The Ultimate Guide on Deep Learning for Web Developers

(Brought to you for free from the team at https://wwww.zerotodeeplearning.com)

Deep Learning is revolutionizing the world! It's literally eating the world. There are endless supplies of examples as to what deep learning actually means and what it does, but we're mostly web developers and write all of our applications in JavaScript… deep learning is *only for the lab and academics, right?*

As it turns out, **web developers can use deep learning** too! We've spent the year and some months figuring out how to use deep learning and compiled all of our learning into a book (Zero to Deep Learning). However, this isn't an advertisement about the book, but instead an actual step-by-step process of integrating deep learning with an **actual web app**.

The power of deep learning is incredible. If you haven't heard of it, it's making some previously thought-of impossible actions possible.

Amazing examples of deep learning

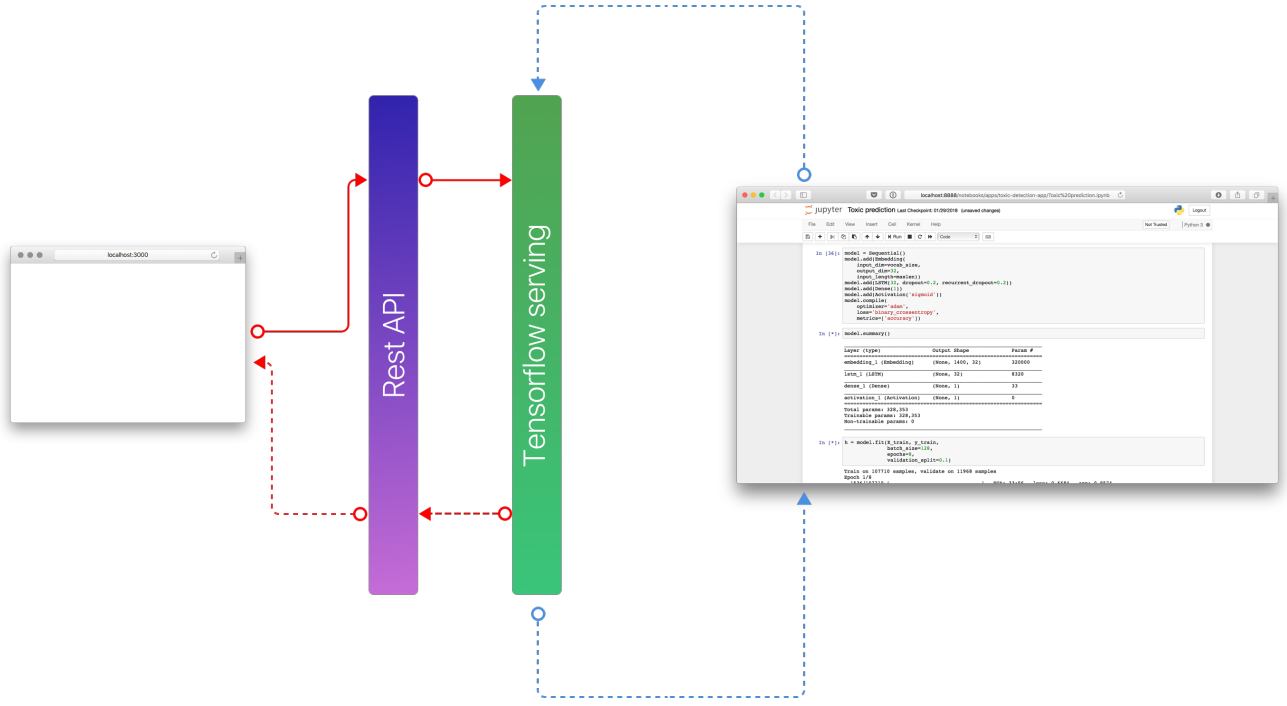
Instead of needing to know the exact names of products we're searching for, we can use deep learning to search via sketches or we can turn those sketches into photorealistic images.

Deep Learning powers real-time speech translation on Skype, automated email replies, and even self-driving cars. Deep learning is really revolutionizing the world.

How do we use deep learning as web developers?

Although there are several different ways that we can integrate deep learning software into our applications, we'll be writing our project in the way that allows us to scale the different parts of our application.

For the app we're working through, we'll have the front-end, a simple restful api server, and tensorflow serving serving up our model in production. We'll develop our neural network using keras and tensorflow, which we'll serve up through tensorflow serving.



Don’t worry if that didn’t make much sense to you yet. When I first started this project, I only knew about half of those words.

Over the course of the next few posts, we’ll be building a web application that integrates deep learning (a Recurrent Neural Network, to be specific) to bring intelligence into our app.

What we’ll be building

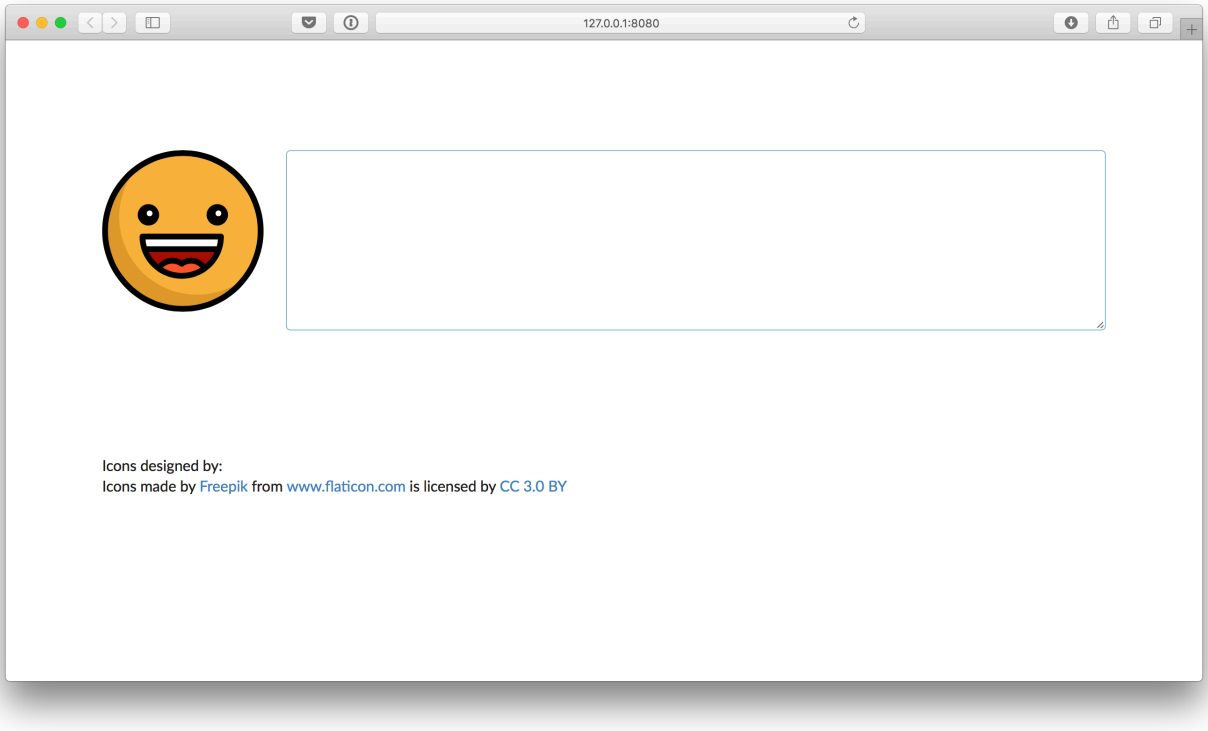
The application we’ll build is a simple toxic language detection application which will visually show if our language is malignant or not. In a real application (outside of this demo), we might take action either on the front-end or the back-end depending upon the response, like preventing the user from posting or adding a visual tag that indicates it’s toxicity level.

We have 4 major parts to discuss throughout this post:

Front-end

We’ll have a web application which we’ll build in plain ole’ JavaScript (no framework flame-wars here #sorrynotsorry).

> If you have a favorite framework, the JavaScript we’ll be writing is straight-forward and should easily translate into your framework preference.

