```python
import pandas as pd
import numpy as np
import torch
from PIL import Image
```

Exploratory Data Analysis

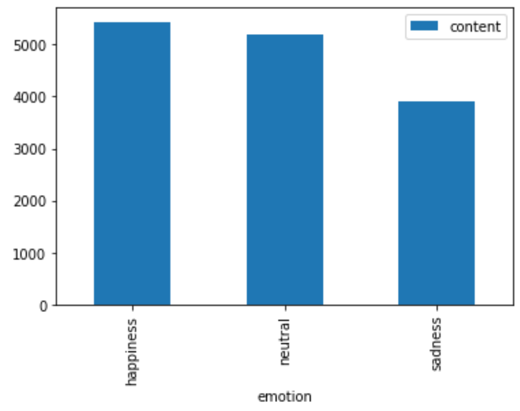
```python
train = pd.read_csv('trainEmotions.csv')
# train_tensor = torch.tensor(train.values)
print(f"Emotions are: {set(train['emotion'])}")
print("Data preview")
train.head()
```

```
Emotions are: {'happiness', 'sadness', 'neutral'}
Data preview
```

	emotion	content
0	happiness	victory for the bulldogs was celebrated by 3 w...
1	happiness	@saraLDS Thanks for that, Sara
2	happiness	@Tony_Mandarich well welcome back from the dar...
3	happiness	@sai_shediddy lol , you gotta share too
4	happiness	first up, make up for lost time with jelly. Ja...

```python
print("Labels ditribution is:")
train.groupby('emotion').count().plot.bar()
print("We can see that there is some kind of balance between the different labels.")
```

```
Labels ditribution is:
We can see that there is some kind of balance between the different labels.
```



Experiments

Preprocessing steps

In this section, we will describe the phases of preprocessing which also correspond to the experiments we ran

##

Experiment 1:

At our first trial, we created a baseline model with basic preprocessing steps:

```
 1. Remove punctuaions
2. Remove hyperlinks and mentions
3. Keep only chars in english (remove special chars i.e: '#@$%', ♡, ◐)
4. Padding all tweets to max len (~145)
5. Creating a vocabulary based on train data only - vocabulary built by encoding each word to an integer number between 1-Vocab_len
6. filter out words which appear less than 3-4 times
```

Vocabulary size: ~2500 unique words

##
Experiments 2:

In this experiment, we played around with 2-3 steps:

```
 1. Do not filter out rare words (step 6)
2. Keep specific punctuation (step 1) - we looked into keeping '!' and '?', as we thought that this is informative
```

Vocabulary size: ~15000 unique words

##
Experiments 3.1 - simple glove:

We kept our main steps as in Experiments 1-2, but created an embedding matrix from GloVE (Tweeter).
Experiments 3.2 - complex glove (We used this preprocess and embedding in our final model):

In this experiment, we looked into a whole different preprocessing steps. We took inspiration from the preprocessing that the GloVE creators made (originally written in Ruby): https://nlp.stanford.edu/projects/glove/preprocess-twitter.rb

The preprocessing is as follows:

```
 1. Ensure 2 spaces between everything
2. Replace urls with token <url>
3. Replace hashtag with token <hashtag>
4. Replace basic emojis such as ':)', ':/', ':(' with tokens : <sadface>, <smile>, 'lolface', 'heart'
5. Replace numbers with token <number>
6. Lower case
7. Seperate punctuations from word. eg. 'happy!' => 'happy !'
8. Mark elongated words (eg. "wayyyy" => "way <ELONG>")
```

