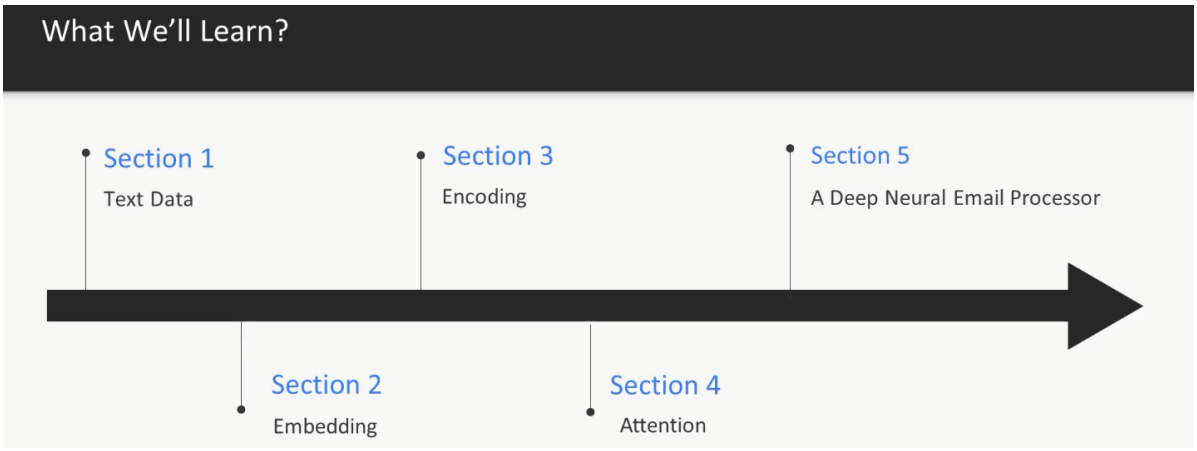
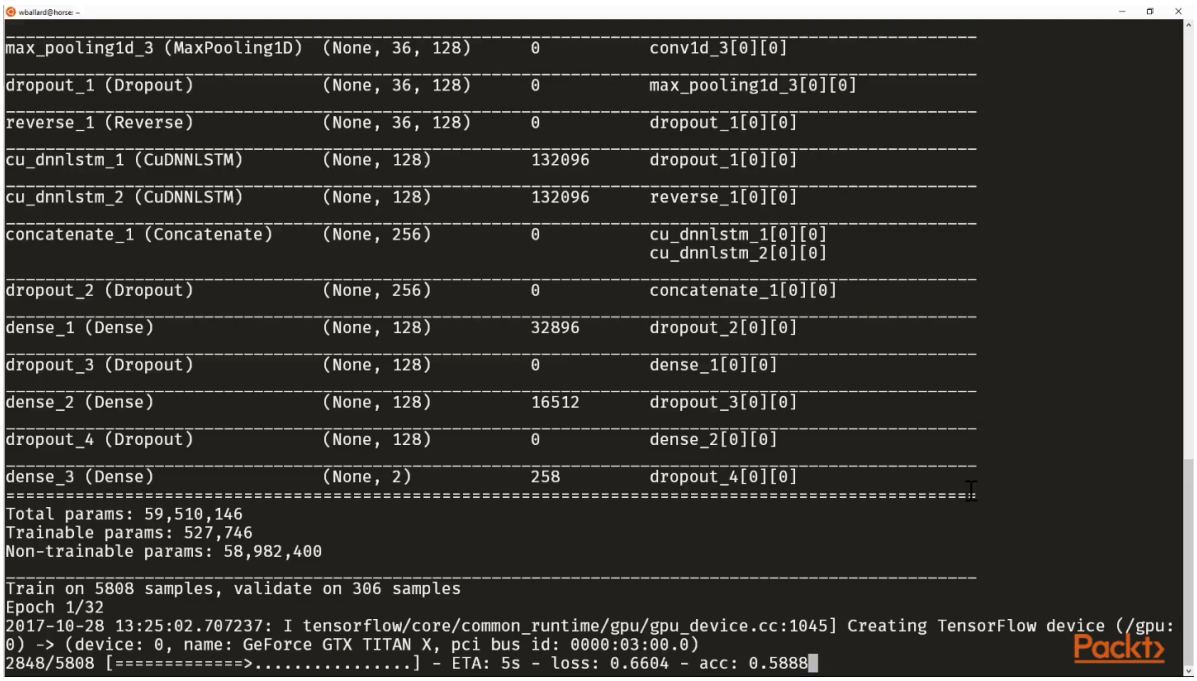
Tensorflow Solutions for Text

