# Text Classification, Part I - Convolutional Networks

*Nov 26, 2016*

*6 minute read*

Text classification is a very classical problem. The goal is to classify documents into a fixed number of predefined categories, given a variable length of text bodies. It is widely use in sentimental analysis (IMDB, YELP reviews classification), stock market sentimental analysis, to GOOGLE's smart email reply. This is a very active research area both in academia and industry. In the following series of posts, I will try to present a few different approaches and compare their performances. Ultimately, the goal for me is to implement the paper Hierarchical Attention Networks for Document Classification.

Given the limitation of data set I have, all exercises are based on Kaggle's IMDB dataset. And implementation are all based on Keras.

## Text classification using CNN

In this first post, I will look into how to use convolutional neural network to build a classifier, particularly Convolutional Neural Networks for Sentence Classification - Yoo Kim.

First use BeautifulSoup to remove some html tags and remove some unwanted characters.

```
def clean_str(string):
    """
    Tokenization/string cleaning for dataset
    Every dataset is lower cased except
    """
    string = re.sub(r"\\", "", string)
    string = re.sub(r"\'", "", string)
    string = re.sub(r"\"", "", string)
    return string.strip().lower()


texts = []
labels = []


for idx in range(data_train.review.shape[0]):
    text = BeautifulSoup(data_train.review[idx])
```

```
    texts.append(clean_str(text.get_text().encode('ascii','ignore')))
    labels.append(data_train.sentiment[idx])
```

Keras has provide very nice text processing functions.

```
tokenizer = Tokenizer(nb_words=MAX_NB_WORDS)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)


word_index = tokenizer.word_index
print('Found %s unique tokens.' % len(word_index))


data = pad_sequences(sequences, maxlen=MAX_SEQUENCE_LENGTH)
```

For this project, I have used Google Glove 6B vector 100d. For Unknown word, the following code will just randomize its vector.

```
GLOVE_DIR = "~/data/glove"
embeddings_index = {}
f = open(os.path.join(GLOVE_DIR, 'glove.6B.100d.txt'))
for line in f:
    values = line.split()
    word = values[0]
    coefs = np.asarray(values[1:], dtype='float32')
    embeddings_index[word] = coefs
f.close()


embedding_matrix = np.random.random((len(word_index) + 1, EMBEDDING_DIM))
for word, i in word_index.items():
    embedding_vector = embeddings_index.get(word)
    if embedding_vector is not None:
        # words not found in embedding index will be all-zeros.
        embedding_matrix[i] = embedding_vector
```

