REM SPEEDWAGON

CLASSIFYING TWITTER DISASTERS WITH A NEURAL NETWORK

February 27, 2020

Benjamin Jones

Contents

1 A Note from the Author

While the document is geared toward novices, it is assumed the reader has a basic understanding of artificial neural networks (ANNs). Tools used are Python[1], Jupyter[2], TensorFlow[3], Keras[4], ~~and ambition~~. (note to self: too corny, never write that again)

That being said, I myself am a beginner. After researching and following along with tutorials, this is my first attempt at creating my own ANN from scratch. This also serves as a learning exercise to reinforce important topics and aid in future projects.

This project aims to build a model to classify if a Twitter tweet is about a *disaster*. In this context, disaster is reserved for instances of significant damage. Think earthquakes, tornadoes, or terrorism. So no, the tweet describing your ”disastrous” breakfast in which the soufflé deflated before you could snap your Kodak moment does not count.

The data used in this project is publicly available here. Titled *Relevancy of terms to a disaster relief topic*, it contains about 10,000 tweets with surveys asking how relevant each tweet was to being about a disaster. I also want to note that not much information was provided about the particulars of the dataset, so assumptions had to be made.

I hope that you find this document informative and encouraging. It is meant to act as the first stepping stone from theory to application. If you would like to replicate and follow along, the script is available here.

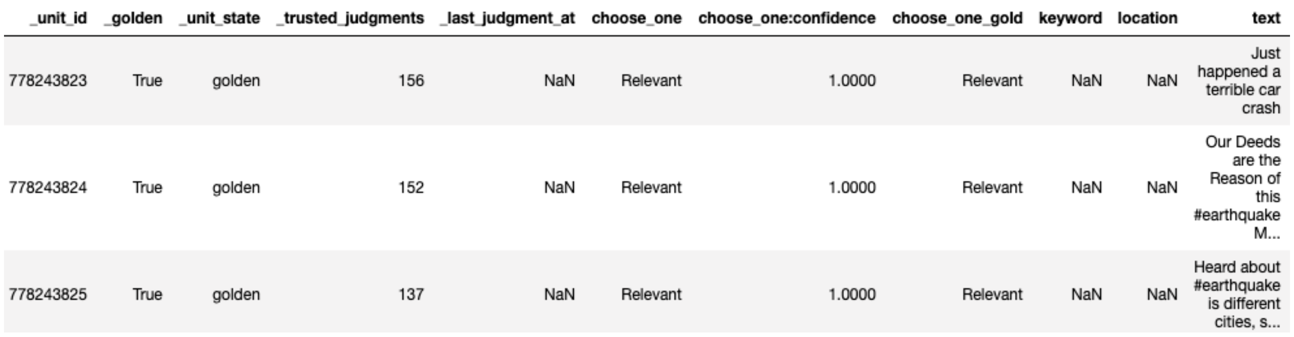
“Make sure to include a clever quote” -Abraham Lincoln, probably

2 Exploring and Cleaning the Dataset

It is good practice to become familiar with the data before any modeling. Exploring the metadata provides important information which can greatly impact decisions later in the project.

Let's start by loading the file as a pandas[1] dataframe. Note that it is vital to declare `encoding='latin-1'`. This is because our data contains informal text which often includes non ASCII characters (e.g. emojis).

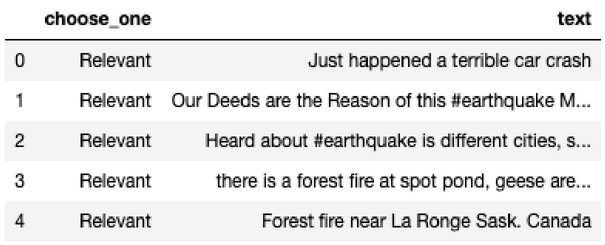
```
raw_data = pd.read_csv('/.../socialmedia-disaster-tweets-DFE.csv',
                       encoding='latin-1')
raw_data.head()
```



_unit_id	_golden	_unit_state	_trusted_judgments	_last_judgment_at	choose_one	choose_one:confidence	choose_one_gold	keyword	location	text
778243823	True	golden	156	NaN	Relevant	1.0000	Relevant	NaN	NaN	Just happened a terrible car crash
778243824	True	golden	152	NaN	Relevant	1.0000	Relevant	NaN	NaN	Our Deeds are the Reason of this #earthquake M...
778243825	True	golden	137	NaN	Relevant	1.0000	Relevant	NaN	NaN	Heard about #earthquake is different cities, s...