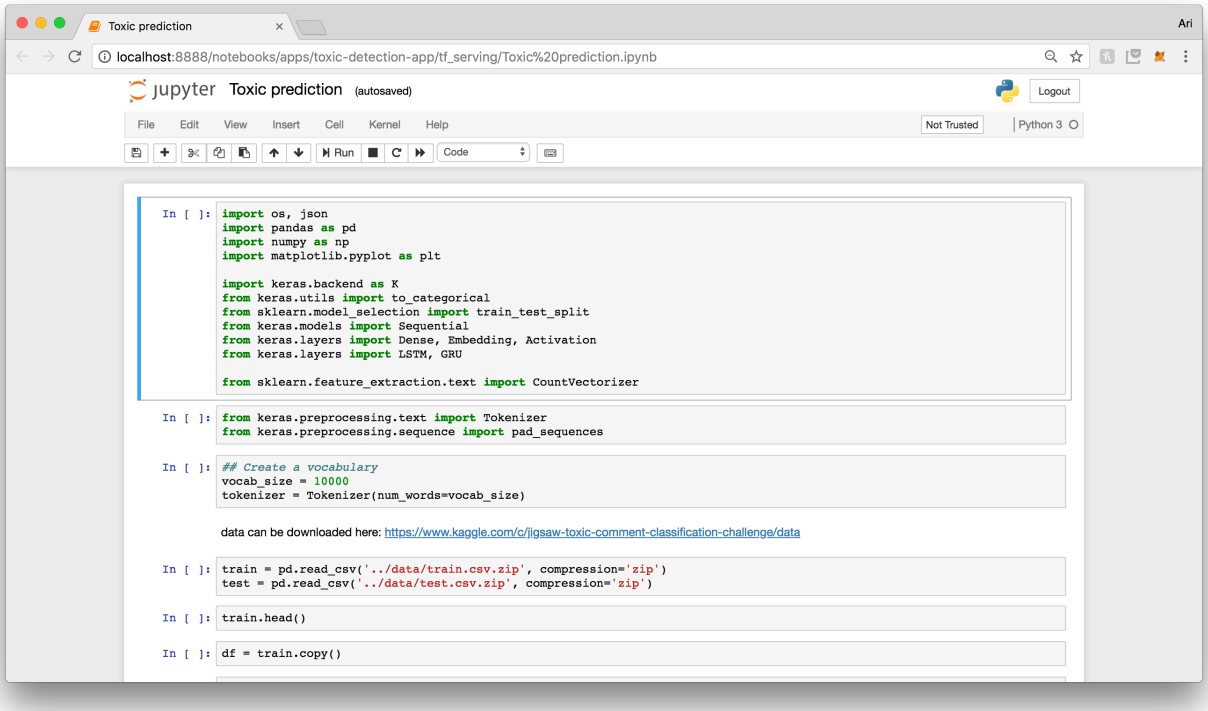


In this application, we have two main moving parts. We have the text box where our users type their text. As they type their text, we’ll *visually* show the user how toxic our neural network detects the language to be.

Neural Network

We’ll write the actual prediction handling by writing a recurrent neural network using the incredibly powerful keras high-level neural network library for Python.

We’ll walk through the basics of building our neural network using publicly available (although you’ll have to download them) data.



Tensorflow Serving

In order to serve our application to the front-end, we'll be using the Tensorflow Serving system which is designed for production environments released by Google. Tensorflow Serving provides deep integration with tensorflow and we'll walk through how to convert our Keras models into tensorflow graphs so that we can use the highly-performant Tensorflow Serving system.

Restful API

Finally, because Tensorflow serving *talks over RPC* and not HTTP (although it does understand HTTP/2), we'll write a little Restful gateway that will handle accepting HTTP requests from our front-end to forwarding the prediction request to Tensorflow serving.

What you need to know

This application will be written using a combination of JavaScript and Python. Although the two languages have differences, they are fairly similar to each other.

> In our book Zero to Deep Learning, we will have a chapter devoted to teaching Python from the eyes of a JavaScript developer by the time the book goes live. It currently is in pre-release.

We won't be using any complex Pythonic constructs in our application, but if you have a question, feel free to write us at help@zerotodeeplearning.com.

Since we'll be using plain ole' JavaScript, we won't be needing to do much setup with our web application. We'll be using ES6 and the `fetch` api, but we won't need to install any dependencies. We *will* use `yarn/npm` so that we can instantiate a project with developer tooling (to support auto-reloading).

We will use the Google Chrome web browser for development, which is important because the `fetch` api is built-in. Other web browsers may not support it directly, so as to avoid needing to install any other dependencies or drop into a lower-level HTTP api, let's stick to Chrome for development.

We'll be using Anaconda, which is a free Python environment that handles virtual environments seamlessly. If you don't already have Anaconda setup on your computer, now is a good time to install it. For details on installing the Anaconda Python distribution, check out the installation page at https://www.anaconda.com/download.

Building the Recurrent Neural Network

In this section, we will build a similar model that we will use to detect toxic language. We will also learn how to package the model and deploy it in order to serve predictions. Let's get started.

First, we will retrieve the data from the Toxic Comment Classification Challenge Data on Kaggle. This dataset contains text comments that have been flagged with one or more of the following issues: `toxic`, `severe_toxic`, `obscene`, `threat`, `insult` and `identity_hate`.

We will use this data to train a model that is able to perform a binary classification problem to flag messages that contain any of these problems. As you'll see, in order to do this, we will have to do a little bit of pre-processing and manipulation of the dataset.

Once we have trained our model, we will save it to disk, so that we can re-use it later for deployment.

This is the master plan, let's get started, by importing the common libraries `numpy`, `pandas` and `matplotlib`:

```
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
```

Data preparation

Let's start our journey by loading the data. This data can be retrieved at https://www.kaggle.com/c/jigsaw-toxic-comment-classification-challenge/data and it's stored as a zip file. Let's save it to a directory called `data/`.

Next, let's read the zip file and load it using pandas

```
df = pd.read_csv('./data/toxic_comment_train.csv.zip', compression='zip')
```

Let's have a look at the data with the `.head()` and `.info()` methods:

```
df.head()
```

```
In [7]: df.head()
Out[7]:
```

	id	comment_text	toxic	severe_toxic	obscene	threat	insult	identity_hate
0	0000997932d777bf	Explanation\nWhy the edits made under my usern...	0	0	0	0	0	0
1	000103f0d9cfb60f	D'aww! He matches this background colour I'm s...	0	0	0	0	0	0
2	000113f07ec002fd	Hey man, I'm really not trying to edit war. It...	0	0	0	0	0	0
3	0001b41b1c6bb37e	"\nMore\nI can't make any real suggestions on ...	0	0	0	0	0	0
4	0001d958c54c6e35	You, sir, are my hero. Any chance you remember...	0	0	0	0	0	0

And let's check out the info

```
df.info()
```

```
In [9]: df.info()

        <class 'pandas.core.frame.DataFrame'>
        RangeIndex: 159571 entries, 0 to 159570
        Data columns (total 8 columns):
        id               159571 non-null object
        comment_text     159571 non-null object
        toxic            159571 non-null int64
        severe_toxic     159571 non-null int64
        obscene          159571 non-null int64
        threat           159571 non-null int64
        insult           159571 non-null int64
        identity_hate    159571 non-null int64
        dtypes: int64(6), object(2)
        memory usage: 9.7+ MB
```

Target definition

We'll see the training dataset contains almost 160 thousand comments, and for each we have 6 possible binary labels. The first thing we want to do is check how frequent each of the labels is. Let's define a variable with the target columns names:

```
targets = ['toxic', 'severe_toxic', 'obscene',
           'threat', 'insult', 'identity_hate']
```

And then let's sum along the rows to count the non-zero entries:

```
df[targets].sum(axis='rows')

toxic            15294
severe_toxic      1595
obscene           8449
threat             478
insult            7877
identity_hate     1405
dtype: int64
```

Let's see the percentage of the dataset that these labels occupy:

```
df[targets].sum(axis='rows') / len(df)
toxic            0.095844
severe_toxic     0.009996
obscene          0.052948
threat           0.002996
insult           0.049364
identity_hate    0.008805
dtype: float64
```

We'll immediately see that most labels are actually very rare. For example, `threat` only appears in 0.3% of the dataset.

The most common label is `toxic`, which appears in almost 9.5% of the dataset.

To simplify the problem a little, let's build a model that will predict if a comment is marked with any of these flags. Let's define a column called `bad_message` which is `True` if any of the flags are present and `False` otherwise. In other words, we are performing an `OR` condition on the targets.

```
df['bad_message'] = df[targets].any(axis='columns')

df['bad_message'].value_counts() / len(df)
False    0.898321
True     0.101679
Name: bad_message, dtype: float64
```

This column still only accounts for about 10% of the messages, which makes our dataset quite unbalanced. We will have to keep this in mind when we perform the training.

Text encoding

In Chapter 8 of Zero to Deep Learning we have seen several ways of encoding text in order to train a neural network model on it. Here, we will use a simple model with an Embedding followed by a Recurrent Neural Network. We will give you suggestions on other things to try in order to get to a better model.

In order to use an Embedding layer and a Recurrent Neural Network, we need to create a dictionary of the words in the text and then transform each sentence in a sequence of indices in the dictionary.

We will use the `Tokenizer` class from the `keras.preprocessing.text` module:

```
from keras.preprocessing.text import Tokenizer
```

A quick look at the documentation, shows us that the tokenizer can be used with different configurations:

```
Init signature: Tokenizer(
  num_words=None,
  filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n',
  lower=True, split=' ',
  char_level=False,
  oov_token=None, **kwargs)
Docstring:
Text tokenization utility class.
```

In particular notice that we can use the tokenize a sentence using words as tokens or characters as tokens, using the `char_level` flag. The default behavior is to use word tokens, so we'll go ahead and use that.

> TIP: feel free to experiment with character level models. These models are more flexible than word level models because they can generate predictions for any string, even when the words contained were not seen during training, so for example these models can handle typos and misspellings.