## A simplified Convolutional

First, I will just use a very simple convolutional architecture here. Simply use total 128 filters with size 5 and max pooling of 5 and 35, following the sample from this blog

```
sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')
embedded_sequences = embedding_layer(sequence_input)
l_cov1= Conv1D(128, 5, activation='relu')(embedded_sequences)
l_pool1 = MaxPooling1D(5)(l_cov1)
l_cov2 = Conv1D(128, 5, activation='relu')(l_pool1)
l_pool2 = MaxPooling1D(5)(l_cov2)
l_cov3 = Conv1D(128, 5, activation='relu')(l_pool2)
l_pool3 = MaxPooling1D(35)(l_cov3)  # global max pooling
l_flat = Flatten()(l_pool3)
l_dense = Dense(128, activation='relu')(l_flat)
preds = Dense(2, activation='softmax')
```

```
Layer (type)                 Output Shape              Param #     Connected to
====================================================================================================
input_1 (InputLayer)         (None, 1000)              0
____________________________________________________________________________________________________
embedding_1 (Embedding)      (None, 1000, 100)         8057000     input_1[0][0]
____________________________________________________________________________________________________
convolution1d_1 (Convolution1D)  (None, 996, 128)      64128       embedding_1[0][0]
____________________________________________________________________________________________________
maxpooling1d_1 (MaxPooling1D)    (None, 199, 128)      0           convolution1d_1[0][0]
____________________________________________________________________________________________________
convolution1d_2 (Convolution1D)  (None, 195, 128)      82048       maxpooling1d_1[0][0]
____________________________________________________________________________________________________
maxpooling1d_2 (MaxPooling1D)    (None, 39, 128)       0           convolution1d_2[0][0]
____________________________________________________________________________________________________
convolution1d_3 (Convolution1D)  (None, 35, 128)       82048       maxpooling1d_2[0][0]
```

```
_________________________________________________________________
maxpooling1d_3 (MaxPooling1D)    (None, 1, 128)        0         convolution1d_3[0][0]
_________________________________________________________________
flatten_1 (Flatten)              (None, 128)           0         maxpooling1d_3[0][0]
_________________________________________________________________
dense_1 (Dense)                  (None, 128)           16512     flatten_1[0][0]
_________________________________________________________________
dense_2 (Dense)                  (None, 2)             258       dense_1[0][0]
=================================================================
Total params: 8301994
_________________________________________________________________
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
20000/20000 [==============================] - 43s - loss: 0.6347 - acc: 0.6329 - val_loss: 0.6107 - val_acc: 0.7024
Epoch 2/10
20000/20000 [==============================] - 43s - loss: 0.4141 - acc: 0.8188 - val_loss: 0.4098 - val_acc: 0.8180
Epoch 3/10
20000/20000 [==============================] - 43s - loss: 0.3252 - acc: 0.8651 - val_loss: 0.4162 - val_acc: 0.8148
Epoch 4/10
20000/20000 [==============================] - 44s - loss: 0.2651 - acc: 0.8929 - val_loss: 0.3545 - val_acc: 0.8640
Epoch 5/10
20000/20000 [==============================] - 43s - loss: 0.2170 - acc: 0.9140 - val_loss: 0.2764 - val_acc: 0.8906
Epoch 6/10
20000/20000 [==============================] - 43s - loss: 0.1666 - acc: 0.9382 - val_loss: 0.4196 - val_acc: 0.8496
Epoch 7/10
```

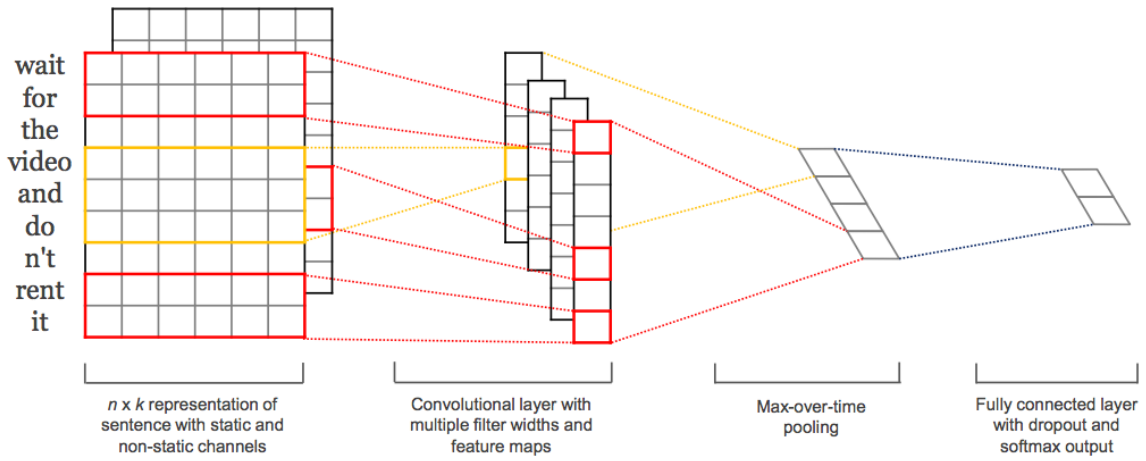
```
20000/20000 [==============================] - 43s - loss: 0.1223 -
acc: 0.9568 - val_loss: 0.4271 - val_acc: 0.8680

Epoch 8/10

20000/20000 [==============================] - 43s - loss: 0.0896 -
acc: 0.9683 - val_loss: 0.8233 - val_acc: 0.8308

Epoch 9/10

20000/20000 [==============================] - 43s - loss: 0.0830 -
acc: 0.9770 - val_loss: 0.5868 - val_acc: 0.8852

Epoch 10/10

20000/20000 [==============================] - 43s - loss: 0.0667 -
acc: 0.9794 - val_loss: 0.5159 - val_acc: 0.8872
```