The only relevant columns for this project are **choose_one** and **text**. Create a separate dataframe to isolate these two columns:

```
data = raw_data[['choose_one','text']]
data.head()
```



	choose_one	text
0	Relevant	Just happened a terrible car crash
1	Relevant	Our Deeds are the Reason of this #earthquake M...
2	Relevant	Heard about #earthquake is different cities, s...
3	Relevant	there is a forest fire at spot pond, geese are...
4	Relevant	Forest fire near La Ronge Sask. Canada

The first column is the **label** or answer key; this is what is going to be predicted. The second is the raw tweet, which will be used as input data to predict the label.

[1]library for data analysis and manipulation

Next is a check for missing value and count of unique labels:

```
data.isna().sum()
```

```
choose_one   0
text         0
dtype: int64
```

```
set(data['choose_one'])
```

```
{"Can't Decide", 'Not Relevant', 'Relevant'}
```

The first output shows there are no missing values. The second output shows there are three unique values for our label. Since it only matters whether a tweet is about a disaster (”Relevant”) or not (”Not Relevant”), remove all records with the label ”Can’t Decide”:

```
data = data[data['choose_one']!="Can't Decide"]
set(data['choose_one'])
```

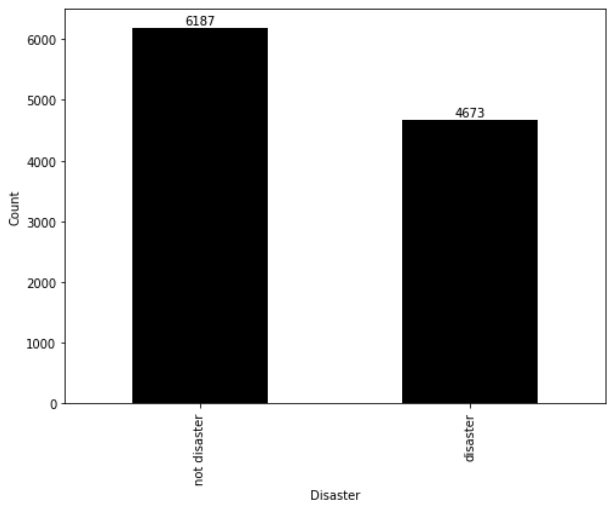
```
{'Not Relevant', 'Relevant'}
```

The final item of cleanup will be converting the values of ”Relevant” to 1 and ”Not Relevant” to 0. Let’s also relabel the ’choose_one’ column to ’label’:

```
data['label'] = 0
data.loc[data.choose_one == 'Relevant', 'label'] = 1
data.pop('choose_one')
```

The cleaned dataset has a sample size of 10,860 with 43% cases of disaster and 57% of non-disaster.

```python
plt.figure(figsize=(8, 6))
ax = pd.Series(data['label'].value_counts()).plot(kind='bar', color='black')
ax.set_xlabel('Disaster')
ax.set_ylabel('Count')
ax.set_xticklabels(['not disaster', 'disaster'])

labels = data['label'].value_counts()
rects = ax.patches

for rect, label in zip(rects, labels):
    height = rect.get_height()
    ax.text(rect.get_x() + rect.get_width() / 2, height + 5, label,
            ha='center', va='bottom')
```



Verify the dataframe to make sure it is formatted as expected:

```python
data = data.sample(frac=1) #shuffle for good measure
data.head()
```

```
                                                    text  label
2264                 I might go losing it n drive off a cliff fall ...      0
6993              Saturday Night Massacre ? 25 Years Later #TomL...      1
5138                   CA Cops: Illegal Immigrant with 4 Prior Arrest...      1
2054         @Cameron_Surname @chrisg0000 The Allies foreca...      1
9414  4 THOSE WHO CARE ABOUT SIBLING ABUSE SURVIVORS...      0
```

3 Split into Training and Validation

Now that the dataset is cleaned and properly formatted, it needs to be divided into two parts: training and validation. The majority of the data will go into the training set and will be used as learning material for the model. The validation set is kept hidden from the model and won't be used in the learning process. The point is there needs to be some outside data source to which the model has not been exposed to in order to evaluate how well the model performs on net new data (just like it would in production).