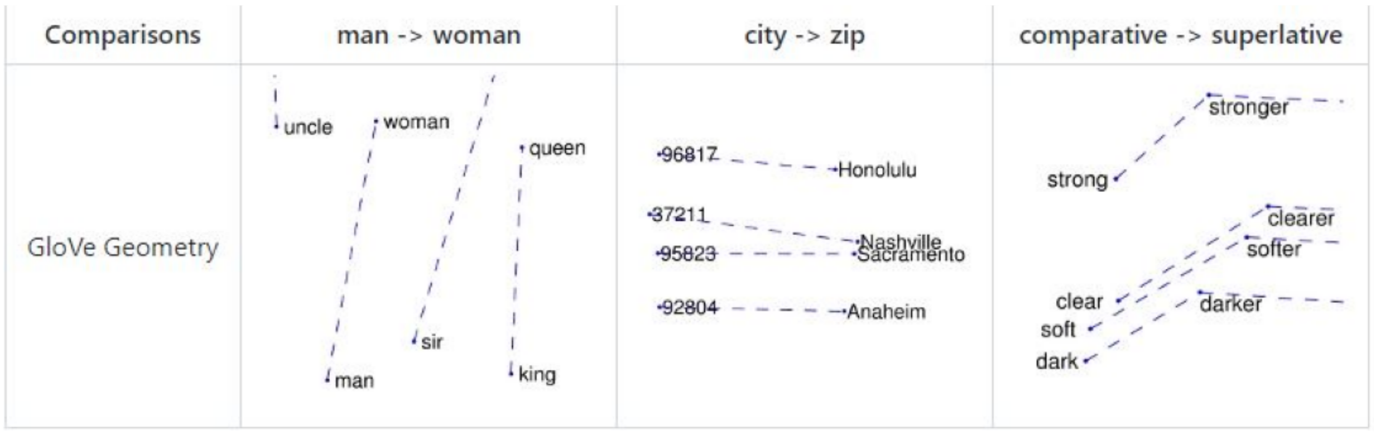
step had some problems to it : 'well' -> 'wel ', which is not the same as 'wellll'. We considered removing this step but in the end decided to keep the step as it is. 1. Mark punctuation repetitions (eg. "!!!" => "! ")

And the last 2 steps were the same as previous attempts: 1. Padd all tweets to max len (~260) 2. Create a vocabulary based on train data only - vocabulary built by encoding each word to an integer number between 1-Vocab_len
Vocabulary size: ~16000 unique words

Glove Embedding main idea:

```python
import matplotlib.pyplot as plt
img = Image.open('GLOVE.JPG')
fig, axs = plt.subplots(nrows= 1, ncols=1, figsize= (20,20))
plt.axis('off')
axs.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7fa06cbc62b0>
```



Preprocessed Data preview

```
train_glove = pd.read_csv('train_glove.csv')
train_glove = train_glove.drop(columns=["Unnamed: 0"])
train_glove.head(10)
```

	emotion	content	clean_tweet
0	happiness	victory for the bulldogs was celebrated by 3 w...	victory for the bulldogs was celebrated by <nu...
1	happiness	@saraLDS Thanks for that, Sara	<user> thanks for that , sara
2	happiness	@Tony_Mandarich well welcome back from the dar...	<user> wel <elong> welcome back from the dark ...
3	happiness	@sai_shediddy lol , you gotta share too	<user> lol , you gotta share to <elong>
4	happiness	first up, make up for lost time with jelly. Ja...	first up , make up for lost time with jelly . ...
5	happiness	@redrobinrockn next one for you!	<user> next one for you !
6	happiness	Welcome @doeko ! Really glad to know you here....	welcome <user> ! really glad to know you here ...
7	happiness	is happy and clean, squeaky clean	is happy and clean , squeaky clean
8	happiness	@Tottie Thank you, thank you!! Thought this is...	<user> thank you , thank you ! <repeat> though...
9	happiness	@aruky Yes, this NBA song is great!!! Got an ...	<user> yes , this nba song is great ! <repeat>...