The accuracy we can achieve is **89%**

## Deeper Convolutional neural network

In Yoon Kim's paper, multiple filters have been applied. This can be easily implemented using Keras Merge Layer.

---




Convolutional network with multiple filter sizes

```
convs = []

filter_sizes = [3,4,5]


sequence_input = Input(shape=(MAX_SEQUENCE_LENGTH,), dtype='int32')

embedded_sequences = embedding_layer(sequence_input)


for fsz in filter_sizes:
```

```
    l_conv = Conv1D(nb_filter=128,filter_length=fsz,activation='relu')
(embedded_sequences)

    l_pool = MaxPooling1D(5)(l_conv)

    convs.append(l_pool)


l_merge = Merge(mode='concat', concat_axis=1)(convs)

l_cov1= Conv1D(128, 5, activation='relu')(l_merge)

l_pool1 = MaxPooling1D(5)(l_cov1)

l_cov2 = Conv1D(128, 5, activation='relu')(l_pool1)

l_pool2 = MaxPooling1D(30)(l_cov2)

l_flat = Flatten()(l_pool2)

l_dense = Dense(128, activation='relu')(l_flat)

preds = Dense(2, activation='softmax')(l_dense)
```

```
____________________________________________________________________________
____________
Layer (type)                 Output Shape              Param #     Connected to
============================================================================
============================================
input_2 (InputLayer)         (None, 1000)              0
____________________________________________________________________________
____________
embedding_2 (Embedding)      (None, 1000, 100)         8057000     input_2[0][0]
____________________________________________________________________________
____________
convolution1d_4 (Convolution1D)  (None, 998, 128)      38528       embedding_2[0][0]
____________________________________________________________________________
____________
convolution1d_5 (Convolution1D)  (None, 997, 128)      51328       embedding_2[0][0]
____________________________________________________________________________
____________
convolution1d_6 (Convolution1D)  (None, 996, 128)      64128       embedding_2[0][0]
____________________________________________________________________________
____________
maxpooling1d_4 (MaxPooling1D)    (None, 199, 128)      0           convolution1d_4[0][0]
```

```
____________________________________________________________________________
_________
maxpooling1d_5 (MaxPooling1D)    (None, 199, 128)      0           convolution1d_5[0][0]
____________________________________________________________________________
_________
maxpooling1d_6 (MaxPooling1D)    (None, 199, 128)      0           convolution1d_6[0][0]
____________________________________________________________________________
_________
merge_1 (Merge)                  (None, 597, 128)      0           maxpooling1d_4[0][0]
                                                                   maxpooling1d_5[0][0]
                                                                   maxpooling1d_6[0][0]
____________________________________________________________________________
_________
convolution1d_7 (Convolution1D)  (None, 593, 128)      82048       merge_1[0][0]
____________________________________________________________________________
_________
maxpooling1d_7 (MaxPooling1D)    (None, 118, 128)      0           convolution1d_7[0][0]
____________________________________________________________________________
_________
convolution1d_8 (Convolution1D)  (None, 114, 128)      82048       maxpooling1d_7[0][0]
____________________________________________________________________________
_________
maxpooling1d_8 (MaxPooling1D)    (None, 3, 128)        0           convolution1d_8[0][0]
____________________________________________________________________________
_________
flatten_2 (Flatten)              (None, 384)           0           maxpooling1d_8[0][0]
____________________________________________________________________________
_________
dense_3 (Dense)                  (None, 2)             770         flatten_2[0][0]
============================================================================
============================================
Total params: 8375850
____________________________________________________________________________
_________
Train on 20000 samples, validate on 5000 samples
Epoch 1/10
```

```
20000/20000 [==============================] - 117s - loss: 0.4950 - acc: 0.7472 - val_loss: 0.2895 - val_acc: 0.8830

Epoch 2/10

20000/20000 [==============================] - 117s - loss: 0.2868 - acc: 0.8807 - val_loss: 0.2460 - val_acc: 0.9036

Epoch 3/10

20000/20000 [==============================] - 118s - loss: 0.2040 - acc: 0.9202 - val_loss: 0.2530 - val_acc: 0.8986

Epoch 4/10

20000/20000 [==============================] - 117s - loss: 0.1293 - acc: 0.9530 - val_loss: 0.2931 - val_acc: 0.8870

Epoch 5/10

20000/20000 [==============================] - 117s - loss: 0.0596 - acc: 0.9788 - val_loss: 0.4155 - val_acc: 0.8896

Epoch 6/10

20000/20000 [==============================] - 117s - loss: 0.0334 - acc: 0.9881 - val_loss: 0.5213 - val_acc: 0.8954

Epoch 7/10

20000/20000 [==============================] - 117s - loss: 0.0173 - acc: 0.9934 - val_loss: 0.5742 - val_acc: 0.8910

Epoch 8/10

20000/20000 [==============================] - 118s - loss: 0.0166 - acc: 0.9949 - val_loss: 0.6220 - val_acc: 0.8944

Epoch 9/10

20000/20000 [==============================] - 117s - loss: 0.0114 - acc: 0.9970 - val_loss: 0.6947 - val_acc: 0.8934

Epoch 10/10

20000/20000 [==============================] - 117s - loss: 0.0095 - acc: 0.9967 - val_loss: 0.8724 - val_acc: 0.8974
```