Let's initialize the tokenizer with default parameters.

```
tokenizer = Tokenizer()
```

Now let's fit the tokenizer on the text strings

```
tokenizer.fit_on_texts(df['comment_text'])
```

```
len(tokenizer.word_index)
210337
```

It has found 210337 unique words. Most of them are likely very rare words. Can we throw away some of them and reduce the dictionary size? `tokenizer.word_counts` contains the counts of each word appearance. If we impose a minimum frequency of at least 3 appearances, the dictionary size shrinks by a factor 4. It is reasonable to do this, because we don't want to memorize unique words.

> TIP: probably a lot of these unique words contain misspellings of known words. Using a character-based NN instead of a word-based NN may handle these cases in a better way.

```
threshold = 3

vocab_size = len([el for el in tokenizer.word_counts.items() if el[1] > threshold])
vocab_size
52286
```

Let's re-initialize the tokenizer with `num_words=vocab_size`, and process all of our text again:

```
tokenizer = Tokenizer(num_words=vocab_size)
tokenizer.fit_on_texts(df['comment_text'])
```

Now we can use the tokenizer to convert every comment to a sequence of integer indices:

```
sequences = np.array(tokenizer.texts_to_sequences(df['comment_text']))
```

Let's check the longest sequence length as well as the maximum word index:

```
longest_sequence = max([len(seq) for seq in sequences])
longest_sequence
1401
```

```
max_index = max([max(seq) for seq in sequences if len(seq) > 1])
max_index
```

The longest sequence has 1401 words, while the max index correctly corresponds to our choice for `vocab_size` (minus 1 since we index from zero).

Let's create a new Dataframe that contains just the labels and the sequences:

```
data = df[['bad_message']].copy()
data['seq'] = sequences
```

This dataframe object has two columns:

- the label `bad_message`
- and the sequence `seq`

Let's check the `head()` again:

```
data.head()
```

```
In [21]: data.head()
```

Out[21]:

	bad_message	seq
0	False	[688, 75, 1, 126, 130, 177, 29, 672, 4511, 120...
1	False	[52, 2635, 13, 555, 3809, 73, 4556, 2706, 21, ...
2	False	[412, 437, 73, 134, 14, 249, 2, 71, 314, 78, 5...
3	False	[57, 7, 228, 97, 54, 328, 1436, 15, 2133, 7, 6...
4	False	[6, 1677, 19, 29, 3516, 54, 1069, 6, 579, 39, ...

Let's split this data in train and test set, using a test size of 30% using `sklearn`'s `train_test_split` method:

```
from sklearn.model_selection import train_test_split
data_train, data_test =
  train_test_split(data,
                   test_size=0.3,
                   random_state=0,
                   stratify=data['bad_message'])
```

The ratio of labels has been preserved using the `stratify=data['bad_message']` keyword. Let's verify that by checking the percentage counts again:

```
data_train['bad_message'].value_counts() / len(data_train)
False    0.898325
True     0.101675
Name: bad_message, dtype: float64
```

Roughly 10% of the messages are flagged, i.e. our classes are quite unbalanced. There are several ways to deal with unbalanced classes, including downsampling, upsampling, generating synthetic data, applying different weights to the 2 classes and more. In this case we will proceed with the simplest method, i.e. upsampling the minority class, with a minor twist.

Instead of simply copying the sentences marked as `bad_data` multiple times we will proceed by creating training batches with an equal amount of good and bad messages.

Let's start by separating good messages from bad messages.

```
data_train_good_messages =
  data_train[data_train['bad_message'] == False].copy()
data_train_bad_messages =
  data_train[data_train['bad_message'] == True].copy()

data_test_good_messages =
  data_test[data_test['bad_message'] == False].copy()
data_test_bad_messages =
  data_test[data_test['bad_message'] == True].copy()
```

Let's define a `batch_generator` function that takes a random sample of good sentences and bad sentences and combines them in a batch made of 50% good and 50% bad.

We also define a helper function called `random_eliminate` that randomly drops a few elements in the sequence. We do this in order to introduce a little bit of noise in our data, which helps prevent overfitting.

```
from keras.preprocessing.sequence import pad_sequences

## Randomly eliminate some elements in the sequence
def random_eliminate(sequences, max_drop=6):
    seqs = []
    for sequence in sequences:
        r = range(len(sequence))
        keep =  max(1, len(r) - np.random.randint(0, max_drop))
        try:
            keep_idx =
              sorted(np.random.choice(r, keep, replace=False))
            new_sequence =
              [sequence[k] for k in keep_idx]
        except:
            new_sequence = sequence
        seqs.append(new_sequence)
    return seqs

## Combine some good sentences and bad sentences in a batch
## and keep them roughly 50/50
def batch_generator(
  good_seq,
  bad_seq,
  batch_size=256,
  random_drop=True):
    half_batch = batch_size // 2
    if half_batch > min(len(good_seq), len(bad_seq)):
        raise Exception("choose a smaller batch size")

    while True:
        good_batch = good_seq.sample(half_batch).values.copy()
        bad_batch = bad_seq.sample(half_batch).values.copy()

        if random_drop:
            good_batch = random_eliminate(good_batch)
            bad_batch = random_eliminate(bad_batch)

        combined_seq = np.hstack([good_batch, bad_batch])
        X = pad_sequences(combined_seq)
        y = half_batch * [False] + half_batch * [True]

        yield (X, y)
```

We use the `batch_generator` function to create a generator that feeds from the training sequences, with a batch size of 256:

```
batch_size = 256
train_gen = batch_generator(data_train_good_messages['seq'],
                            data_train_bad_messages['seq'],
                            batch_size=batch_size)
```

Now we are ready to build and train a model.

Model

Our model will be fairly simple: an `Embedding` layer as input, a `GRU` layer to deal with sequences and a final `Dense` layer to generate the prediction.

If you're unfamiliar with these layers, we go deep in-depth in Zero to Deep Learning.

> TIP: This model can be extended in several ways, but let's start with a simple one and then iterate.

Let's import the `Sequential` model class and the necessary layers. We also import the `Adam` optimizer:

```
from keras.models import Sequential
from keras.layers import Embedding, Dense, GRU
from keras.optimizers import Adam
```

Let's define the model:

```
model = Sequential()
model.add(Embedding(input_dim=vocab_size, output_dim=16))
model.add(GRU(32, dropout=0.15, recurrent_dropout=0.15))
model.add(Dense(1, activation='sigmoid'))
model.compile(
    optimizer=Adam(lr=0.01),
    loss='binary_crossentropy',
    metrics=['accuracy'])

model.summary()
```

```
In [49]: model = Sequential()
         model.add(Embedding(input_dim=vocab_size, output_dim=16))
         model.add(GRU(32, dropout=0.15, recurrent_dropout=0.15))
         model.add(Dense(1, activation='sigmoid'))
         model.compile(
             optimizer=Adam(lr=0.01),
             loss='binary_crossentropy',
             metrics=['accuracy'])

         model.summary()

         _________________________________________________________________
         Layer (type)                 Output Shape              Param #
         =================================================================
         embedding_2 (Embedding)      (None, None, 16)          836576
         _________________________________________________________________
         gru_2 (GRU)                  (None, 32)                4704
         _________________________________________________________________
         dense_2 (Dense)              (None, 1)                 33
         =================================================================
         Total params: 841,313
         Trainable params: 841,313
         Non-trainable params: 0
         _________________________________________________________________
```

The model has roughly a million parameters.

Using the `batch_generator` function, we also create a batch of 1024 sequences from the test data that we will use for validation during training.

```
for X_val, y_val in batch_generator(
  data_test_good_messages['seq'],
  data_test_bad_messages['seq'],
  batch_size=1024,
  random_drop=False):
    break
```

Now we are ready to train the model. We will use the `.fit_generator` method which takes batches of data from the `train_gen` generator.

Since we are using a generator, the concept of epoch is not well defined any longer, so we'll say an epoch is finished when we have generated enough batches to cover the size of `data_train_bad_messages`, i.e. when our model has probably seen every bad message in the training set once.