Optimization

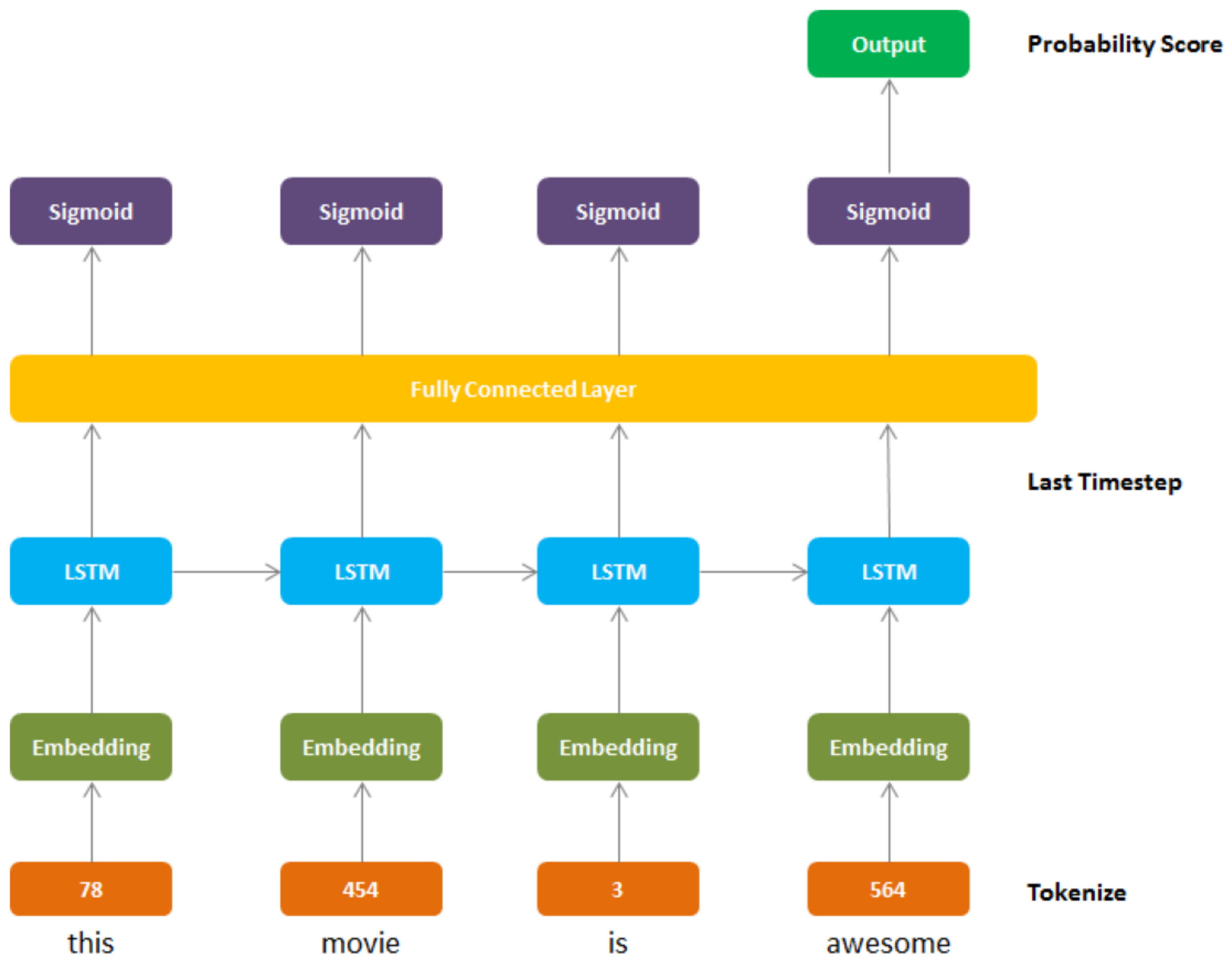
We run random optimization on the following parameters:

- learning rate: [0.1, 0.01, 0.05, 0.001, 0.005]
- epochs: [30, 60, 100]
- batch size: [32, 64, 128]
- number layers (of LSTM): [1, 2, 3, 4, 5]
- optimizer: Adam
- number hidden units: [32, 64, 128]
- dropout probability: [0.1, 0.2, 0.3, 0.4, 0.5, 0.7]
- preprocessing: according to the preprocess steps described above

Architecture

```
import matplotlib.pyplot as plt
img = Image.open('lstm architecture.png')
fig, axs = plt.subplots(nrows= 1, ncols=1, figsize= (20,20))
plt.axis('off')
axs.imshow(img)
```

```
<matplotlib.image.AxesImage at 0x7f2fa9279d90>
```



Architecture Explanation:

1. **Embedding Layer:**

   The different Embeddings we tried were:

   1. Simple Embedding:nn.Embedding, we tried different embedding_dims as hyper-parameter.
   2. Embedding with glove (the one we used): we used nn.Embedding but load to it the glove_weights. Glove_weights were created from glove embedding file, only on train vocabulary- such that: going through all the words in train_vocab: a. If the word exists in glove embedding, we use it's glove embedding b. Else we initialize a random vector
2. **LSTM layer:**

   The changes between the experiments were:

   1. **Number of LSTM layers before the Fully Connected Layer:**

      a hyper-parameter we tried to optimize
   2. **Dropout probability between LSTM layers:**

a hyper-parameter we tried to optimize

3. **Number of hidden units:**

   a hyper-parameter we tried to optimize

3. **Last layers:**

   1. **Dropout layer:**

      As not described in the picture, there is also a dropout layer before the FC layer. We tried to optimize the dropout probability - this was the same hyper-parameter as inside of LSTM layers because we tried to reduce the number of hyper-params.