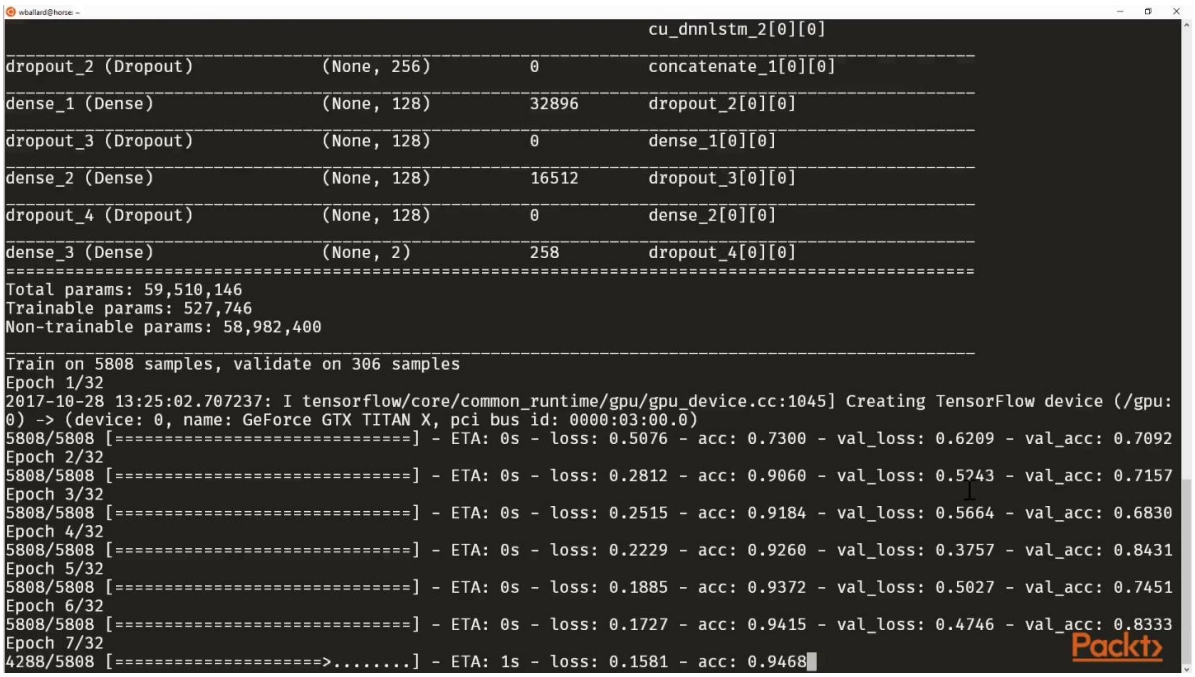
Will Ballard

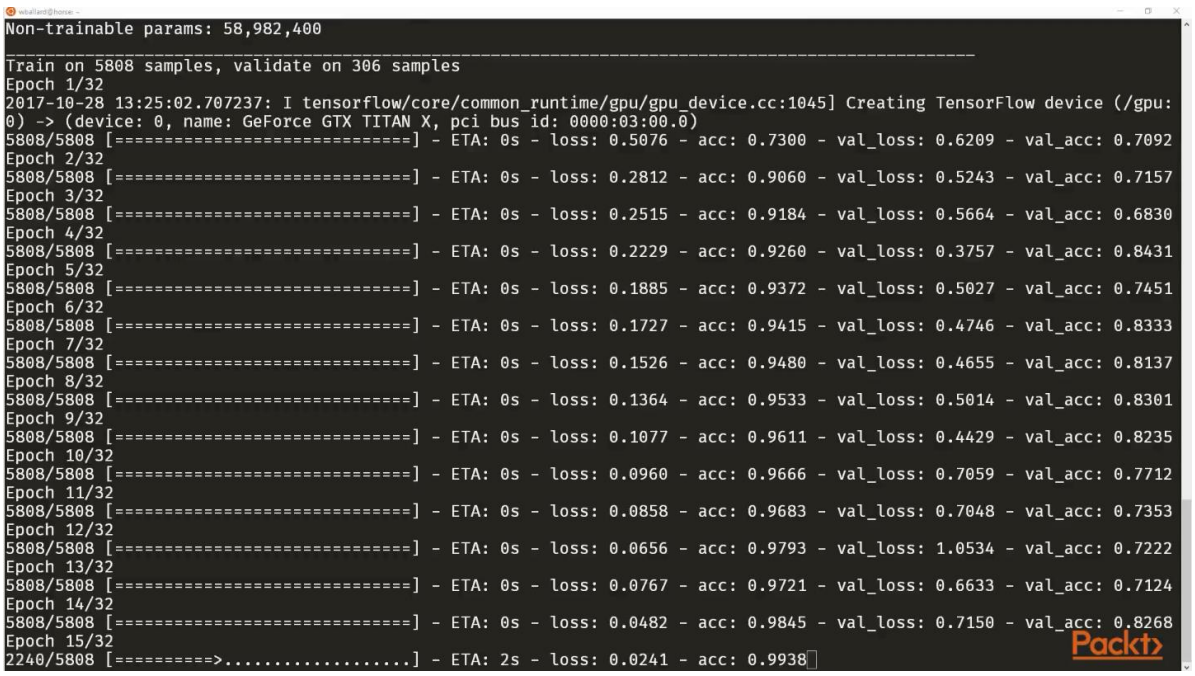
The Course Overview



Embedding data is a method of taking text and converting it into numbers, Encoding is where we start using NNs, Attention is an advanced technique for creating more accurate NNs.




wballard@horse: ~
cu_dnnlstm_2[0][0]
dropout_2 (Dropout) (None, 256) 0 concatenate_1[0][0]
dense_1 (Dense) (None, 128) 32896 dropout_2[0][0]
dropout_3 (Dropout) (None, 128) 0 dense_1[0][0]
dense_2 (Dense) (None, 128) 16512 dropout_3[0][0]
dropout_4 (Dropout) (None, 128) 0 dense_2[0][0]
dense_3 (Dense) (None, 2) 258 dropout_4[0][0]
Total params: 59,510,146
Trainable params: 527,746
Non-trainable params: 58,982,400
Train on 5808 samples, validate on 306 samples
Epoch 1/32
2017-10-28 13:25:02.707237: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1045] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GeForce GTX TITAN X, pci bus id: 0000:03:00.0)
5808/5808 [==========] - ETA: 0s - loss: 0.5076 - acc: 0.7300 - val_loss: 0.6209 - val_acc: 0.7092
Epoch 2/32
5808/5808 [==========] - ETA: 0s - loss: 0.2812 - acc: 0.9060 - val_loss: 0.5243 - val_acc: 0.7157
Epoch 3/32
5808/5808 [==========] - ETA: 0s - loss: 0.2515 - acc: 0.9184 - val_loss: 0.5664 - val_acc: 0.6830
Epoch 4/32
5808/5808 [==========] - ETA: 0s - loss: 0.2229 - acc: 0.9260 - val_loss: 0.3757 - val_acc: 0.8431
Epoch 5/32
5808/5808 [==========] - ETA: 0s - loss: 0.1885 - acc: 0.9372 - val_loss: 0.5027 - val_acc: 0.7451
Epoch 6/32
5808/5808 [==========] - ETA: 0s - loss: 0.1727 - acc: 0.9415 - val_loss: 0.4746 - val_acc: 0.8333
Epoch 7/32
4288/5808 [==========>.....] - ETA: 1s - loss: 0.1581 - acc: 0.9468
Packt>