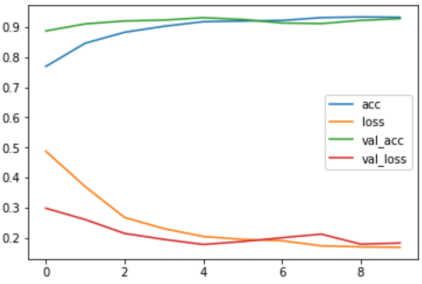
We run the training for 10 such epochs, and we validate the model on the validation batch we've created above.

```
h = model.fit_generator(train_gen,
                        steps_per_epoch=len(data_train_bad_messages) / batch_size,
                        epochs=10,
                        verbose=1,
                        validation_data=(X_val, y_val))


Epoch 1/10
45/44 [==============================] - 61s 1s/step - loss: 0.4878 - acc: 0.7698 - val_loss: 0.2982 - val_acc: 0.8877
Epoch 2/10
45/44 [==============================] - 61s 1s/step - loss: 0.3699 - acc: 0.8470 - val_loss: 0.2604 - val_acc: 0.9111
Epoch 3/10
45/44 [==============================] - 73s 2s/step - loss: 0.2676 - acc: 0.8832 - val_loss: 0.2144 - val_acc: 0.9209
Epoch 4/10
...
Epoch 10/10
45/44 [==============================] - 70s 2s/step - loss: 0.1683 - acc: 0.9331 - val_loss: 0.1823 - val_acc: 0.9287
```

Since the batches are balanced with 50% samples of bad messages and 50% samples of good messages, a score of 90% accuracy is actually quite an acceptable score in this case. We plot the history of training to see if we have converged:

```
dfhistory = pd.DataFrame(h.history)
dfhistory.plot()
```

```
In [32]: dfhistory = pd.DataFrame(h.history)
         dfhistory.plot()

Out[32]: <matplotlib.axes._subplots.AxesSubplot at 0x1a32734908>
```



Finally, let's evaluate the model on the entire test set:

```
X_test = pad_sequences(data_test['seq'])
y_test = data_test['bad_message'].values
```

Let's generate predictions for the entire test set:

```
y_test_pred =
  model.predict_classes(X_test, batch_size=2048, verbose=1)

47872/47872 [==============================] - 73s 2ms/step
```

Then, let's evaluate some standard metrics like `accuracy` and `confusion_matrix`:

```
from sklearn.metrics import accuracy_score, confusion_matrix, classification_report

accuracy_score(y_test, y_test_pred)
0.9150442847593583

confusion_matrix(y_test, y_test_pred)
array([[39355,  3649],
       [  418,  4450]])
```

Finally we print out the `classification_report` and look at `precision` and `recall`:

```
print(classification_report(y_test, y_test_pred))

             precision    recall  f1-score   support

      False       0.99      0.92      0.95     43004
       True       0.55      0.91      0.69      4868

avg / total       0.94      0.92      0.92     47872
```

Our model is not great, as it generates quite a few false positives, but it is good enough for now, so let's go ahead and wrap it up for serving.

> TIP: have a look at some of the more complex models proposed by the participants in the competition. For example this one, this one and this one. These models yield a better performance but are much bigger and longer to train.

In the next section we will package the model and get it ready for serving.

Preparing our models

In order to get our model up into production and ready for use, we'll need to take our keras models and convert them over to tensorflow-compatible models.

Converting our `keras` model into a `tensorflow` compatible model can be pretty sticky, but we can work through this together.

Before we do, however, let's discuss the multiple methods we can use to export our keras models. In no particular order, three popular possible methods are:

- Take snapshots throughout training and save after each epoch
- Exporting the model using the tensorflow estimator.
- Convert the model into JSON and save the final weights
- Freeze the model and save to a tensorflow graph

We'll take arguably the simplest method for exporting our model directly through keras itself. The keras model itself has a `.save()` method available directly on the instance. Let's use this to save the weights *after* we've trained the model:

```
dest_dir = "./result"
model_name = "toxic_model"
model_basename = os.path.join(dest_dir, model_name)
model.save(model_basename + ".h5")
print("Saved model to disk")
```

This will save our model into the `./result` directory under the name `toxic_model.h5`. Let's add this code to the *end of our model* training (that is, after the `model.fit()` method call.

For example purposes, we can also export our model into a JSON structure that allows us to save and rehydrate our model simply across systems using JSON. Exporting our model into JSON might look like:

```
dest_dir = "./result"
model_name = "toxic_model"
model_basename = os.path.join(dest_dir, model_name)

model_json = model.to_json()
with open(model_basename + ".json", "w") as json_file: json_file.write(model_json)
model.save_weights(model_basename + ".weights.h5")
print("Saved model to disk")
```

Regardless of the method we use to save our model to the filesystem, this will allow us to rehydrate our model elsewhere and export it to tensorflow.

In a new file, let's rehydrate the model.

> We're moving to a new file to both demonstrate how to rehydrate the model, but we are also detaching the export process from the training.

First, let's set up a few variables. Since we'll be using the `os` package, let's import it the `os` package. Let's also bring in the tensorflow package as well:

```
import os
import tensorflow as tf
```

Next, let's set up our variables and build an `export_path` that comprises the base path, the name of the model, and the version of the model we're serving. Tensorflow serving requires us to place the name of the model and the model version in this way, so that the version is set as a subdirectory of the model name:

```
export_base = "./result"
model_version = 1
model_name = "toxic_model"

export_path = os.path.join(
  export_base, model_name, str(model_version))
print(export_path)
```

Now that our variables are set up, we need to load the saved model into a Tensorflow session.

Reading from the documentation, a Tensorflow session encapsulates the environment in which Tensorflow Operation objects are executed, and Tensor objects are evaluated. Even when we define a model using Keras, the model is actually built as a set of operations on the Tensorflow graph. In order to evaluate these operations we Tensorflow uses a Session object. So, for example, when you call `model.fit` in Keras, behind the scenes Keras tells Tensorflow to open a `tf.Session` and evaluate the operations that form the model.

> TIP: For more details about the Session object, check out the Tensorflow Session documentation page.

In order to load the model we need to explicitly start a `tf.Session`. Here we will open the session directly, later, when we package our code in a script we will actually use a `with` context manager (more on this below). For now, let's open a `tf.Session`:

```
sess = tf.Session()
```

Keras uses Tensorflow as backend, so now we will load the backend as `K` in order to perform a few operations directly on the backend from within Keras. First of all let's load the backend:

```
import keras.backend as K
```

Then let's set the backend session to be the session we just created:

```
K.set_session(sess)
```

Then we load the model from the path where we saved it. First of all let's get the model path:

```
model_path = os.path.join(export_base, model_name + ".h5")
model_path
```

Then let's use the `load_model` function in keras to load the model:

```
from keras.models import load_model

model = load_model(model_path)
```

Next we need to populate the variables in our tensorflow graph. In order to load the variables, we'll need to initialize the global variables. We'll use the `tf.global_variables_initializer()` function which returns a tensorflow operation that initializes global variables.

> TIP: Since the `tf.global_variables_initializers()` function returns an operation, we'll need to execute it through the session, which we can do with the `sess.run()` method.

```
sess.run(tf.global_variables_initializer())
```

In order to set the model to just the prediction (test) phase, we'll need to set it to the value of 0 (zero). If we just wanted to run it on the train phase, we'd set the learning phase to 1.

In keras, some layers (like Dropout) behave differently while being trained vs testing, so we'll be sure to set this learning phase before we export our our model.

```
K.set_learning_phase(0)
```

Next, let's load the `SaveModelBuilder` class from the package `tensorflow.python.saved_model.builder`:

```
from tensorflow.python.saved_model.builder import SavedModelBuilder
```

The `SaveModelBuilder` class provides functionality to build a `SavedModel` instance protocol buffer. We'll use this instance to save our model. Before we get that far, let's create the instance:

```
builder = SavedModelBuilder(export_path)
```

In order to build a SavedModel instance, the first meta graph (tensorflow language for our model) must be saved with tags. In order to save our meta graph, we'll need to provide a map of how to call the model from tensorflow serving.

Before we attach this meta graph, we'll need to create a signature map. This is basically a method for pulling in the inputs and outputs of the model which we'll use when we call the model from the front-end.

We'll need to create at least one signature to be able to use the model from within Tensorflow serving:

```
from tensorflow.python.saved_model.signature_def_utils_impl
  import predict_signature_def
signature = predict_signature_def(
    inputs={"inputs": model.input},
    outputs={"outputs": model.output})
```

Finally, we can attach our signature with the builder. As we stated earlier, the first metagraph must be tagged (subsequent graphs can use this tag, but it's not a strict requirement). We'll tag our graph with the `SERVING` tag provided by the Tensorflow serving python package when we attach our meta graph and variables to the builder instance:

```
from tensorflow.python.saved_model import tag_constants
# ...
builder.add_meta_graph_and_variables(
    sess=sess,
    tags=[tag_constants.SERVING],
    signature_def_map={'predict': signature})
```

We can also call `save()` on the builder instance at this point and we'll have the tensorflow `.pb` file ready to deploy to Tensorflow serving:

```
builder.save()
```

Putting all of this together, our complete script looks like

```
import os

from tensorflow.python.saved_model.builder \
    import SavedModelBuilder
from keras.models import load_model
from tensorflow.python.saved_model.signature_def_utils_impl \
    import predict_signature_def
from tensorflow.python.saved_model import tag_constants
import tensorflow as tf

import keras.backend as K

with tf.Session() as sess:
    K.set_session(sess)

    model_path = os.path.join(export_path + model_name + ".h5")
    model = load_model(model_path)

    sess.run(tf.global_variables_initializers())

    K.set_learning_phase(0)

    builder = SavedModelBuilder(export_path)

    signature = predict_signature_def(
        inputs={"inputs": model.input},
        outputs={"outputs": model.output})

    builder.add_meta_graph_and_variables(
        sess=sess,
        tags=[tag_constants.SERVING],
        signature_def_map={
            'predict': signature})

    builder.save()
```

Getting ready for production (deployment)

What good are our models if we can't use them in production? I'd say they aren't good at all, if not…

The remainder of this section is dedicated to building a front-end for our toxic detection application model and shipping it.