As you can see, the result slighly improved to **90.3%**

To achieve the best performances, we can 1) fine tune hyper parameters 2) further improve text preprocessing 3) use drop out layer

Full source code is in [my repository in github](#).

# Conclusion

Based on the observation, the complexity of convolutional neural network doesn't seem to improve performance, at least using this small dataset. We might be able to see performance

improvement using larger dataset, which I won't be able to verify here. One observation I have is allowing the embedding layer training or not does significantly impact the performance, same did pretrained Google Glove word vectors. In both cases, I can see performance improved from 82% to 90%.

# Math behind 1D convolution with advanced examples in TF

`To calculate 1D convolution by hand, you slide your kernel over the input, calculate the element-wise multiplications and sum them up.

# The easiest way is for padding=0, stride=1

So if your `input = [1, 0, 2, 3, 0, 1, 1]` and `kernel = [2, 1, 3]` the result of the convolution is `[8, 11, 7, 9, 4]`, which is calculated in the following way:

- *8 = 1 * 2 + 0 * 1 + 2 * 3*
- *11 = 0 * 2 + 2 * 1 + 3 * 3*
- *7 = 2 * 2 + 3 * 1 + 0 * 3*
- *9 = 3 * 2 + 0 * 1 + 1 * 3*
- *4 = 0 * 2 + 1 * 1 + 1 * 3*

TF's conv1d function calculates convolutions in batches, so in order to do this in TF, we need to provide the data in the correct format (doc explains that input should be in `[batch, in_width, in_channels]`, it also explains how kernel should look like). So

```
import tensorflow as tf
i = tf.constant([1, 0, 2, 3, 0, 1, 1], dtype=tf.float32, name='i')
k = tf.constant([2, 1, 3], dtype=tf.float32, name='k')

print i, '\n', k, '\n'

data   = tf.reshape(i, [1, int(i.shape[0]), 1], name='data')
kernel = tf.reshape(k, [int(k.shape[0]), 1, 1], name='kernel')

print data, '\n', kernel, '\n'

res = tf.squeeze(tf.nn.conv1d(data, kernel, 1, 'VALID'))
with tf.Session() as sess:
    print sess.run(res)
```

which will give you the same answer we calculated previously: `[ 8. 11. 7. 9. 4.]`