wballard@horse: ~
Non-trainable params: 58,982,400
Train on 5808 samples, validate on 306 samples
Epoch 1/32
2017-10-28 13:25:02.707237: I tensorflow/core/common_runtime/gpu/gpu_device.cc:1045] Creating TensorFlow device (/gpu:0) -> (device: 0, name: GeForce GTX TITAN X, pci bus id: 0000:03:00.0)
5808/5808 [==========] - ETA: 0s - loss: 0.5076 - acc: 0.7300 - val_loss: 0.6209 - val_acc: 0.7092
Epoch 2/32
5808/5808 [==========] - ETA: 0s - loss: 0.2812 - acc: 0.9060 - val_loss: 0.5243 - val_acc: 0.7157
Epoch 3/32
5808/5808 [==========] - ETA: 0s - loss: 0.2515 - acc: 0.9184 - val_loss: 0.5664 - val_acc: 0.6830
Epoch 4/32
5808/5808 [==========] - ETA: 0s - loss: 0.2229 - acc: 0.9260 - val_loss: 0.3757 - val_acc: 0.8431
Epoch 5/32
5808/5808 [==========] - ETA: 0s - loss: 0.1885 - acc: 0.9372 - val_loss: 0.5027 - val_acc: 0.7451
Epoch 6/32
5808/5808 [==========] - ETA: 0s - loss: 0.1727 - acc: 0.9415 - val_loss: 0.4746 - val_acc: 0.8333
Epoch 7/32
5808/5808 [==========] - ETA: 0s - loss: 0.1526 - acc: 0.9480 - val_loss: 0.4655 - val_acc: 0.8137
Epoch 8/32
5808/5808 [==========] - ETA: 0s - loss: 0.1364 - acc: 0.9533 - val_loss: 0.5014 - val_acc: 0.8301
Epoch 9/32
5808/5808 [==========] - ETA: 0s - loss: 0.1077 - acc: 0.9611 - val_loss: 0.4429 - val_acc: 0.8235
Epoch 10/32
5808/5808 [==========] - ETA: 0s - loss: 0.0960 - acc: 0.9666 - val_loss: 0.7059 - val_acc: 0.7712
Epoch 11/32
5808/5808 [==========] - ETA: 0s - loss: 0.0858 - acc: 0.9683 - val_loss: 0.7048 - val_acc: 0.7353
Epoch 12/32
5808/5808 [==========] - ETA: 0s - loss: 0.0656 - acc: 0.9793 - val_loss: 1.0534 - val_acc: 0.7222
Epoch 13/32
5808/5808 [==========] - ETA: 0s - loss: 0.0767 - acc: 0.9721 - val_loss: 0.6633 - val_acc: 0.7124
Epoch 14/32
5808/5808 [==========] - ETA: 0s - loss: 0.0482 - acc: 0.9845 - val_loss: 0.7150 - val_acc: 0.8268
Epoch 15/32
2240/5808 [==========>.....] - ETA: 2s - loss: 0.0241 - acc: 0.9938
Packt>

replies_ensemble.py examples

```
# one hot encode the labels
targets = replies.one_hot_labels

# now set up sequencing and embedding
sources = replies.texts

HIDDEN = 32
ACTIVATION = 'selu'
INITIALIZER = 'lecun_normal'

# data is loaded and preprocesed, now create a keras model to encode
import keras

inputs = keras.laye                                    xlen,))

# embedding to turn
embedded = replies.trigram.model(inputs)


# plain old dense
dense = keras.layers.Dense(HIDDEN, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(embedded)
dense = keras.layers.Dropout(0.5)(dense)
dense = keras.layers.Dense(HIDDEN, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(dense)
dense = keras.layers.Dropout(0.5)(dense)

# convolution to learn word and phrase like features
conv = keras.layers.Conv1D(HIDDEN, 3, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(embedd
conv = keras.layers.MaxPooling1D(3)(conv)
conv = keras.layers.Conv1D(HIDDEN, 3, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(conv)
conv = keras.layers.MaxPooling1D(3)(conv)

# recurrent with attention, this generates sequences
# note this is GPU only, and keras
# '>=2.0.9, it is shocking slow otherwise
recurrent_forward = keras.layers.CuDNNLSTM(HIDDEN, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(conv)
recurrent_backward = keras.layers.CuDNNLSTM(HIDDEN, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(mailscanner.lay
recurrent = keras.layers.Concatenate()([recurrent_forward, recurrent_backward])
```

trigram

self.trigram = CharacterTrigramEmbedding()

replies_ensemble.py examples

```
embedded = replies.trigram.model(inputs)


# plain old dense
dense = keras.layers.Dense(HIDDEN, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(embedded)
dense = keras.layers.Dropout(0.5)(dense)
dense = keras.layers.Dense(HIDDEN, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(dense)
dense = keras.layers.Dropout(0.5)(dense)

# convolution to learn word and phrase like features
conv = keras.layers.Conv1D(HIDDEN, 3, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(embedd
conv = keras.layers.MaxPooling1D(3)(conv)
conv = keras.layers.Conv1D(HIDDEN, 3, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(conv)
conv = keras.layers.MaxPooling1D(3)(conv)

# recurrent with attention, this generates sequences
# note this is GPU only, and keras
# '>=2.0.9, it is shocking slow otherwise
recurrent_forward = keras.layers.CuDNNLSTM(HIDDEN, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(conv)
recurrent_backward = keras.layers.CuDNNLSTM(HIDDEN, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(mailscanner.lay
recurrent = keras.layers.Concatenate()([recurrent_forward, recurrent_backward])

# now attend to the most important
self_attention = mailscanner.layers.TimeDistributedSelfAttention(activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_init
time_attention = mailscanner.layers.SelfAttention(activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIAL

# now make a consistent shape and ensemble together as a stack, using global max pooling
# to take out any remaining time steps and keep the strongest signals
dense = keras.layers.GlobalMaxPooling1D()(dense)
conv = keras.layers.GlobalMaxPooling1D()(conv)
self_attention = keras.layers.GlobalMaxPooling1D()(self_attention)
time_attention = keras.layers.GlobalMaxPooling1D()(time_attention)
ensemble = keras.layers.Concatenate()([dense, conv, recurrent, self_attention, time_attention])
```

replies_ensemble.py examples

```
time_attention = keras.layers.GlobalMaxPooling1D()(time_attention)
ensemble = keras.layers.Concatenate()([dense, conv, recurrent, self_attention, time_attention])


# dense before final output
stack = keras.layers.Dense(HIDDEN, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(ensemble)
stack = keras.layers.Dropout(0.5)(stack)
stack = keras.layers.Dense(HIDDEN, activation=ACTIVATION, kernel_regularizer=keras.regularizers.l2(0.), kernel_initializer=INITIALIZER)(stack)
stack = keras.layers.Dropout(0.5)(stack)

# softmax on two classes -- which map to our 0, 1 one hots
outputs = keras.layers.Dense(2, activation='softmax')(stack)

model = keras.models.Model(inputs=inputs, outputs=outputs)
model.compile(
    loss='categorical_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)
model.summary()

# model was impressive in overfitting -- hitting 100% accuracy in train
# so -- change the mix shift of the data, more for testing, less data for overfitting
# moving even less data to validation, meaning more training data --> less overfitting
model.fit(
    x=sources,
    y=targets,
    validation_split=0.01,
    batch_size=128,
    epochs=256
)
```