In order to provide a stable environment for working with our front-end applications (be it a mobile, web, or desktop application we want to be able to iterate on our model as well as provide a rapid response for our application.

In order to create this environment, we'll create a system with the following requirements:

- Easily update our machine learning models
- Low latency for serving predictions
- Scalable across machines
- Easily allows for us to update to later models of Tensorflow
- Extensible to deploy new models
- Accessible from a web browser
- Allow deployable models without shutting down the server

In order to fulfill these requirements, we'll turn to a system designed by Google itself: Tensorflow Serving.

Tensorflow Serving is a “flexible, high-performance serving system for machine learning models, designed for production environments.” One of the main goals for tensorflow serving is to make it easy to deploy an extensible environment for hosting production machine learning algorithms in production environments.

How does this work? Essentially Tensorflow serving looks at each model as a “Servable” object. Since we want it to be able to serve new versions of our algorithms/models, Tensorflow serving will scan the filesystem/sources for new models and new versions of each model.

Tensorflow is *incredibly* extensible and allows us to extend it in many novel ways, including hosting our models on extenal filesystems, like Amazon S3. For simplicity, we’ll stick to the out-of-the-box version of Tensorflow serving, but more information on building custom servable objects can be found on the tensorflow documentation.

One aspect of Tensorflow serving is that it exposes it’s communication through a gRPC interface, which means we cannot access it directly through our web clients, but instead will have to rely on a middleman for handling these requests.

> Actually, this is a limitation on the browser’s part in that Tensorflow Serving uses HTTP/2 for it’s communication protocols. In the future, this won’t be a limitation in the same way. For the time being, we’ll need to rely on a middleman to handle requests.

For our use, we’ll build a very minimal Flask application to handle these requests from our browser application.

We’ll use the following technologies to build the entire infrastructure:

- Tensorflow serving
- Flask server
- Front-end application

Let’s get started working on these three distinct parts.

Tensorflow serving

Let’s get Tensorflow serving set up. In order to run tensorflow serving, we’ll need to prepare/build or install the binary.

Although the later versions of tensorflow serving have exported binaries for OS package managers (such as `apt` for Ubuntu), we’ll install the tensorflow binaries by building them from source. In this way, we won’t restrict ourselves to a specific hosting OS.

In order to build and install tensorflow serving, we’ll need to install the build tool called bazel. Bazel is a build tool that can be used to build multiple languages and makes it incredibly easy and fast to build and rebuild binaries.

> We *could* use bazel to build our front-end application as well, but our application will be quite simple. Bazel provides a great interface for dealing with build systems.

In order to install Bazel, check out the latest up-to-date instructions for installation on the bazel website at www.bazel.build.

Before we get to actually building Tensorflow Serving, let’s install the python package `tensorflow-serving-api`.

As of the writing of this application, the package is only available for Python 2 and below. Since we’re using Python 3, we’ll need to handle installation slightly differently than we would for a package on PyPi.

First, let’s head to the PyPi website and grab the Python 2 package at https://pypi.python.org/pypi/tensorflow-serving-api/1.5.0 and download it to our local computer. We’ll want to take this python package and install it in our Python system path.

> Note that this is only tested to work with linux environments. For detailed instructions on Windows, check out the documentation.

In order to find our system path, we can run the following in our terminal to reveal the list of search paths:

```
python3 -c "import sys;import pprint; pprint.pprint(sys.path)"
```

We'll choose one of the generic search paths (alternatively, we can install it in the current directory). Let's unzip the `.whl` package and copy it to one of the search paths:

```
unzip tensorflow_serving_api-1.5.0-py2-none-any.whl
# This is the directory I've chosen for my development machine:
cp -R ./tensorflow_serving $HOME/.local/lib/python3.6/site-packages
```

Let's test out our installation by making sure we can import the `tensorflow_serving` package:

```
python3 -c "import tensorflow_serving"
```

Now let's get to building the Tensorflow serving package by cloning the repository. We'll grab it from Github and clone all of the git submodules along with the rest of the source code.

```
git clone --recurse-submodules \
          https://github.com/tensorflow/serving
cd serving/
```

When the `git` clone process completes, we'll have the entire source code for the Tensorflow Serving package. From here, we'll need to build the dependencies required to set up the tensorflow serving model server.

Setting up the Tensorflow dependency is straight-forward. We'll only need to run the `./configure` command inside the `tensorflow/` directory.

```
cd tensorflow/
./configure
cd ..
```

> If you see any errors occur with this process, check the Tensorflow Serving Docs for troubleshooting ideas.

Once the `./configure` command is complete, we'll be ready to build out the Tensorflow model server.

Out of the entire process, this build step will take the longest, so now is a good time to get the coffee brewing. Building tensorflow serving is another straight-forward command:

```
bazel build -c opt tensorflow_serving/...
```

While this command is running, let's discuss the next steps. After Tensorflow serving is built, we'll have a bunch of new directories created in the `serving/` working directory that are all prefixed with `bazel-`.

The `tensorflow_model_server` binary will be built in the `bazel-bin` directory. In order to serve our model through Tensorflow serving, we'll run the binary with a few options. We'll need to include the `model_name` as well as the `model_base_path` (which serves the model version).

For example, for our export_base path with of `$HOME/toxic-detection-app/export/toxic_model/` we can launch the model server like so:

```
bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server\
  --port=9000 \
  --model_name=toxic_model \
  --model_base_path=/Users/auser/toxic-detection-app/export/toxic_model
```

To see all of the possible options available, we can run the model server with the `--help` flag:

```
bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server\
    --help
```

The model server will now serve our model over it's gRPC interface. Notice that this only allows us to serve a single model. Instead of using the command-line flags, let's write a configuration file which will allow us to host multiple models over the model server.

Let's take our `model_name` and `model_base_path` and move those into a configuration file:

```
echo 'model_config_list: {
  config: {
    name: "toxic_model",
    base_path: "/Users/auser/toxic-detection-app/tf_serving/export/toxic_model",
    model_platform: "tensorflow"
  }
}' > serving.conf
```

With our configuration file filled out, let's update the launch command-line to use the configuration file:

```
bazel-bin/tensorflow_serving/model_servers/tensorflow_model_server\
  --port=9000 \
  --model_config_file=./serving.conf
```

Building our REST Api

Tensorflow serving provides a communication interface over gRPC, rather than HTTP (despite gRPC being powered by HTTP/2). The easiest way to support all major browsers, we'll create a middleman gateway to provide a RESTful interface.

As teaching Flask is outside of the scope of this section, let's look at a skeleton of a Flask application. We'll then walk through it and add the parts that we'll use.

Let's get all of the dependencies we'll need for our application. This is a good time to set up our `environment.yml` for our anaconda environment. Let's wrap our dependencies into an anaconda environment.

Create an `environment.yml` and let's add the dependencies there:

```
echo "name: tf_serving_ztdl
channels:
  - anaconda-fusion
  - defaults
```

```
dependencies:
  - pip:
    - flask
    - flask-cors
    - grpcio
    - grpcio-tools
    - keras
    - h5py
    - tensorflow
    - gunicorn
" > environment.yml
```

Now we can create this conda environment by calling:

```
conda env create
```

Running this command creates the environment called `tf_serving_ztdl`. Let's activate the environment and continue working in it.

```
source activate tf_serving_ztdl
```

The initial Flask app that we'll work with looks like:

```
import argparse, os, pickle
from flask import Flask, request, jsonify

application = Flask(__name__)

@application.route("/")
def index():
  return jsonify({"name": "model_server"})


if __name__ == "__main__":

  # argument parser from command line
  parser = argparse.ArgumentParser(add_help=True)

  # set of arguments to parse
  parser.add_argument(
      "--port", type=int, default=9001,
      help="port to run flask server")

  # debug or not
  parser.add_argument(
      "--dev", action='store_true',
      help='Run in debug mode')

  # host
  parser.add_argument(
      '--host', default="0.0.0.0",
      help='host to run flask server')

  # tensorflow server
  parser.add_argument(
      "--server", "-s",
      help='Server for tensorflow')

  # parse arguments
  args = parser.parse_args()
```

```
    # launch flask server accessible from all hosts
    application.run(debug=args.dev, port=args.port, host=args.host)
```

In our flask server, we'll want to make a call to the Tensorflow serving model server instance. We'll create a new route that we can call directly from our front-end application to handle this call.

Before we get to setting up our gRPC, we'll want too make sure we *can* call our Flask server from our front-end application. We'll need to set up the application to accept **Cross-Origin Resource Sharing** (or CORS, for short) requests.

> For more information on what CORS is and why we'll need to use this, check out the MDN guide on CORS at https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS.

Fortunately adding CORS rules to our Flask webserver is straight-forward. Let's add a pip package called `flask-cors`:

```
pip install flash-cors
```

Next we'll integrate the CORS setup in our Flask application. Let's import the CORS method in our flask app and set it up (with the least restrictive settings for our app):

```
# ...
from flask_cors import CORS

application = Flask(__name__)
CORS(application)
```

Now we're ready to build up a predictor model to actually make the connection to our tensorflow serving.