# Convolution with padding

Padding is just a fancy way to tell append and prepend your input with some value. In most of the cases this value is 0, and this is why most of the time people name it zero-padding. TF support 'VALID' and 'SAME' zero-padding, for an arbitrary padding you need to use tf.pad(). 'VALID' padding means no padding at all, where the same means that the output will have the same size of the input. Let's calculate the convolution with `padding=1` on the same example (notice that for our kernel this is 'SAME' padding). To do this we just append our array with 1 zero at the beginning/end: `input = [0, 1, 0, 2, 3, 0, 1, 1, 0]`.

Here you can notice that you do not need to recalculate everything: all the elements stay the same except of the first/last one which are:

- *1 = 0 * 2 + 1 * 1 + 0 * 3*
- *3 = 1 * 2 + 1 * 1 + 0 * 3*

So the result is `[1, 8, 11, 7, 9, 4, 3]` which is the same as calculated with TF:

```
res = tf.squeeze(tf.nn.conv1d(data, kernel, 1, 'SAME'))
with tf.Session() as sess:
    print sess.run(res)
```

# Convolution with strides

Strides allow you to skip elements while sliding. In all our previous examples we slided 1 element, now you can slide `s` elements at a time. Because we will use a previous example, there is a trick: sliding by `n` elements is equivalent to sliding by 1 element and selecting every n-th element.

So if we use our previous example with `padding=1` and change `stride` to 2, you just take the previous result `[1, 8, 11, 7, 9, 4, 3]` and leave each 2-nd element, which will result in `[1, 11, 9, 3]`. You can do this in TF in the following way:

```
res = tf.squeeze(tf.nn.conv1d(data, kernel, 2, 'SAME'))
with tf.Session() as sess:
    print sess.run(res)
```

## tf.nn.conv1d

```
conv1d(

    value,

    filters,

    stride,
```

```
    padding,

    use_cudnn_on_gpu=None,

    data_format=None,

    name=None

)
```

Defined in tensorflow/python/ops/nn_ops.py.

See the guide: Neural Network > Convolution

Computes a 1-D convolution given 3-D input and filter tensors.

Given an input tensor of shape [batch, in_width, in_channels] if data_format is "NHWC", or [batch, in_channels, in_width] if data_format is "NCHW", and a filter / kernel tensor of shape [filter_width, in_channels, out_channels], this op reshapes the arguments to pass them to conv2d to perform the equivalent convolution operation.

Internally, this op reshapes the input tensors and invokes `tf.nn.conv2d`. For example, if `data_format` does not start with "NC", a tensor of shape [batch, in_width, in_channels] is reshaped to [batch, 1, in_width, in_channels], and the filter is reshaped to [1, filter_width, in_channels, out_channels]. The result is then reshaped back to [batch, out_width, out_channels] (where out_width is a function of the stride and padding as in conv2d) and returned to the caller.

*Args:*

- `value`: A 3D `Tensor`. Must be of type `float32` or `float64`.
- `filters`: A 3D `Tensor`. Must have the same type as `input`.
- `stride`: An `integer`. The number of entries by which the filter is moved right at each step.
- `padding`: 'SAME' or 'VALID'
- `use_cudnn_on_gpu`: An optional `bool`. Defaults to `True`.
- `data_format`: An optional `string` from `"NHWC"`, `"NCHW"`. Defaults to `"NHWC"`, the data is stored in the order of [batch, in_width, in_channels]. The `"NCHW"` format stores data as [batch, in_channels, in_width].
- `name`: A name for the operation (optional).

*Returns:*

A `Tensor`. Has the same type as input.