Prerequisites

- Experience with command line shell
- Python scripting or application development

Course Goals

- Build machine learning environments
- Understand how to prepare data for machine learning
- Create classical, convolutional and deep neural networks
- Serve neural network models via REST

Embedding

Section 2

In this Section, we are going to take a look at…

- Representing words with embeddings
- Pretrained word embeddings with Keras
- Pretrained character embeddings with Keras
- Pretrained FastText embeddings with Keras

Representing Words with Embeddings

In this Video, we are going to take a look at…

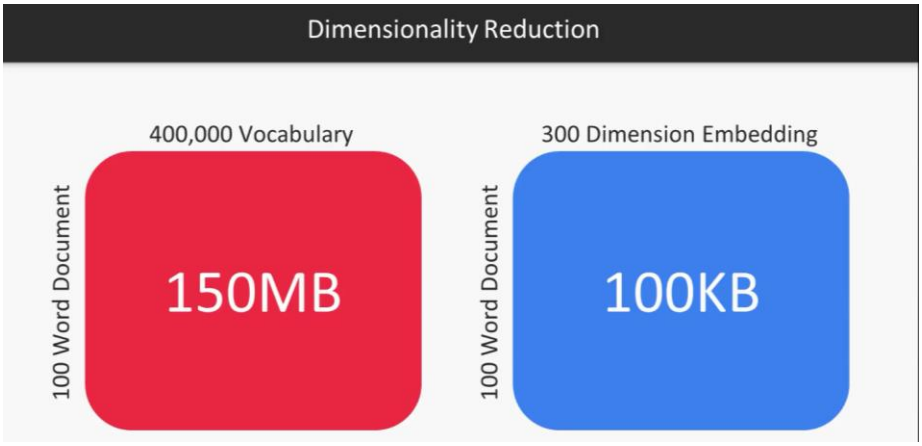
- Why embedding
- Dimensionality reduction
- Embedding algorithms
- How embedding learns

Dimensionality Reduction is related to enhancing computability as well as improving your model's ability to learn.

Why Embed?

Generalization and Computability

- Pure text is a very high dimension space when one-hot, and can easily overfit
- Very large dimension tensors for sequences of words easily exceed memory
- Bag-of-words loses word order!



We want to be able to create models that understand that words exist in a sequence and not just count words like the classic bag of words model

Embedding Algorithms

- GLOVE: https://nlp.stanford.edu/projects/glove/
- Word2Vec: https://arxiv.org/abs/1301.3781
- FastText: https://github.com/facebookresearch/fastText

FastText actually builds relationships out of the character positions inside of words. We generally have a set of about 100 - 300 positions that are the embeddings generated out of the numbers.

Learning Embedding

- All about context, learn the surrounding words
- Given words, predict the next word
- Give words with one removed, fill in the blank
- Given a pair of words, predict the probability of them being near

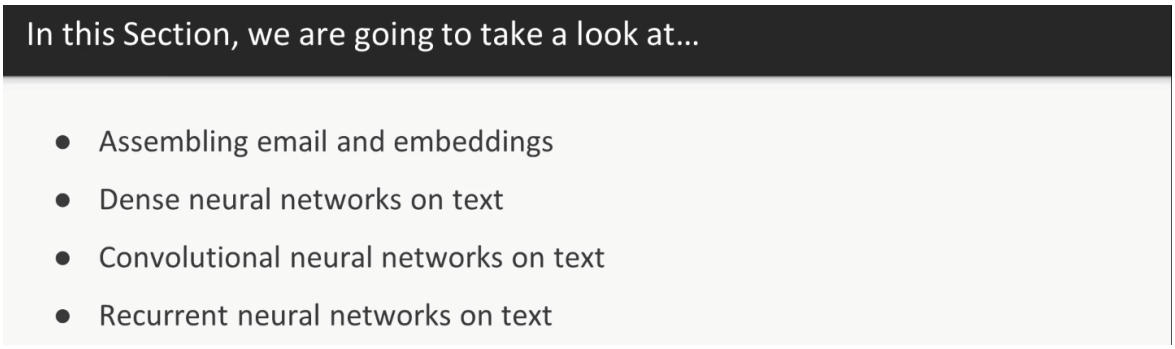
The embedding algorithms generally learn the context of words by building a sort of language model of the surrounding words. There are three main embedding strategies as listed above that provide an unsupervised model approach to train a model that understands language and words.

https://github.com/wballard/vectoria/tree/embedding

Encoding

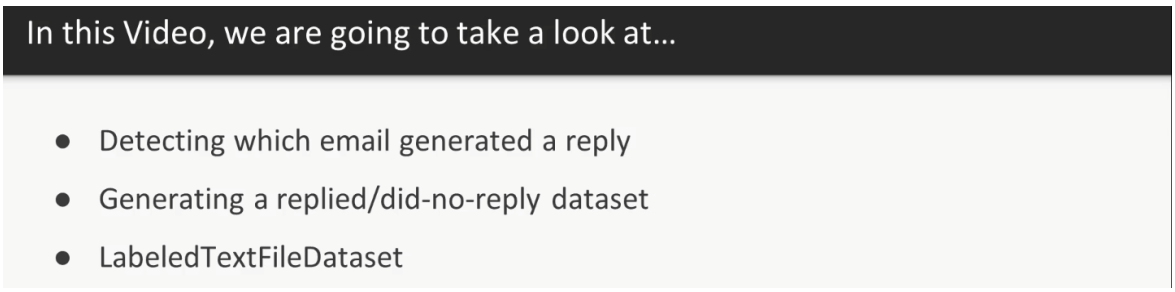
Section 3

We will now move from looking at words to looking at entire emails.