Let's create a new file called `predictor.py`. We'll write our `Predictor` class that will be responsible for making the call to Tensorflow serving inside this class.

```
class Predictor:
    def __init__(self, server="localhost:9000"):
        host, port = server.split(':')
        ## Set up channel to gRPC server
```

Obviously we haven't done very much inside here.

Before we can make a call to the Tensorflow serving server, we'll need to set up a communications channel for it. We'll create a communication channel by using the `grpcio` library.

Let's pull in the `implementations` object from the `grpc.beta` package:

```
from grpc.beta import implementations
```

We'll use the `implementations` object to create an insecure channel which we'll use to create a PredictionService stub using the `tensorflow_serving.apis.prediction_service_pb2` object. Let's first import this object form the `tensorflow_serving.apis` package:

```
from tensorflow_serving.apis import prediction_service_pb2
```

Now, back in our constructor of the `Predictor` class, let's create the channel and the connection stub:

```
from grpc.beta import implementations
from tensorflow_serving.apis import prediction_service_pb2

class Predictor:
    def __init__(self, server="localhost:9000"):
        host, port = server.split(':')
        ## Set up channel to gRPC server
        channel = implementations.insecure_channel(host, int(port))
        self.stub =
        prediction_service_pb2
            .beta_create_PredictionService_stub(channel)
```

Although it seems like a lot of work to create the communication channel, this is hiding away a lot of the implementation challenges of creating a line of communication between the gRPC server (our Tensorflow serving instance) and our `Predictor` instance.

With the channel created, we'll want to set up the method we'll call to communicate with our Tensorflow serving model. We'll call this method `predict_toxicity()`:

```
class Predictor:
    def __init__(self, server="localhost:9000"):
        host, port = server.split(':')
        ## Set up channel to gRPC server
        channel = implementations.insecure_channel(host, int(port))
        self.stub =
        prediction_service_pb2
            .beta_create_PredictionService_stub(channel)

    def predict_toxicity(self, text, model_version=1):
        # Set up our prediction
        # Call to the tensorflow serving instance
        # return the prediction
```

In this method, we'll first clean up our text for our model instance. We'll do this by converting any non-alphanumeric characters into a blank space. This way, we don't clutter our prediction text with things like commas and periods. We can easily clean up the text using the `re` module.

```
import re

class Predictor:
    def __init__(self, server="localhost:9000"):
        host, port = server.split(':')
        ## Set up channel to gRPC server
        channel = implementations.insecure_channel(host, int(port))
        self.stub =
        prediction_service_pb2
            .beta_create_PredictionService_stub(channel)

    def predict_toxicity(self, text, model_version=1):
        text = re.sub(r'\W+', ' ', text)
        # Set up our prediction
        # Call to the tensorflow serving instance
        # return the prediction
```

With that, let's actually take a step back and talk about the tokenizer we created previously. We use the tokenizer from keras to create a list of sequences that we'll want to predict upon. If we want to use the same tokenizer, we should update our training code.

Alternatively, we can create a completely new Tokenizer instance. We'll go for the initial approach and save the tokenizer using the pickle library in our training code.

Let's modify our training notebook with the following code to save the tokenizer and the maxlen to a `pickle` file:

```
import pickle

with open('tokenizer.pickle', 'wb') as handle:
    pickle.dump([
        tokenizer,
        maxlen
    ], handle, protocol=pickle.HIGHEST_PROTOCOL)
```

Once this is set, we'll have a `tokenizer.pickle` file available in the local directory which we can load the instance and use the tokenizer and maxlen values we retrieved from our training notebook.

Back in the `predictor.py` file, we can now load this tokenizer and the maxlen values. Since we'll only need to do this once when the instance loads, let's update our constructor method to include the loading of the `tokenizer.pickle` file:

```
import re

class Predictor:
    def __init__(self, server="localhost:9000"):
        with open('../tf_serving/tokenizer.pickle', 'rb') as f:
          self.tokenizer, self.maxlen = pickle.load(f)
        host, port = server.split(':')
        ## Set up channel to gRPC server
        channel = implementations.insecure_channel(host, int(port))
        self.stub =
            prediction_service_pb2
                    .beta_create_PredictionService_stub(channel)
```

With our tokenizer on our instance, we can decode our predicting text into a tokenizer word index. For each word, we'll want to find the index of the word in the tokenizer.

Let's get our word array set up for the passed in text:

```
class Predictor:
    # ...
    def predict_toxicity(self, text, model_version=1):
        text = re.sub(r'\W+', ' ', text)
        intArray = []
        wordArray = text.lower().split()
        for word in wordArray:
          if word in self.tokenizer.word_index:
            intArray.append(self.tokenizer.word_index[word])
          else:
            print("Not including word: {}".format(word))
```

Great, now our `intArray` variable includes all the words by their tokenized index. We'll need to convert this into a list of padded sequences (thanks to how we trained our tensor with our RNN).

This task is straight-forward again. We'll use the `pad_sequences` method from keras again

```
import numpy as np
from keras.preprocessing.sequence import pad_sequences

class Predictor:
    # ...
    def predict_toxicity(self, text, model_version=1):
        text = re.sub(r'\W+', ' ', text)
        intArray = []
        wordArray = text.lower().split()
        for word in wordArray:
          if word in self.tokenizer.word_index:
            intArray.append(self.tokenizer.word_index[word])
          else:
            print("Not including word: {}".format(word))

        x = pad_sequences(np.array([intArray]),
              maxlen=self.maxlen)
```

With this, we're ready to actually make the request to our Tensorflow serving model server.

In order to make this request to the tensorflow server, we'll need to create a `PredictRequest` instance from the `predict_pb2` object exported by the `tensorflow_serving.apis` package.

Without this package, we *could* create the request using raw `gRPC` method, but this is a lot more tedious. Instead, we'll use this helper to create the request.

```
# ...
from tensorflow_serving.apis import predict_pb2
# ...

class Predictor:
    # ...
    def predict_toxicity(self, text, model_version=1):
        text = re.sub(r'\W+', ' ', text)
        intArray = []
        wordArray = text.lower().split()
        for word in wordArray:
            if word in self.tokenizer.word_index:
                intArray.append(self.tokenizer.word_index[word])
            else:
                print("Not including word: {}".format(word))

        x = pad_sequences(np.array([intArray]), maxlen=self.maxlen)

        request = predict_pb2.PredictRequest()
        ## Set up the request here
```

Great, now let's set the `name`, the method name (remember the signature we created before… we'll reference this with the method name), and the version of the model we want to call.

```
# ...
from tensorflow_serving.apis import predict_pb2
# ...
```

```
class Predictor:
    # ...
    def predict_toxicity(self, text, model_version=1):
        text = re.sub(r'\W+', ' ', text)
        intArray = []
        wordArray = text.lower().split()

        for word in wordArray:
          if word in self.tokenizer.word_index:
            intArray.append(self.tokenizer.word_index[word])
          else:
            print("Not including word: {}".format(word))

        x = pad_sequences(np.array([intArray]), maxlen=self.maxlen)

        request = predict_pb2.PredictRequest()
        ## Set up the request here
        request.model_spec.name = 'toxic_model'
        request.model_spect.signature_name = 'predict'
        request.model_spec.version.value = model_version
```

Great, the last thing we need to do with our request object is pass in our input (which we called `inputs` when creating the model for Tensorflow serving). We can create a tensor of our input using the `tf` package.

```
# ...
import tensorflow as tf
# ...

class Predictor:
    # ...
    def predict_toxicity(self, text, model_version=1):
        text = re.sub(r'\W+', ' ', text)
        intArray = []
        wordArray = text.lower().split()

        for word in wordArray:
          if word in self.tokenizer.word_index:
            intArray.append(self.tokenizer.word_index[word])
          else:
            print("Not including word: {}".format(word))

        x = pad_sequences(np.array([intArray]), maxlen=self.maxlen)

        request = predict_pb2.PredictRequest()
        ## Set up the request here
        request.model_spec.name = 'toxic_model'
        request.model_spect.signature_name = 'predict'
        request.model_spec.version.value = model_version

        tp = tf.contrib.util.make_tensor_proto(x,
                                               dtype='float32',
                                               shape=[1, x.size])
        request.inputs['inputs'].CopyFrom(tp)
```

Now we're ready to fire off the request to our Tensorflow serving model server. All we need to do is call `.Predict()` on the communication stub. The `.Predict()` method accepts two arguments, the first one being the request object we've created and the second being the timeout time (as a float) to wait for the rpc response. We'll set it to a reasonable 5 seconds:

```
# ...

class Predictor:
```

```
    # ...
    def predict_toxicity(self, text, model_version=1):
        text = re.sub(r'\W+', ' ', text)
        intArray = []
        wordArray = text.lower().split()

        for word in wordArray:
          if word in self.tokenizer.word_index:
            intArray.append(self.tokenizer.word_index[word])
          else:
            print("Not including word: {}".format(word))

        x = pad_sequences(np.array([intArray]), maxlen=self.maxlen)

        request = predict_pb2.PredictRequest()
        ## Set up the request here
        request.model_spec.name = 'toxic_model'
        request.model_spect.signature_name = 'predict'
        request.model_spec.version.value = model_version

        tp = tf.contrib.util.make_tensor_proto(x,
                                               dtype='float32',
                                               shape=[1, x.size])
        request.inputs['inputs'].CopyFrom(tp)

        return self.stub.Predict(request, 5.0)
```

With our `predictor.py` complete, we can now create a new Flask endpoint.