First separate and convert the labels and text into arrays and then randomly split, 75% for training and 25% for validation. Scikit-learn's[2] `train_test_split()` will do the heavy lifting:

```
x = data.iloc[:,0].values #inputs
y = data.iloc[:,1].values #labels
x_train, x_val, y_train, y_val = train_test_split(x, y, test_size = 0.25)
```

The last step is ensuring that the distribution of labels in the training set is roughly equivalent to the validation set. It is very important to make note of when there is a large imbalance. As noted earlier, there are about the same amount of disaster and non-disaster cases, no further actions is needed. Double check that the ratio of labels is consistent between the two sets:

```
unique_elements, counts_elements = np.unique(y_train, return_counts=True)
print("Training Set")
print(np.asarray((unique_elements, counts_elements)))
print(round(counts_elements[0]/(sum(counts_elements))*100,2), "% 0 (not disaster)")
print(round(100-(counts_elements[0]/(sum(counts_elements))*100),2), "% 1 (disaster)")

unique_elements, counts_elements = np.unique(y_val, return_counts=True)
print("Validation Set")
print(np.asarray((unique_elements, counts_elements)))
print(round(counts_elements[0]/(sum(counts_elements))*100,2), "% 0 (not disaster)")
print(round(100-(counts_elements[0]/(sum(counts_elements))*100),2), "% 1 (disaster)")
```

```
Training Set                     Validation Set
[[   0    1]                     [[   0    1]
 [4680 3465]]                     [1507 1208]]
57.46 % 0 (not disaster)         55.51 % 0 (not disaster)
42.54 % 1 (disaster)             44.49 % 1 (disaster)
```

[2] library for machine learning

4 Model

4.1 Saving

This part is optional but highly encouraged. This enables automatic saving which allows one to pause training and resume later. It also enables the model to be portable - allowing the model/weights to be used on a separate machine with none of the original code.

This is done using the `callbacks` feature:

```
checkpoint_path = '/.../cp-{epoch:04d}'
checkpoint_dir = os.path.dirname(checkpoint_path)
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                save_weights_only=True,
                                                verbose=1,
                                                save_freq='epoch'
                                                )
```

Brief parameter definitions are as follows:

`filepath` - define the directory of where to save[3]
`save_weights_only` - save just the weights or the entire model
`verbose` - enable/disable messages for debugging
`save_freq` - how often to save; 'epoch' will save after each epoch

[3] using '{epoch:04d}' at the end generates a unique file name for each save file

4.2 Embedding Vectors

Since our input data consists of text, it must first be converted to a numerical form. One possible method is **embedding vectors**[4]. This is an area of study all on its own, so for the sake of brevity we won't go into further detail. Basically, the wheel doesn't need to be reinvented and a pre-trained text embedding model from TensorFlow Hub[5] can be implemented.

This will serve as the first layer of the neural network. A few examples of how the text will be numerically represented is also displayed:

```
embedding = "https://tfhub.dev/google/tf2-preview/gnews-swivel-20dim/1"
hub_layer = hub.KerasLayer(embedding, input_shape=[],
                           dtype=tf.string, trainable=True)
hub_layer(data['text'][:3])
```

```
<tf.Tensor: shape=(3, 20), dtype=float32, numpy=
array([[ 0.28795514, -0.33089155,  0.4694359 ,  0.25962555, -0.3899307 ,
         0.519044  ,  0.02347763,  0.20918302,  0.40586933,  0.15561381,
        -0.5326305 ,  0.4416759 , -0.209601  , -0.14638771,  0.4305011 ,
        -0.34971237, -0.6025258 , -0.24149889, -0.37806344,  0.29764125],
       [ 0.3428518 , -1.2258928 ,  1.1127486 , -0.09706269, -1.6098815 ,
        -2.0234637 , -1.4268725 ,  0.89119524,  1.8129581 ,  0.74163836,
        -1.5837238 ,  0.5597518 ,  0.7064632 , -0.06522568, -1.801542  ,
         0.95429164,  1.0367122 , -0.57998896, -1.228939  , -0.07077654],
       [-0.7684545 , -0.3240194 , -0.4590403 , -0.78238493, -0.6346923 ,
         0.84253484,  0.3933478 ,  0.3958523 , -0.12424278,  0.36349782,
        -0.243992  , -0.09113774,  0.36228633, -0.17937404,  0.5450773 ,
         0.85830885,  0.44275758, -0.47713774, -0.7886732 ,  0.7035698 ]],
      dtype=float32)>
```