*Raises:*

- `ValueError`: if `data_format` is invalid.

# Introduction to 1D Convolutional Neural Networks in Keras for Time Sequences

Many articles focus on two dimensional convolutional neural networks. They are particularly used for image recognition problems. 1D CNNs are covered to some extend, e.g. for natural language processing (NLP). Few articles provide an explanatory walkthrough on how to construct a 1D CNN though for other machine learning problems that you might be facing. This article tries to bridge this gap.

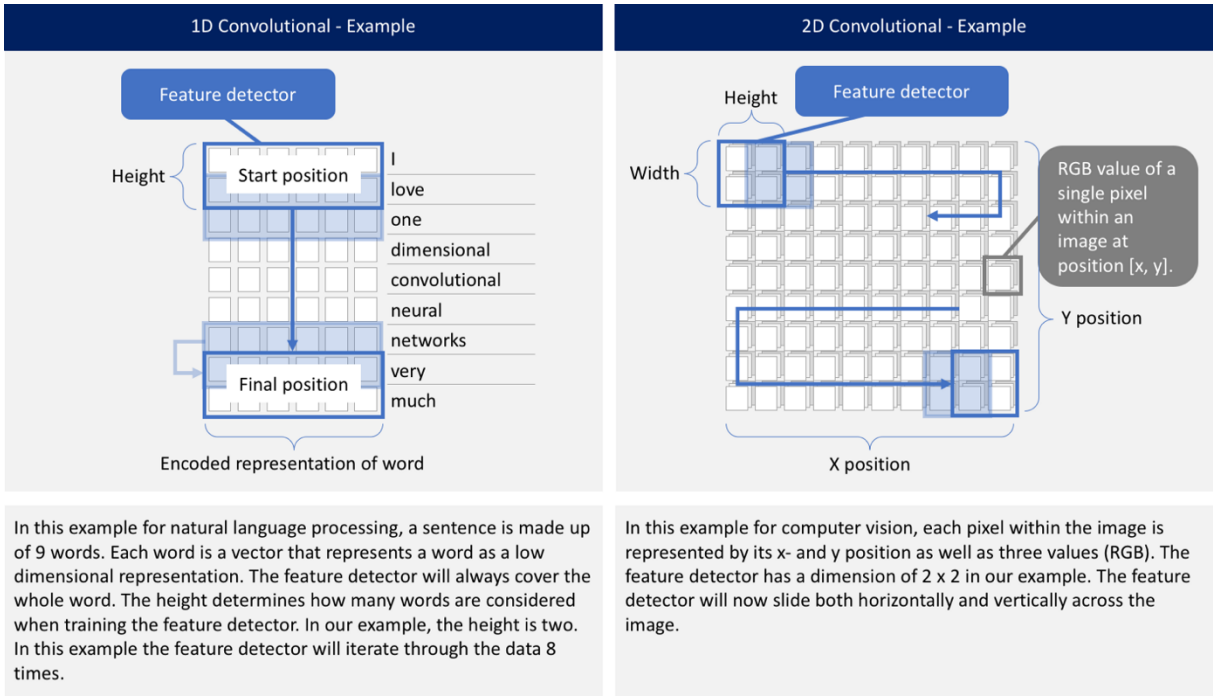
## When to Apply a 1D CNN?

A CNN works well for identifying simple patterns within your data which will then be used to form more complex patterns within higher layers. A 1D CNN is very effective when you expect to derive interesting features from shorter (fixed-length) segments of the overall data set and where the location of the feature within the segment is not of high relevance.

This applies well to the analysis of time sequences of sensor data (such as gyroscope or accelerometer data). It also applies to the analysis of any kind of signal data over a fixed-length period (such as audio signals). Another application is NLP (although here LSTM networks are more promising since the proximity of words might not always be a good indicator for a trainable pattern).