   2. **Fully Connected layer**

   3. **Log Soft Max:**

      In order to get a probability for each label

Training process:

Loss Functions:

Negative Log likelihood (as our last layer is Logsoftmax)

Optimizers:

We used only Adam optimizer with and without learning rate scheduler. Scheduler parameters:

- gamma = 0.1
- step size = 7

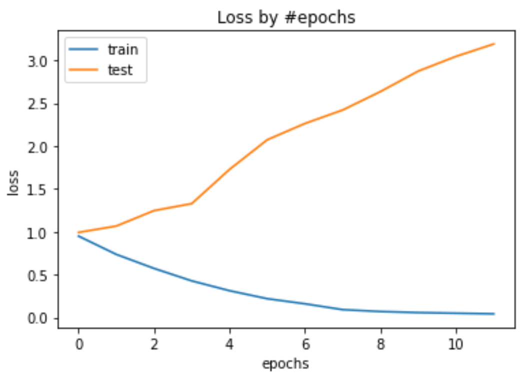
Regularization

- We used Dropout (as hyper parameter) layer, and also dropout inside LSTM cells.
- We used clip grad to avoid exploding gradients - in our experiments we tried to remove it, but the acc dropped to random acc (around 0.37).

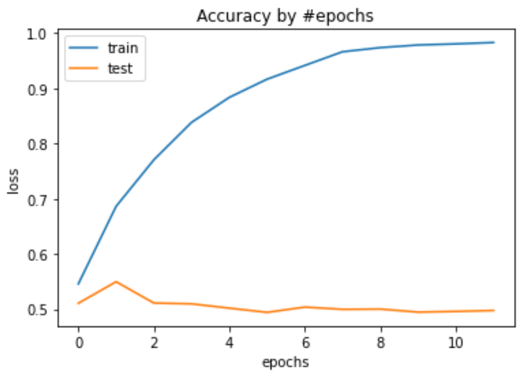
Early stopping

While running the model we implemented an early stopping approach. The idea was that if the model isn't improving within $|patience|$ epochs from his best result, then we stopped the running and saved the model that achieved the best results so far. Also, we implemented a "warm up" phase, so that the early stopping mechanism would be invoked only after 5 epochs.

```
best_model_loss = pd.read_csv('LOSS_lr=0.005_b=128_t=30_n_layers=2_n_hid=128_p=0.3_best_acc=0.5498')
best_model_loss.plot('Unnamed: 0', ['train', 'test'])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('Loss by #epochs')
plt.legend()
plt.show()
```



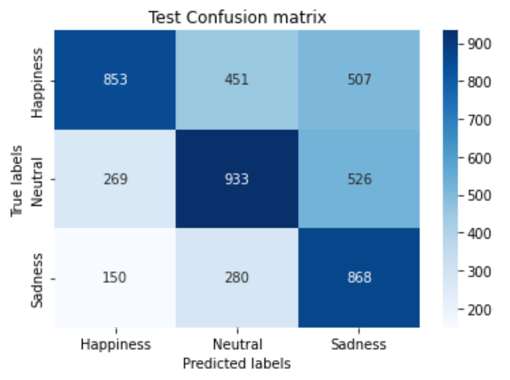
```python
best_model_acc = pd.read_csv('ACC_lr=0.005_b=128_t=30_n_layers=2_n_hid=128_p=0.3_best_acc=0.5498')
best_model_acc.plot('Unnamed: 0', ['train', 'test'])
plt.xlabel('epochs')
plt.ylabel('loss')
plt.title('Accuracy by #epochs')
plt.legend()
plt.show()
```



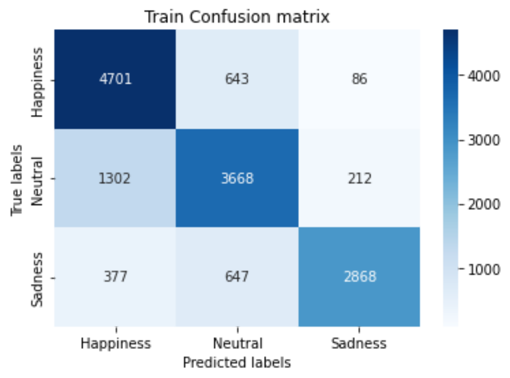
```python
## Confusion Matrix calculation
from sklearn.metrics import confusion_matrix
import seaborn as sns
import matplotlib.pyplot as plt

def plot_confusion_mat(y_lab, preds, Data):
    ##plot results
    cm = confusion_matrix(y_lab, preds)
    fig = plt.figure()
    ax = fig.add_subplot(111)
    sns.heatmap(cm, annot=True, cmap="Blues", fmt='g', ax=ax)
    # labels, title and ticks
    ax.set_xlabel('Predicted labels')
    ax.set_ylabel('True labels')
    ax.set_title(f'{Data} Confusion matrix')
    ax.xaxis.set_ticklabels(['Happiness', 'Neutral', 'Sadness'])
    ax.yaxis.set_ticklabels(['Happiness', 'Neutral', 'Sadness'])
    plt.show()
```

```python
pred_test = pd.read_csv('prediction_test.csv')['emotion'].values
real_test = pd.read_csv('TestEmotions.csv')['emotion'].values
pred_train = pd.read_csv('prediction_train.csv')['emotion'].values
real_train = pd.read_csv('TrainEmotions.csv')['emotion'].values
```

```python
plot_confusion_mat(real_test, pred_test, 'Test')
```



```
plot_confusion_mat(real_train, pred_train, 'Train')
```