[4]a learned, vector (numerical) representation for text where words that have similar meaning are represented similarly

4.3 Framework

In this section the model architecture will be built. Start by creating a **sequential** model, which is commonly used for non-complex settings. This type of network takes a single input and returns a single output.

After declaring the model type as sequential, layers will be added.

1. The first layer will be the embedding vectors referenced earlier.
2. The second will be a **dense layer**, which contains a pre-specified number of nodes n, with every input connected to every output and each node having its own weight. The **activation function**[5] will also be defined: **rectified linear**[6] in this case.
3. The final layer will also be a dense layer with a single output node.

A summary can be displayed which shows the total number of **parameters**, which is how many weights can be trained and adjusted.

```
model = tf.keras.Sequential()
model.add(hub_layer)
model.add(tf.keras.layers.Dense(16, activation='relu'))
model.add(tf.keras.layers.Dense(1))

model.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
keras_layer (KerasLayer)     (None, 20)                400020
_________________________________________________________________
dense (Dense)                (None, 16)                336
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 17
=================================================================
Total params: 400,373
Trainable params: 400,373
Non-trainable params: 0
```

[5]functions which defines the output
[6]common activation function which only outputs non-negative values; $f(x) = max(0, x)$

4.4 Compile

Compiling a model defines the method of how it learns. This is done through `.compile()`:

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

`optimizer` - method by which the model will adjust its weights and learning rate
`loss` - function which the model will try to minimize
`metrics` - list of metrics to be evaluated by the model during training and testing

The optimizing algorithm used here is **Adam**[6]. It is a solid approach and useful for general applications. At the beginner level, Adam should be the default optimizer.

Loss measures how far away the model is from the correct answer. The goal of the model is to continuously adjust its weights to minimize the loss. **Binary Cross Entropy** is a good starting point for binary classifiers and more info can be found here. A more familiar sounding measure for loss, which won't be used here, is MSE (mean squared error).

The metric of interest in this situation, accuracy, returns a value $[0, 1]$, depending on the percentage of successful predictions.

4.5 Training

At this point the model framework has been set up and the learning parameters have been established. It is now time to train the model using `.fit()`:

```
history = model.fit(x=x_train,
                    y=y_train,
                    batch_size=32,
                    epochs=30,
                    validation_data=(x_val, y_val),
                    shuffle=True,
                    verbose=1,
                   callbacks=[cp_callback])
```

```
Train on 8145 samples, validate on 2715 samples
Epoch 1/30
8032/8145 [====================>.] - ETA: 0s - loss: 0.6049 - accuracy: 0.6846
Epoch 00001: saving model to /.../cp-0001
8145/8145 [======================] - 1s 150us/sample -
loss: 0.6046 - accuracy: 0.6853 - val_loss: 0.5247 - val_accuracy: 0.7466
Epoch 2/30
7488/8145 [====================>.] - ETA: 0s - loss: 0.4980 - accuracy: 0.7666
Epoch 00002: saving model to /.../cp-0002
8145/8145 [======================] - 1s 92us/sample -
loss: 0.4958 - accuracy: 0.7681 - val_loss: 0.4961 - val_accuracy: 0.7683
Epoch 3/30
8064/8145 [====================>.] - ETA: 0s - loss: 0.4529 - accuracy: 0.7953
8145/8145 [======================] - 1s 94us/sample -
loss: 0.4542 - accuracy: 0.7946 - val_loss: 0.4759 - val_accuracy: 0.7768
```