Let's first instantiate a `predictor` instance before we start up our flask app. This way we can use it in any of our endpoint calls and avoid loading from the `tokenizer.pickle` method multiple times (inefficient, otherwise).

In our `app.py`, let's add the `Predictor` instance:

```
# ...
from predictor import Predictor

predictor = Predictor()
```

We'll our new endpoint at the path or `/predict` and we'll only accept a `POST` HTTP method type.

```
@application.route('/predict', methods=['POST'])
def predict():
    pass
```

Let's fill out the rest of the method by grabbing the version of the model we want to request (defaulting to `1`) and the text (we'll store in a data key called `q`):

```
@application.route('/predict', methods=['POST'])
def predict():
    try:
        json = request.get_json()
        model_version = json.get('version', 1)
        text = json.get('q')

        ## Make the request
    except Execption as e:
```

```
        return jsonify({
            "status": "error",
            "error": str(e)
        })
```

The last thing we need to do is call out to our `predictor` instance to predict the toxicity and then return the value as JSON.

Our completed method then looks like:

```
@application.route('/predict', methods=['POST'])
def predict():
    try:
        json = request.get_json()
        model_version = json.get('version', 1)
        text = json.get('q')

        ## Make the request
        toxicity = predictor.predict_toxicity(
            text,
            model_version=model_version
        ).outputs['outputs'].float_val
        return jsonify({
            "status": "ok",
            "text": text,
            "toxicity":: str(toxicity[0])
        })

    except Execption as e:
        return jsonify({
            "status": "error",
            "error": str(e)
        })
```

Since our front-end only cares about the first (and highest) prediction value, we'll just return the first value as a float.

Let's boot up our server by running the application directly:

```
python app.py
```

Now we'll have a running restful endpoint at http://localhost:9001.

Let's test our setup between the rest server and the Tensorflow serving model server using `curl`.

Let's add the following in our terminal:

```
curl "http://localhost:9001/predict" \
    -XPOST \
    -H"Content-type: application/json" \
    -d'{"q":"You are an awesome person"}'
```

Although your toxicity response will be different than mine, you should see a JSON response value similar to:

```
{
```

```
  "status": "ok",
  "text": "You are an awesome person",
  "toxicity": "0.2245955765247345"
}
```

Now we have our restful endpoint completed. We can finally start implementing the front-end.

Building our front-end

Let's build our front-end application now. Since we don't want to rely on any specific web-server, let's create a raw-javascript based web application that requires no specific web framework.

If you're unfamiliar with JavaScript or any JavaScript frameworks, this part might be a bit more confusing than previous sections. However, JavaScript and Python share a lot in common. The biggest difference between the two is that we'll be interacting with the web browser which is event-based. We'll be interacting with the DOM. For more information about the Document Object Model, check out the Mozilla documentation at https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction.

We'll use a few minor tools to make the development easy, so we will create an `npm` project.

Let's navigate to a directory where we'll create our front-end and run either `yarn init` or `npm init`:

```
yarn init # or npm init
```

We can just answer yes for all of the defaults (we can change these values later in the `package.json` file).

We'll use a webserver tool for running our server as well as including the semantic-ui web framework. In the directory with our newly created `package.json`, let's install the following npm packages:

- live-server - a package that runs a webserver and reloads the browser upon any changes
- semantic-ui - the semantic-ui CSS framework

```
yarn add live-server semantic-ui
# Or using npm
# npm install --save live-server semantic-ui
```

When `semantic-ui` installs, it will ask a few questions. Select the default for the first question and `Yes` for the second. Feel free to change these values around, but we recommend accepting the defaults for ease.

Finally, we'll want to build the distribution version of semantic-ui. Let's run the following command:

```
(cd semantic && gulp build)
```

> Note: it's possible to create this entire project without using `npm` or `yarn`, but these tools make it much easier for development purposes.

In order to make use of the `live-server` package, let's update the `package.json` with a new script in the "scripts" section:

```
{
  "name": "toxic-prediction-frontend",
  "version": "1.0.0",
  "main": "index.js",
  "license": "MIT",
  "dependencies": {
    "live-server": "^1.2.0",
    "semantic-ui": "^2.2.14"
  },
  "scripts": {
    "dev": "live-server ."
  }
}
```

With this `package.json` updated with the `dev` script, we can launch our application using the command: `yarn dev` (or alternatively, we can use `npm run dev`).

With our package set up, let's create an `index.html` file in the same directory and fill it up with a basic template. We'll come back and update this shortly.

```
touch index.html
```

Now, let's fill in the base html template:

```
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Toxic prediction</title>

  <link rel="stylesheet" type="text/css" href="semantic/dist/semantic.min.css">
</head>
<body>
  <div class="ui container holder">
    <!-- App will go here -->
  </div>
  <script src="./app.js"></script>
</body>
</html>
```

Our base HTML template sets the usual `<meta />` tags as well as imports the semantic-ui css file (this way our page looks decent initially). We're also including an `app.js` JavaScript file where we'll write our main JavaScript application code. Let's generate this `app.js` file in the same directory as the `index.html` file. We'll come back and fill this out in detail in a bit.

When we run our application, we'll see a blank page at http://localhost:8080 in the browser. Let's download a few assets to create our initial version of our app.

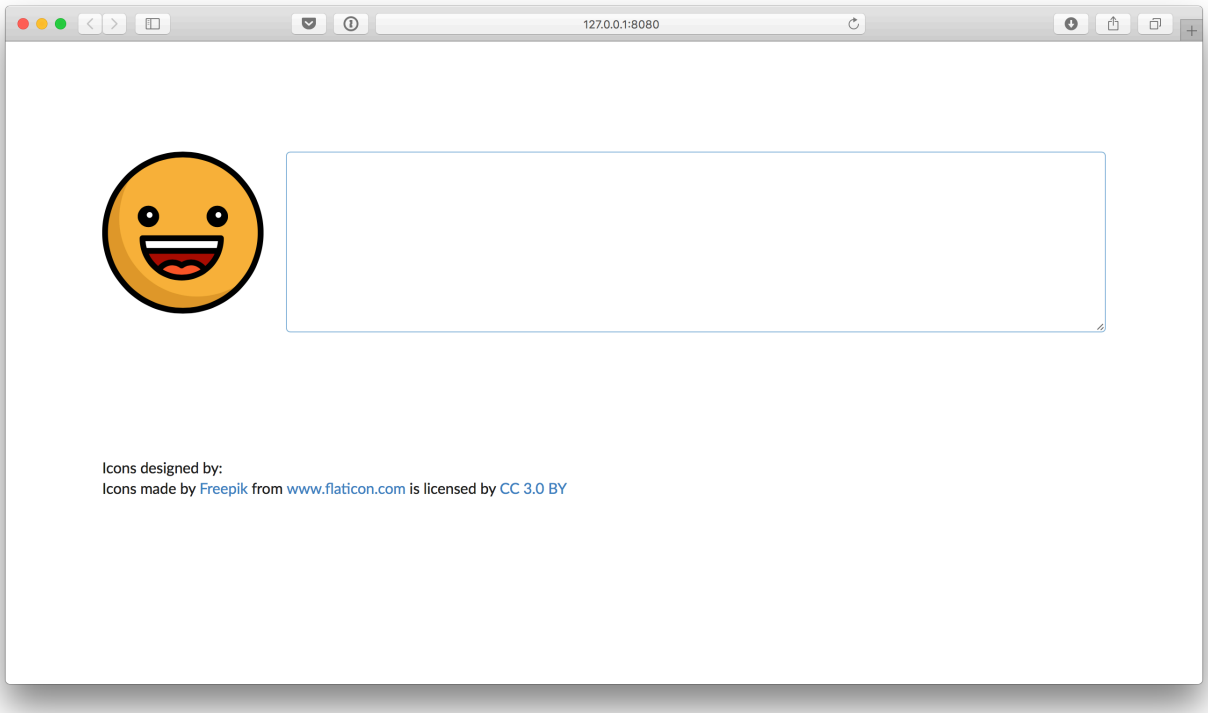
We'll be creating a `<textarea />` input box with an emojii on the side reflecting the "toxicity" of the language in the comment box. Let's head to get some sample emojii icons from www.flaticon.com/. We grabbed a several icons indicating happy, sad, angry, upset, etc. etc. We'll take the `.svg` formats of the icons.

We'll store these packs of icons in a directory in our app directory called `/assets`. Let's unzip these icons into this directory.

