM RNN	LSTM RNN + Attention
Accuracy	0.1758	0.3133	0.3352	0.4390	0.4387
Precision (weighed)	0.0309	0.3139	0.3313	0.4369	0.4459
Recall (weighted)	0.1758	0.3133	0.3352	0.4390	0.4387
F1 Score (weighted)	0.0526	0.3043	0.3254	0.4288	0.4309

As a final test, we used the final model (LSTM with attention) to classify some sentences, and these are the results:

Sentence	Predicted Emoji
I had an amazing weekend with my friends	💕
My vacation is coming up soon	😁
Yoav had a great time at the party	😁
I cannot believe this had just happened	😂
Magic is in the air	✨
I had a lovely time around the city, such great scenes	📸
I am so excited for the concert tonight	💕
The birthday party was a blast	🔥