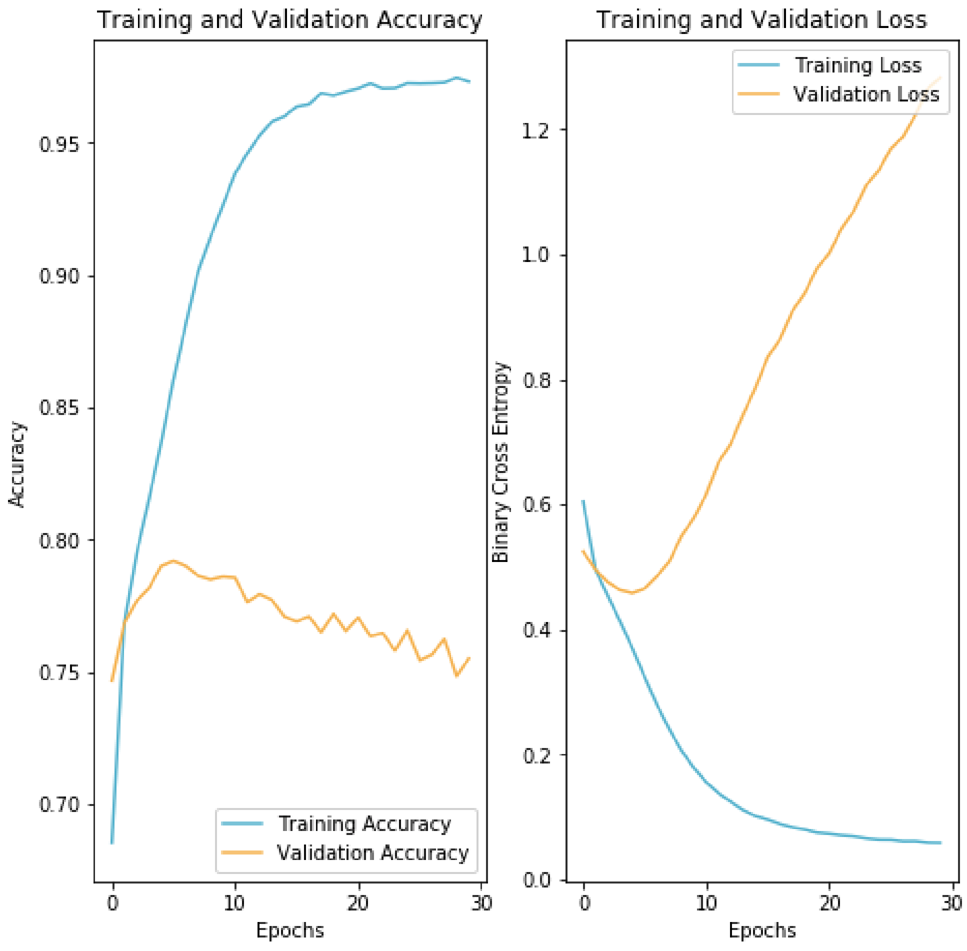
`x` - inputs (text)
`y` - labels
`batch_size` - number of samples per gradient update, default is 32
`epoch` - number of epochs to train the model; an epoch is a single iteration over the entire dataset
`validation_data` - optional, evaluate the model on outside data
`shuffle` - optional, whether or not to reorder the training data
`verbose` - enable/disable messages for debugging
`callbacks` - list of callbacks to apply

The output shows the progress of the model throughout its training. For each epoch, it returns training loss, training accuracy, save location, validation loss, and validation accuracy. Notice that the loss decreases during the first three epochs - this is the model learning from the training data and adjusting its weights.