# What is the Difference Between a 1D CNN and a 2D CNN?

CNNs share the same characteristics and follow the same approach, no matter if it is 1D, 2D or 3D. The key difference is the dimensionality of the input data and how the feature detector (or filter) slides across the data:



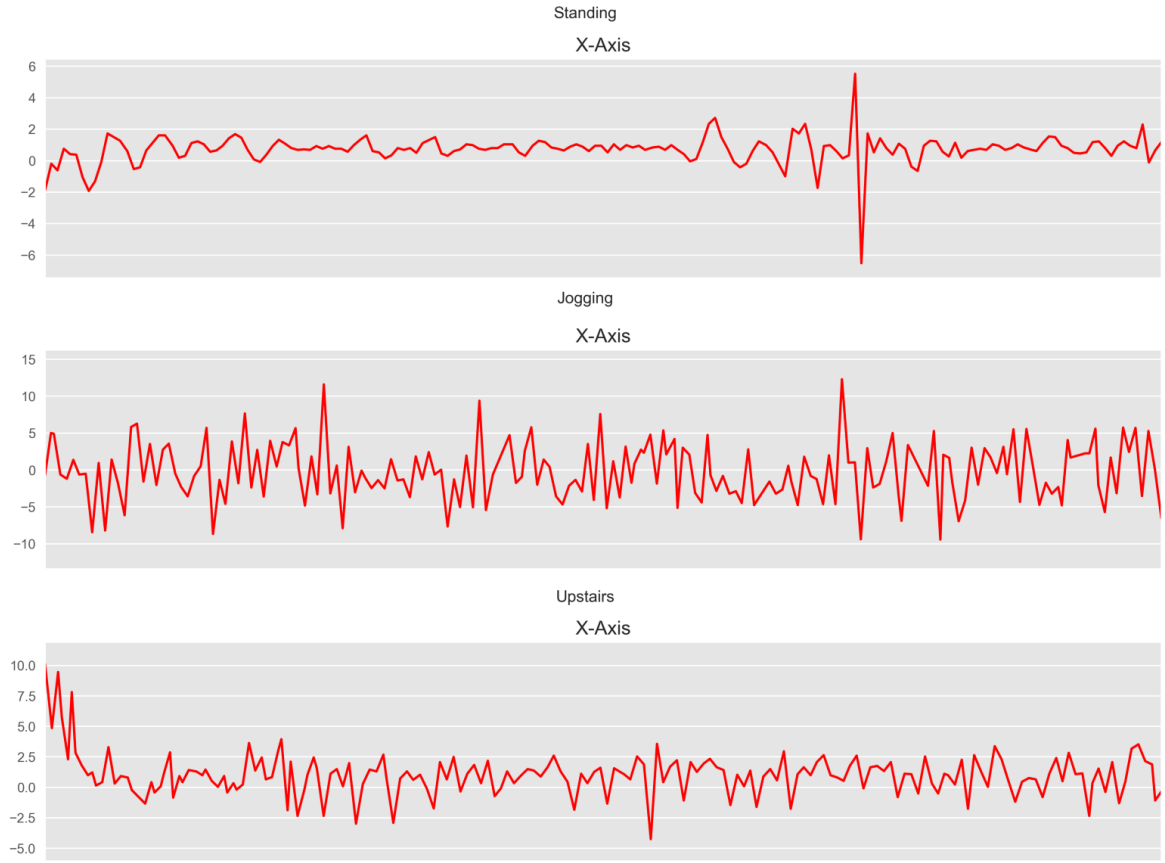

In this example for natural language processing, a sentence is made up of 9 words. Each word is a vector that represents a word as a low dimensional representation. The feature detector will always cover the whole word. The height determines how many words are considered when training the feature detector. In our example, the height is two. In this example the feature detector will iterate through the data 8 times.

In this example for computer vision, each pixel within the image is represented by its x- and y position as well as three values (RGB). The feature detector has a dimension of 2 x 2 in our example. The feature detector will now slide both horizontally and vertically across the image.