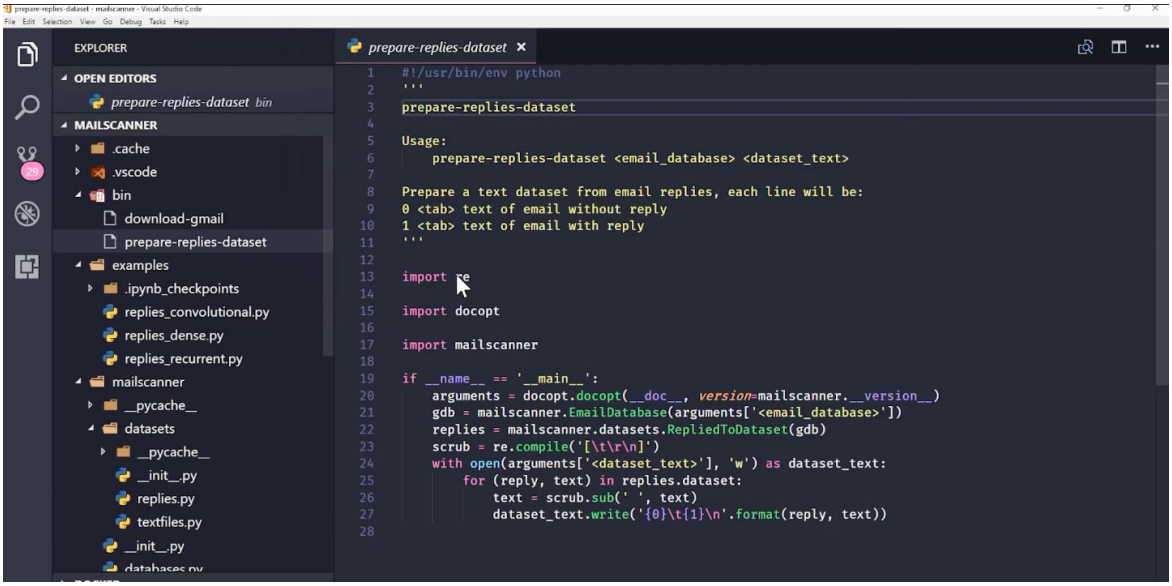
We will take an entire email collection we downloaded earlier and turn that into a dataset for machine learning. We then build 3 different ML networks that attempt to predict whether or not the recipient will reply to an email mainly based on its text.

Assembling Email with Embeddings

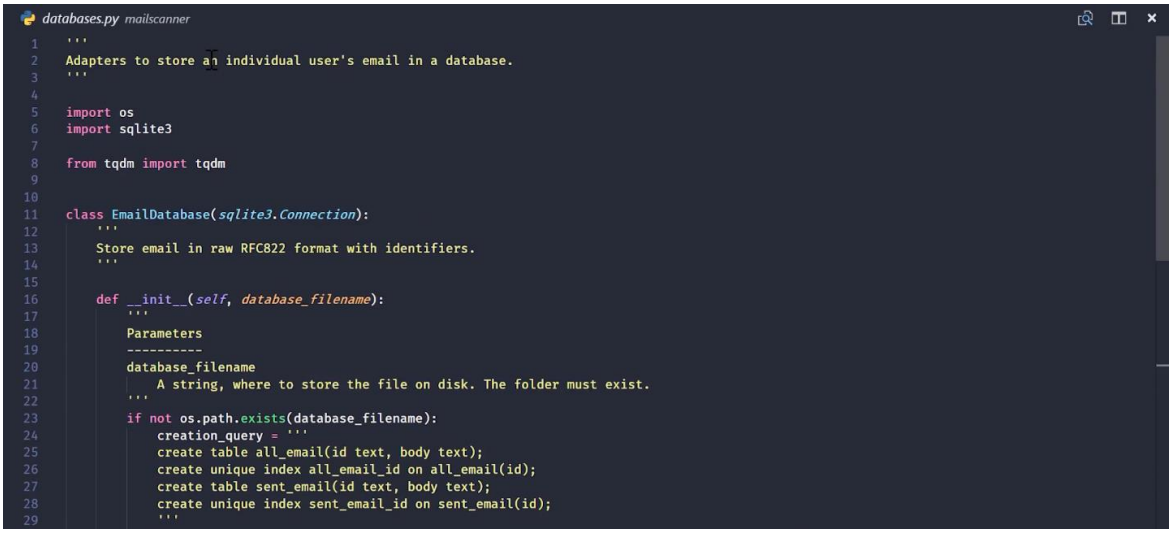


We will look at all the sent emails, look at their identifiers and then find the emails to which they replied. We then generate this list as a text file dataset to which we store the label and all the text in the particular email. We then create a class **LabeledTextFileDataset** that will load these emails into memory in an embedding format.

https://github.com/wballard/mailscanner/tree/section_3



Let us see code to generate a text file from our emails using the tool **docopt**



databases.py mailscanner

```
        Visit all sent emails.

        Parameters
        ----------
        visitor
            A callable that receives each email text.
        '''
        cursor = self.cursor()
        cursor.execute('select count(*) from sent_email')
        count = cursor.fetchall()[0][0]
        cursor.execute('select body from sent_email')
        for row in tqdm(cursor.fetchall(), total=count, desc="Sent", unit='email', disable=(not verbose)):
            visitor(row[0])

    def all(self, visitor, verbose=True):
        '''
        Visit all emails.

        Parameters
        ----------
        visitor
            A callable that receives each email text.
        '''
        cursor = self.cursor()
        cursor.execute('select count(*) from all_email')
        count = cursor.fetchall()[0][0]
        cursor.execute('select body from all_email')
        for row in tqdm(cursor, total=count, desc="All", unit='email', disable=(not verbose)):
            visitor(row[0])
```

replies.py mailscanner\datasets

```
'''
Create a dataset to learn which emails you are likely to reply to.
'''

from ..parser import parse


class RepliedToDataset:
    '''
    Build a dataset of emails that generated replies, along with a balanced number of
    negative samples that did not generate a reply.

    This visits every sent email in the provided database, extracts identifiers of emails that
    generated replies, and hashes them.

    With a hash of all emails that generated replies in hand, all received emails are visited
    in order to extract the text of the email for each reply generating message. The very next
    non-reply-generating message encountered is used as a negative sample to offset and generate
    balanced classes.

    Attributes
    ----------
    dataset
        A list of (Replied|DidNotReply, email text) tuples.
    '''

    def __init__(self, email_database):
        '''
        Parameters
        ----------
        email_database
            Visit this database to create training samples.
        '''
        replied_to = {}

        def is_a_reply(email):
```


replies.py mailscanner\datasets

```
        Parameters
        ----------
        email_database
            Visit this database to create training samples.
        '''
        replied_to = {}

        def is_a_reply(email):
            reply = parse(email).get('In-Reply-To', None)
            if reply:
                replied_to[reply] = True

        email_database.sent(is_a_reply)

        self.dataset = []

        def extract_replies(email):
            email = parse(email)
            if replied_to.get(email.get('Message-ID'), False):
                # a message that generated a reply!
                self.dataset.append(('Replied', ' '.join(map(str, email.values()))))
                return
            # if we get here, this was not a reply, use it as a negative sample
            # if we have an odd number of entries to balance out
            if len(self.dataset) % 2 == 1:
                self.dataset.append(('DidNotReply', ' '.join(map(str, email.values()))))

        email_database.all(extract_replies)
```