4.6 Evaluation 1

The `history` object created stores the loss and accuracy metrics for each epoch. This is useful to create a visual which makes model evaluation easier:

```
acc = history.history['accuracy']
val_acc = history.history['val_accuracy']

loss = history.history['loss']
val_loss = history.history['val_loss']

epochs_range = range(30)

#plot aesthetics
color_train = '#5DB9D2'
color_val = '#F6B956'

plt.figure(figsize=(8, 8))
plt.subplot(1, 2, 1)
plt.plot(epochs_range, acc, label='Training Accuracy', color=color_train)
plt.plot(epochs_range, val_acc, label='Validation Accuracy', color=color_val)
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epochs')

plt.subplot(1, 2, 2)
plt.plot(epochs_range, loss, label='Training Loss', color=color_train)
plt.plot(epochs_range, val_loss, label='Validation Loss', color=color_val)
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')
plt.ylabel('Binary Cross Entropy')
plt.xlabel('Epochs')
plt.show()
```



Clearly there is **overfitting** - the model has been trained too specifically for the training data and performs poorly on unseen data. This can be seen due to the fact that after the first few epochs, the validation loss (and accuracy) worsens and greatly diverges from the training loss.

In the next section adjustments will be made in attempt to improve.

4.7 Adjustments

A few adjustments to the model have been made in attempt to combat overfitting. Briefly, these include:

Regularizer[7] - penalizes weights by increasing loss, causing weights to be adjusted only if they improve the model enough to overcome the penalty

Dropout - nodes are ignored with probability p in effort to prevent specialization

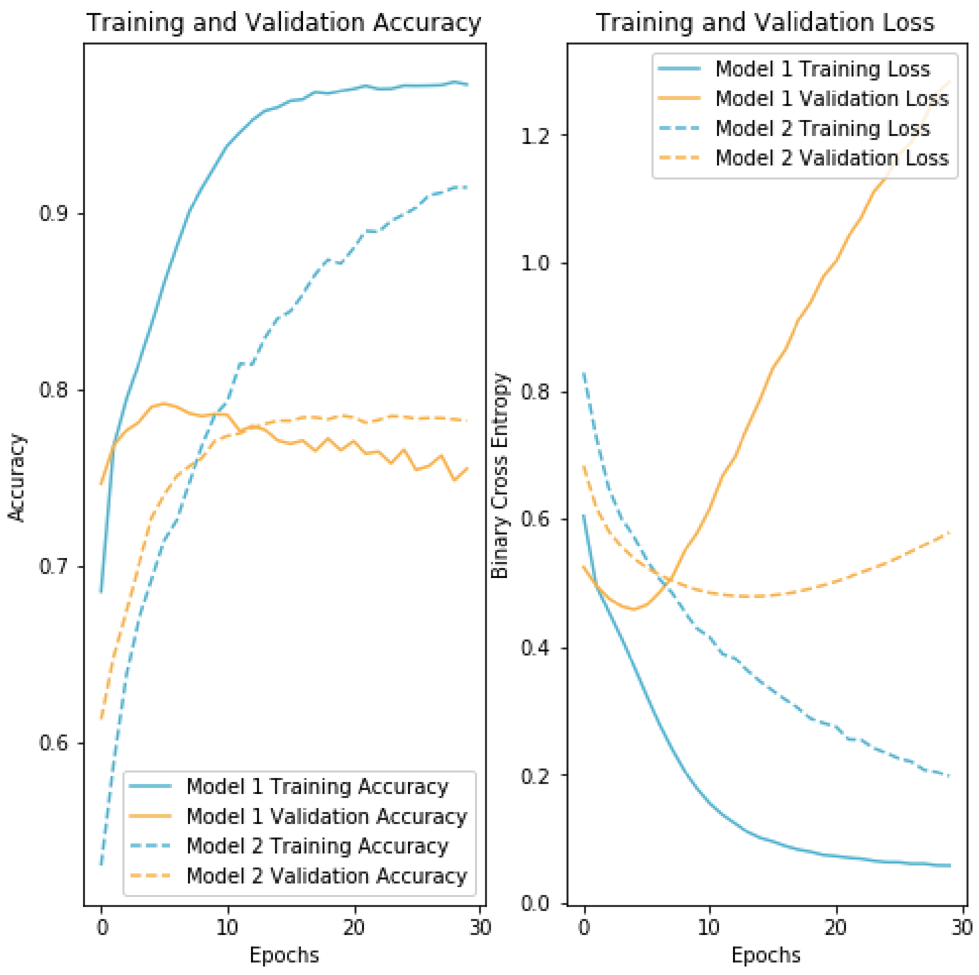
Batch Size - increasing the batch size from 32 to 650 reduces the frequency at which the weights are changed

```
from tensorflow.keras import regularizers
checkpoint_path2 = '/.../cp-{epoch:04d}'
checkpoint_dir2 = os.path.dirname(checkpoint_path2)
cp_callback2 = tf.keras.callbacks.ModelCheckpoint(checkpoint_path2,
                                                save_weights_only=True,
                                                verbose=1,
                                                save_freq='epoch'
                                                )

model2 = tf.keras.Sequential()
model2.add(hub_layer)
model2.add(tf.keras.layers.Dense(16, activation='relu',
                                kernel_regularizer=regularizers.l2(0.0001)))
model2.add(tf.keras.layers.Dropout(0.5))
model2.add(tf.keras.layers.Dense(1))
model2.summary()

model2.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])

history2 = model2.fit(x=x_train,
                    y=y_train,
                    batch_size=650,
                    epochs=30,
                    validation_data=(x_val, y_val),
                    shuffle=True,
                    verbose=1,
                   callbacks=[cp_callback2])
```