Let's use one of our assets and create an initial container that contains our first application framework. The following code (including the attribution) is how our page will look in place of the `<!-- App will go here -->`:

```
  <div class="ui container holder">
    <div class="ui grid middle aligned">
      <div class="eight column wide">
        <div class="ui items">
          <div class="item">
            <a class="ui small image">
              <img src="./assets/003.svg">
            </a>
            <div class="content">
              <div class="ui form">
                <div class="field">
                  <textarea></textarea>
                </div>
              </div>
            </div>
          </div>
        </div>
      </div>
    </div>
  </div>
  <div class="ui container">
    Icons designed by:

    <div>Icons made by <a href="http://www.freepik.com" title="Freepik">Freepik</a> from <a href="https://www.flaticon.com/" title="
"Flaticon">www.flaticon.com</a> is licensed by <a href="http://creativecommons.org/licenses/by/3.0/" title="Creative Commons BY 3.0
" target="_blank">CC 3.0 BY</a></div>
</div>
```

When we load this up, our page (depending upon the icons you pick) will look similar to:



With this setup, let's get to actually building an operating application. In the `app.js`, we'll want to run a function when the page has loaded. Let's create an `onLoad` function as well as add the event listener to listen for the `DOMContentLoaded` event to be fired on the `document` object:

```
const onLoad = function() {
    // Set up our application to run here
    console.log('Page loaded');
}

document.addEventListener('DOMContentLoaded', onLoad);
```

When we reload our page and open the console, we'll see the text: `Page loaded` in the console.

The application work-flow we'll build is as follows:

1. The user types in some text.
2. The application will send a request to the REST server we previously created.
3. The REST server makes a request to Tensorflow serving which is returned to the REST server, which in turns responds to the JavaScript.
4. The JavaScript selects an appropriate icon to display in place of the default one.

Let's make this real. First, we'll need to set up a watcher for the `keydown` event inside the text area. Let's grab a hold of the `<textarea />` and the `<img />` elements in our `onLoad` function:

```
const onLoad = function() {
    const ele = document.querySelector('textarea');
    const img = document.querySelector('img');
    // Do stuff
}
```

Let's set an event listener to listen for the `keydown` event on the `ele` instance.

```
const onLoad = function() {
    const ele = document.querySelector('textarea');
    const img = document.querySelector('img');
    // Do stuff
    ele.addEventListener('keydown', function() {
      const text = ele.value;
      console.log('Event ->', text);
    }
}
```

One potential "bug" in using the direct text as we have it is that we might have to make a JSON request for every keypress. This could make a lot of requests generated and sent really quickly for every single request, especially for a really quick typer.

Instead, let's create a debouncing function that only sends a request for the latest typing event. Let's add a small helper function that handles this for us instead.

To create a debounce function, we'll set a timeout for the event to be fired off. For us, we'll set this to half a second, so when a user has typed and not typed after half a second (500 milliseconds) then we'll run a function.

Let's create a function that creates a timeout for every event and canceled that timeout anytime it has yet to fire before the time has expired. If it has to cancel a timeout, we'll reset a new timeout and wait.

```
const debounceOnKeydown = function(ele, timeout, cb) {
  ele.addEventListener('keydown', function() {
    if (this.timeoutId) {
```

```
      // Clear existing timeout
      window.clearTimeout(this.timeoutId);
    }
    this.timeoutId = window.setTimeout(function() {
      cb(this);
    }, timeout);
  });
}
```

Instead of calling `addEventListener` directly on the `ele` instance, let's update our `onLoad` function to use our new `debounceOnKeydown` function instead.

```
const onLoad = function() {
    const ele = document.querySelector('textarea');
    const img = document.querySelector('img');
    // Do stuff
    debounceOnKeydown(ele, 500, function(that) {
      const text = ele.value;
      console.log('Event ->', text);
    });
}
```

Now we have a reasonable rate where we can expect that we'll fire an event at maximum of half a second interval.

When our user has completed typing a message, we'll want to take the text and wrap it into a `POST` request to our backend server. Let's create a helper function that makes a request to our REST server with the text in the data (as the `q` parameter).

Let's also set our backend to the RESTful server address:

```
const BACKEND = 'http://0.0.0.0:9001';

// ...

const makeRequest = function(path, data) {
    const url = BACKEND + path;
    return fetch(url, {
      method: 'POST',
      body: JSON.stringify({q: data}),
      headers: new Headers({
        'Content-Type': 'application/json'
      })
  }).then(resp => resp.json());
}
```

When we're ready to make our request, we can call this function and expect to get JSON in the following function.

Before we send the request, let's ensure we're not sending an empty string. If we do pass in an empty string *or* a value that we already have the resulting request for, let's set the `img.src` back to it's original value:

```
const onLoad = function() {
    const ele = document.querySelector('textarea');
    const img = document.querySelector('img');

    const initialIcon = img.src;
    let prevText;
    // Do stuff
```

```
    debounceOnKeydown(ele, 500, function(that) {
      const text = ele.value;
      if (text === '') {
        img.src = initialIcon;
      } else if (text !== prevText) {
        // Make request
      }
      prevText = text;
    });
}
```

With this setup, we can make the request to our back-end server and check for the `status` of the response. If it's not "ok", we'll log out the error for further debugging:

```
const onLoad = function() {
    const ele = document.querySelector('textarea');
    const img = document.querySelector('img');

    const initialIcon = img.src;
    let prevText;
    // Do stuff
    debounceOnKeydown(ele, 500, function(that) {
      const text = ele.value;
      if (text === '') {
        img.src = initialIcon;
      } else if (text !== prevText) {
        // Make request
        makeRequest('/predict', text).then(json => {
          if (json.status === 'ok') {
            // We have a toxicity
          } else {
            console.log(resp.error);
          }
        });
      }
      prevText = text;
    });
}
```

Finally, we can select the icon to pick for the specific level of toxicity. We'll want to select an appropriate icon for the level of the toxicity. We'll prefer to show this on an ascending scale of icons, that is we'll have an array of icon assets that range from nicest icon to the most toxic icon. For instance

```
const assets = [
  '002.svg', // nicest emojii
  '003.svg',
  '005.svg',
  '007.svg',
  '008.svg',
  '009.svg'  // most toxic emojii
];
```

Ordering our icons in this way allows us to slide up and down the meanness scale based upon the toxicity ranging between zero and one. In order to slide between the two, we'll want to set an input domain and a output range.

> The functionality we'll build a mini-version of the functionality available in the d3 library of the `d3.scale` functionality. For more information on the `d3.scale` object, check out the docs at http://d3indepth.com/scales/.

Building this scaler is pretty straight-forward. We'll build a function that returns a function which accepts a range and a domain (input/output) which returns a reasonable selection between the output range:

```
const scaler = (domain, range) => n => {
  const [minD, maxD] = domain;
  const [minR, maxR] = range;
  if (minD <= maxD) {
    if (n <= minD) return minR;
    if (n >= maxD) return maxR;
  } else {
    if (n >= minD) return minR;
    if (n <= maxD) return maxR;
  }
  return (maxR - minR) * ((n - minD) / (maxD - minD)) + minR;
};
```

In order to use this `scaler` function, let's grab the output function (which accepts a single number to scale) by passing in the input domain (between 0 and 1, returned by the estimator from our model) and the output range (between 0 and the number of assets):

```
const assets = [
  '002.svg',
  '003.svg',
  '005.svg',
  '007.svg',
  '008.svg',
  '009.svg'
];

const scale = scaler([0, 1], [0, assets.length]);
```

Now we have a `scale` object that can select an appropriate icon for the appropriate range. The last thing we'll need to do is select an asset based upon the scale we received back. Let's create one more mini helper function that picks out the asset from the asset array based upon the returned value from the `scale()` function:

```
const selectAsset = toxicity => {
    // We'll use Math.floor to confirm it's an integer
    // below the maximum (aka not beyond the array)
    // this can also be achieved by setting the output range to
    // be [0, assets.length - 1]
    const assetIdx = Math.floor(scale(toxicity));
    return assets[assetIdx];
}
```

Let's take the response we received from our request and convert the `toxicity` into a number and select an asset based upon the response:

```
const onLoad = function() {
    const ele = document.querySelector('textarea');
    const img = document.querySelector('img');

    const initialIcon = img.src;
    let prevText;
    // Do stuff
    debounceOnKeydown(ele, 500, function(that) {
      const text = ele.value;
      if (text === '') {
        img.src = initialIcon;
```

```
        } else if (text !== prevText) {
          // Make request
          makeRequest('/predict', text).then(json => {
            if (json.status === 'ok') {
              // We have a toxicity
              const toxicity = Number(json['toxicity']);
              const asset = selectAsset(toxicity);
              img.src = `assets/${asset}`;
            } else {
              console.log(resp.error);
            }
          });
        }
        prevText = text;
      });
  }
```

With that, our application will make the request to our backend after our user types an input response. It's pretty straight-forward and we have our entire application ready and functional.

Congrats!

Conclusion

As we've mentioned several times throughout this article, this is just a sample of the content we've written in Zero to Deep Learning to be the best, most complete resource for web developers to integrate deep learning techniques in our applications.

Feel free to reach out to us at help@zerotodeeplearning.com for any questions or leave a comment below. We'd love to hear your feedback and get your experience with the work above.

Cheers!