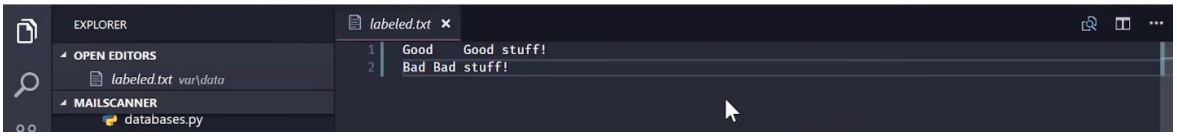
```
wballard (e) root …/git/wballard/mailscanner  section_3
$ ls
bin  examples  gmail.db  mailscanner  mailscanner.egg-info  Makefile  README.md  replies.txt  setup.py  var
wballard (e) root …/git/wballard/mailscanner  section_3
$ ./bin/prepare-replies-dataset ./gmail.db
Usage:
    prepare-replies-dataset <email_database> <dataset_text>
wballard (e) root …/git/wballard/mailscanner  section_3
1 $ ./bin/prepare-replies-dataset ./gmail.db replies.txt
Sent:  14%|                                                  | 1019/7448 [00:01<00:09, 645.64email/s]
```

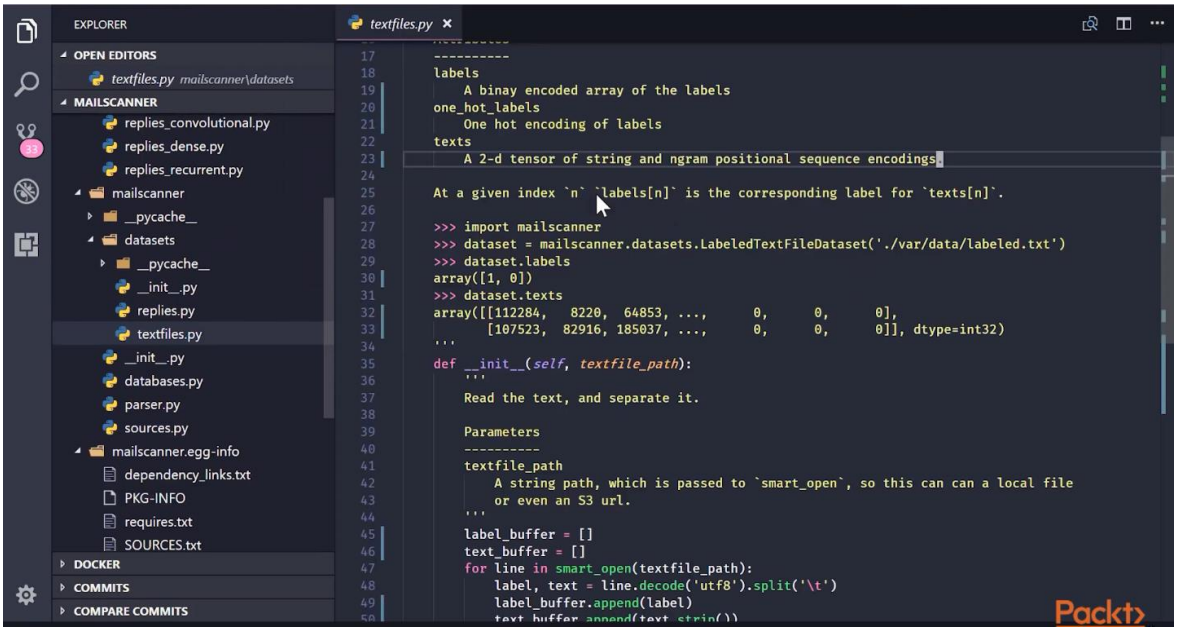
```
wballard (e) root …/git/wballard/mailscanner  section_3
1 $ ./bin/prepare-replies-dataset ./gmail.db replies.txt
Sent: 100%|                                                  | 7448/7448 [00:24<00:00, 298.02email/s]
All:  39%|                                                  | 91627/234594 [04:32<05:33, 428.56email/s]
```