[7]L2 regularization penalizes weights in proportion to the sum of the squares of the weights

4.8 Evaluation 2

Now model 2 will be trained, evaluated, and compared to the previous model:



One immediate difference is model 2's loss; it continuously decreases for about twice as many epochs as model 1. Additionally model 2's accuracy flattens out over time while model 1's declined. Generally speaking, the shape of the curves between training and validation are much more similar for adjusted model.

Admittedly, there is still a large gap between the training and validation curves, indicating overfitting is still an issue. But trial and error is a major part of neural networks, and the adjustments made suggest a step in the right direction.

The best version of model 2 was after training for 14 epochs, as this iteration had the lowest loss. Loading the model at this point and using `.evaluate()`, we see it has an accuracy of 79% with a loss of 0.46:

```
model2_best = tf.keras.Sequential()
model2_best.add(hub_layer)
model2_best.add(tf.keras.layers.Dense(16, activation='relu'))
model2_best.add(tf.keras.layers.Dense(1))

model2_best.compile(optimizer='adam',
              loss=tf.keras.losses.BinaryCrossentropy(from_logits=True),
              metrics=['accuracy'])

model2_best.load_weights('/.../cp-0014')
model2_best.evaluate(x_val, y_val, verbose=1)
```

```
2715/2715 [======================] - 0s 61us/sample - loss: 0.4583 - accuracy: 0.7901
[0.45830289679776676, 0.7900553]
```

4.9 Predicting

Once the finishing touches are complete, the model can serve its ultimate purpose: making prediction. Using `.predict()` on several tweets, the model predicted the first three were not about a legitimate disaster[8] while the last two were:

```
outside_data = ["Judge Smails: Spaulding, get dressed you're playing golf."
                "Spalding Smails: No I'm not grandpa I'm playing tennis."
                "Judge Smails: You're playing golf and you're going to like it."
                "Spalding Smails: What about my asthma?"
                "Judge Smails: I'll give you asthma.",

                "that halftime show was a disaster",

                "now THIS is podracing!",

                "disastrous wildfires in australia continue",

                "the 2010 earthquake in haiti resulted in 230,000 dead"]

model2_best.predict_classes(outside_data)
```

```
array([[0],
       [0],
       [0],
       [1],
       [1]], dtype=int32)
```

[8]Judge Smails may disagree

5 References

[1] Python Core Team (2020). Python: A dynamic, open source programming language.
`https://www.python.org/`

[2] Project Jupyter (2020). Project Jupyter: A dynamic, open source data science and scientific platform.
`https://jupyter.org/`

[3] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Rafal Jozefowicz, Yangqing Jia, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Mike Schuster, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: *Large-scale machine learning on heterogeneous systems*, 2015. Software available from tensorflow.org.

[4] Chollet, François and others, 2015.
`https://keras.io/`

[5] Noam Shazeer, Ryan Doherty, Colin Evans, Chris Waterson. Swivel: Improving Embeddings by Noticing What's Missing.

[6] Diederik P. Kingma, Jimmy Ba. *Adam: A Method for Stochastic Optimization*, 2014.
https://arxiv.org/abs/1412.6980