In the following cells, we will show examples of the mistakes the model made on the Test Data

Real label = happiness, Predicted = sadness

```
i = 0
save_idx = []
for x, y in zip(pred_test, real_test):
    if x != y:
        if x == 'sadness' and y == 'happiness':
            save_idx.append(i)
    i+=1
save_idx = np.array(save_idx)
```

```
import random
tweets = pd.read_csv('TestEmotions.csv')['content'].values
random_idx = random.sample(range(len(save_idx)), 3)
tweets_selected = tweets[save_idx[random_idx]]
for tweet in tweets_selected:
    print(tweet)
    print()
```

```
@realchrissystar we'll be back sunday to play @ the horse, I was out last night at mex you were not  miss you

@ginjagin I think you're pretty! I'm sorry they made you feel bad

I am hungry already. Not impressed! And everybody has gone to Rubys
```

Intersting to see that the 2nd sentence has the word bad which may have confused the model

Also the 1st sentence has not indication of happy besides the last 'miss you'

Real label = happiness, Predicted = neutral

```
i = 0
save_idx = []
for x, y in zip(pred_test, real_test):
    if x != y:
        if x == 'neutral' and y == 'happiness':
            save_idx.append(i)
    i+=1
save_idx = np.array(save_idx)
```

```
import random
random_idx = random.sample(range(len(save_idx)), 3)
tweets_selected = tweets[save_idx[random_idx]]
for tweet in tweets_selected:
    print(tweet)
    print()
```

```
@jamasweetie06 see...u shouldn't be comin' 4 the city, u should be comin' 4 me

@arian_marie:my heart goes out to you

@isacullen i want more
```

Interesting to see that the 3rd sentence is not clearly Happy, but could easily be confused with Neutral (which is what we think fits better)

Summary and Conclusions:

- Most of our time was spent finding the best preprocessing routine, which also eventually improved our results.
- The training & validation process took a lot of time. Finding the best hyper-parameters for the model was a hard task (=a lot of GPU hours).

In regards to the train/test metrics:

- Loss: We can see that at the initial epochs train and test start from the same loss but immediately onwards the train loss drops as expected and the test loss only increases.
- Accuracy: Result is very similar to the above, the only difference is that we see a jump at the first epoch.
- Our input to the above results:
  - We used a GloVe embedding which was trained especially for tweets.
  - We also made sure to match our preprocessing to the glove trained embeddings.
  - We believe that the initial embedding layer was already good from the beginning so, when we enabled backpropagation it was enough to train for two epochs to receive the best result (i.e the short training was enough for fitting the LSTM layers).

Confusion matrix results:

Test:

- It is interesting to see that Happiness (true label) was confused with negative and Neutral (predicted) (=451)
- And also, Neutral (true label) was confused with Sadness (=526)
- In the above examples, we can see that the original labels do not always correspond to the sentence meaning. Actually, in our opinion, while sampling a few samples, our model gave some sentences better labels than the original one (surprisingly!). This can be attributed to human errors. We do not know who labeled the data, as the sentence can be interpreted in more than one way.

Train:

- Interesting to see that the model had a lot of mistakes (~1300) labeling Neutral tweets (true) as Netural (predicted)

Our final model:

- parameters: 1860443, most of the parameters are from the embedding layer (not surprising as this is what we also expected from using the Glove file)
- lr: 0.005
- batch size: 128
- number of epochs: 30 ## because we used early stopping, the actual number of epochs was 10
- number layers: 2
- number of hidden: 128 (in the LSTM layer)
- dropout probability: 0.3
- best accuracy for test: 0.5498
- embedding: glove (as described above)