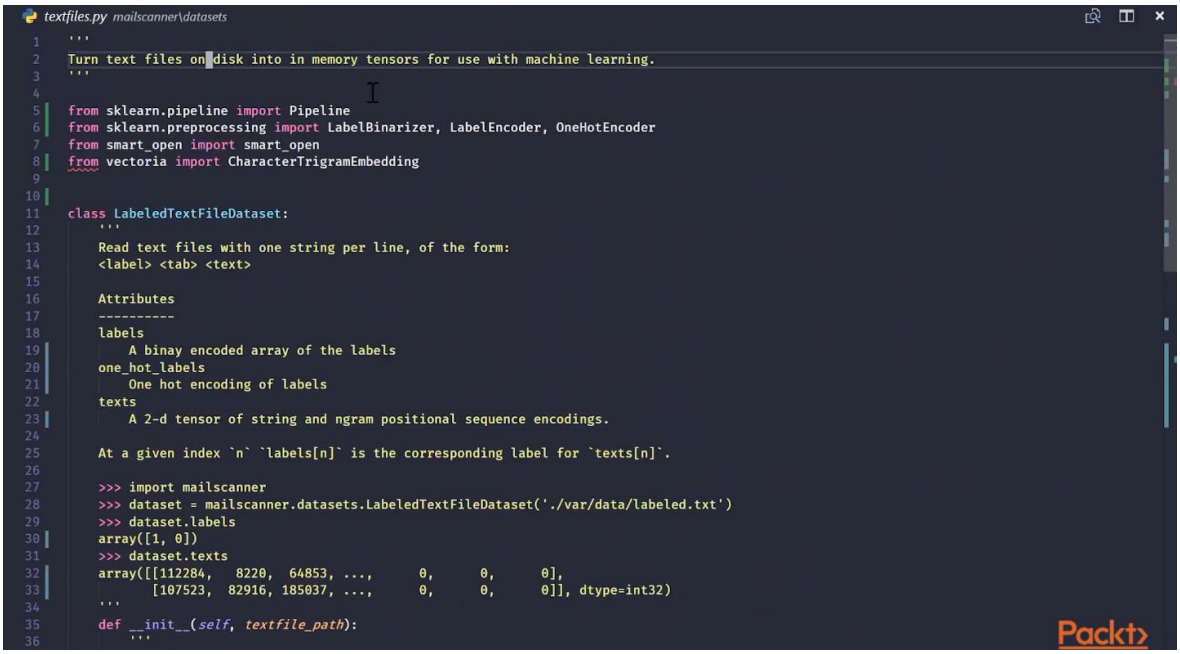
```
wballard (e) root …/git/wballard/mailscanner  section_3
1 $ ./bin/prepare-replies-dataset ./gmail.db replies.txt
Sent: 100%|                                                  | 7448/7448 [00:24<00:00, 298.02email/s]
All: 100%|                                                  | 234594/234594 [11:49<00:00, 330.69email/s]
wballard (e) root …/git/wballard/mailscanner  section_3
$
```

```
wballard (e) root …/git/wballard/mailscanner  section_3
$ head -n 1 ./replies.txt
Replied ('', 'wballard@mailframe.net') by 10.143.31.6 with HTTP; Sun, 15 Jun 2008 18:34:34 -0700 (PDT) <busey@duels.com> neutral (google.com: 72.14.204.169 is neither permitted nor denied by best guess record for domain of busey@duels.com) client-ip=72.14.204.169; mx.google.com; spf=neutral (google.com: 72.14.204.169 is neither permitted nor denied by best guess record for domain of busey@duels.com) smtp.mail=busey@duels.com <42f35d820806151834x70245a08g78e3122c117bf9fd@mail.gmail.com> (2008, 6, 15, 20, 34, 34, 0, 1, -1, -18000) ('Andrew Busey', 'busey@duels.com') ('Will Ballard', 'wballard@mailframe.net') Draft 1.0 multipart/mixed;    boundary="----=_Part_15607_6752670.1213580074705" --   Duels - A Player vs. Player Strategy Game  Challenge me:  www.duels.com/loki  Corporate info:  www.challengegames.com   <br clear="all"><br>-- <br>Duels - A Player vs. Player Strategy Game<br>Challenge me:  <a href="http://www.duels.com/loki">www.duels.com/loki</a><br>Corporate info:  <a href="http://www.challengegames.com">www.challengegames.com</a>
wballard (e) root …/git/wballard/mailscanner  section_3
$
```

We can then check the first line in the replies.txt file







Attention

Section 4

What is attention in a neural network? Demonstrate attention, Learn about the features of a full featured Keras layer and Code Self-Attention.

In this Section, we are going to take a look at...

- SelfAttention
- Time series self attention
- Ensemble networks on text

Self Attention

In this Video, we are going to take a look at…

- What is attention
- Full featured Keras layer implementations
- Self Attention implementation

https://github.com/wballard/mailscanner/tree/attention

Attention!

Focus on the important

Softmax used at the interior of the model

Self normalizing as softmax sum approaches one

Effectively the probability each time step is important, bringing 'attention' to only those steps

Attention is a way to get your model to focus on the most important features.

attention.py mailscanner\layers

```
'''
Layers that implement attention mechanisms.
'''
import keras
from keras import activations, initializers, regularizers, constraints
from keras.engine import InputSpec, Layer
import keras.backend as K


class SelfAttention(Layer):
    """
    Implements an self attention mechanism over time series data, weighting the
    input time series by a learned, softmax scaled attention matrix.

    # Arguments
        activation: Activation function to use
            (see [activations](../activations.md)).
            If you don't specify anything, no activation is applied
            (ie. "linear" activation: `a(x) = x`).
        kernel_initializer: Initializer for the `kernel` weights matrix
            (see [initializers](../initializers.md)).
        kernel_regularizer: Regularizer function applied to
            the `kernel` weights matrix
            (see [regularizer](../regularizers.md)).
        kernel_constraint: Constraint function applied to
            the `kernel` weights matrix
            (see [constraints](../constraints.md)).
    # Input shape
        3D tensor with shape: `(batch_size, time_step, dimensions)
    # Output shape
        3D tensor with shape: `(batch_size, time_step, scaled_dimensions)`.
    """
```

The **keras.backend** gives us direct access to TF's backend.

```
    """

    def __init__(self,
                 activation=None,
                 kernel_initializer='glorot_uniform',
                 kernel_regularizer=None,
                 kernel_constraint=None,
                 **kwargs):
        if 'input_shape' not in kwargs and 'input_dim' in kwargs:
            kwargs['input_shape'] = (kwargs.pop('input_dim'),)
        super(SelfAttention, self).__init__(**kwargs)
        self.activation = activations.get(activation)
        self.kernel_initializer = initializers.get(kernel_initializer)
        self.kernel_regularizer = regularizers.get(kernel_regularizer)
        self.kernel_constraint = constraints.get(kernel_constraint)
        self.input_spec = InputSpec(ndim=3)
        self.supports_masking = True

    def build(self, input_shape):
```

attention.py mailscanner\layers

```
    def build(self, input_shape):
        time_steps = input_shape[1]
        dimensions = input_shape[2]

        self.attention = keras.models.Sequential(name='attention')
        # starting off, each element of the batch is (time_steps, dimensions)
        # turn this into (time_steps, 1)

        # attention matrix, this is the main thing being learned
        self.attention.add(keras.layers.Dense(dimensions,
                                              input_shape=(
                                                  time_steps, dimensions,),
                                              kernel_initializer=self.kernel_initializer,
                                              kernel_regularizer=self.kernel_regularizer,
                                              kernel_constraint=self.kernel_constraint))
        self.attention.add(keras.layers.Activation(self.activation))
        # now convert to an attention vector
        self.attention.add(keras.layers.Dense(1,
                                              kernel_initializer=self.kernel_initializer,
                                              kernel_regularizer=self.kernel_regularizer,
                                              kernel_constraint=self.kernel_constraint))
        # make an attention vector
        self.attention.add(keras.layers.Flatten())
        self.attention.add(keras.layers.Activation('softmax'))
        # repeat this time step weighting for each dimensions
        self.attention.add(keras.layers.RepeatVector(dimensions))
        # reshape to be (time_steps, dimensions)
        self.attention.add(keras.layers.Permute([2, 1]))

        # not using add_weight, so update the weighs manually
        self.trainable_weights = self.attention.trainable_weights
        self.non_trainable_weights = self.attention.non_trainable_weights

        # all done
        self.built = True
```

Packt>

```
        self.built = True

    def call(self, inputs):
        # build the attention matrix
        attention = self.attention(inputs)
        # apply the attention matrix with element wise multiplication
        return keras.layers.Multiply()([inputs, attention])

    def compute_output_shape(self, input_shape):
        # there is no change in shape, the values are just weighted
        return input_shape

    def get_config(self):
        config = {
            'activation': activations.serialize(self.activation),
            'kernel_initializer': initializers.serialize(self.kernel_initializer),
            'kernel_regularizer': regularizers.serialize(self.kernel_regularizer),
            'kernel_constraint': constraints.serialize(self.kernel_constraint),
        }
        return dict(config)
```

Packt>

A Deep Neural Email Processor

Section 5

In this Section, we are going to take a look at...

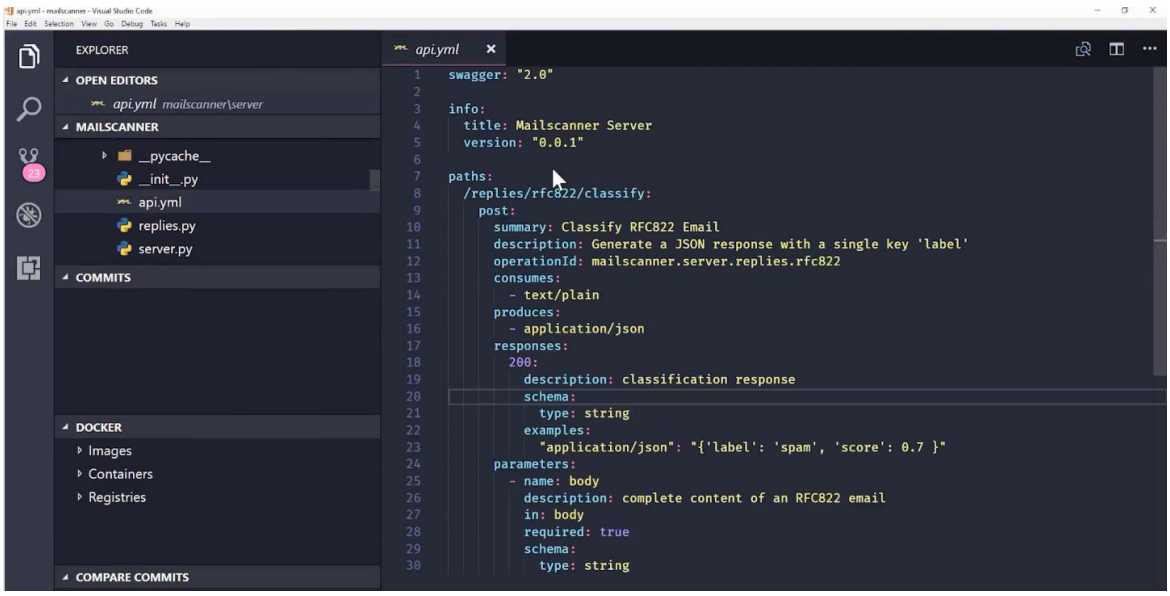
- REST API definition
- Trained models in Docker containers
- Prediction server in Docker containers

REST API Definition

In this Video, we are going to take a look at…

- Getting project source code
- OpenAPI/Swagger with Connexion
- Connexion server
- Connexion handler and model

https://github.com/wballard/mailscanner/tree/server



server.py mailscanner\server

```
'''
Server module, this can be used from the command line or via uWSGI
'''

import os

import connexion
import docopt

import mailscanner

PORT = os.environ.get('PORT', 5000)
DEBUG = os.environ.get('DEBUG', False)

SERVER_IN = os.path.dirname(os.path.abspath(__file__))
# WSGI module level variable
application = connexion.App(__name__, port=PORT, specification_dir=SERVER_IN)
application.add_api('api.yml')
mailscanner.server.replies.load_model_codec(
    os.path.join(SERVER_IN, '../../var/data/replies.weights'),
    os.path.join(SERVER_IN, '../../var/data/replies.pickle')
)

if __name__ == '__main__':
    application.run(debug=DEBUG)

```

replies.py mailscanner\server

```
'''
Individual email handling methods.
'''

from ..datasets import LabeledTextFileDataset
from ..models import Ensemble

# preload this, it has a large tensor inside
# connexion only allows module level functions as handlers
# so module level caching of large data is required
# -- beats loading in on every request!
MODEL = None
CODEC = None

def load_model_codec(path_to_weights, path_to_codec):
    '''
    Load up the module level variables for the codec/dataset
    and the trained machine learning model.
    '''
    global CODEC, MODEL
    print('loading codec from', path_to_codec)
    CODEC = LabeledTextFileDataset.load(path_to_codec)
    print('loading weights from', path_to_weights)
    MODEL = Ensemble(CODEC)
    MODEL.load_weights(path_to_weights)
    print(MODEL.summary())


def rfc822(body):
    '''
    Parameters
    ----------
    body
        An entire RFC822 string.

    Returns
    -------
```


replies.py mailscanner\server

```
    global CODEC, MODEL
    print('loading codec from', path_to_codec)
    CODEC = LabeledTextFileDataset.load(path_to_codec)
    print('loading weights from', path_to_weights)
    MODEL = Ensemble(CODEC)
    MODEL.load_weights(path_to_weights)
    print(MODEL.summary())


def rfc822(body):
    '''
    Parameters
    ----------
    body
        An entire RFC822 string.

    Returns
    -------
    string
        JSON string encoding the classification result.
    '''

    # text, sequenced as ngram, ready to be predicted
    body = body.decode('utf8')
    sequenced = CODEC.trigram.sequencer.transform([body])
    predicted = MODEL.predict(sequenced)
    decode = CODEC.decode_prediction(predicted[0])

    return {
        'label': decode[0],
        # cast off the numpy type
        'score': float(decode[1])
    }

```