# Problem Statement

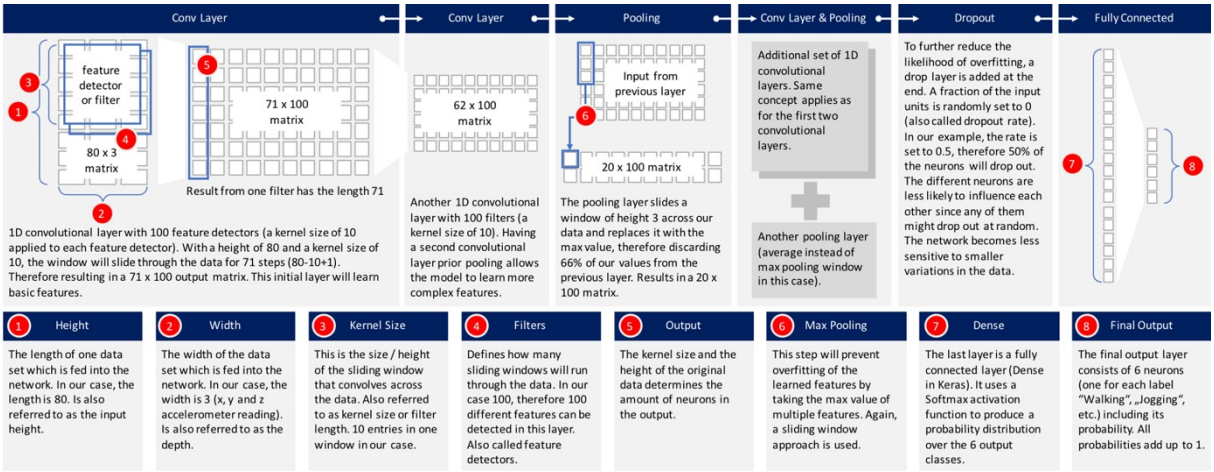
In this article we will focus on time-sliced accelerometer sensor data coming from a smartphone carried by its users on their waist. Based on the accelerometer data of the x, y and z axis, the 1D CNN should predict the type of activity a user is performing (such as “Walking”, “Jogging” or “Standing”). You can find more information in my two other articles here and here. Each time interval of the data will look similar to this for the various activities.



Example time series from the accelerometer data

# How to Construct a 1D CNN in Python?

There are many standard CNN models available. I picked one of the models described on the Keras website and modified it slightly to fit the problem depicted above. The following picture provides a high level overview of the constructed model. Each layer will be explained further.






But let's first take a look at the Python code in order to construct this model:

Running this code will result in the following deep neural network:

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
reshape_45 (Reshape)         (None, 80, 3)             0
_________________________________________________________________
conv1d_145 (Conv1D)          (None, 71, 100)           3100
_________________________________________________________________
conv1d_146 (Conv1D)          (None, 62, 100)           100100
_________________________________________________________________
max_pooling1d_39 (MaxPooling (None, 20, 100)           0
_________________________________________________________________
conv1d_147 (Conv1D)          (None, 11, 160)           160160
_________________________________________________________________
conv1d_148 (Conv1D)          (None, 2, 160)            256160
_________________________________________________________________
global_average_pooling1d_29  (None, 160)               0
_________________________________________________________________
dropout_29 (Dropout)         (None, 160)               0
_________________________________________________________________
dense_29 (Dense)             (None, 6)                 966
=================================================================
Total params: 520,486
Trainable params: 520,486
```

Non-trainable params: 0

_____

None

Let’s dive into each layer and see what is happening:

- **Input data:** The data has been preprocessed in such a way that each data record contains 80 time slices (data was recorded at 20 Hz sampling rate, therefore each time interval covers four seconds of accelerometer reading). Within each time interval, the three accelerometer values for the x axis, y axis and z axis are stored. This results in an 80 x 3 matrix. Since I typically use the neural network within iOS, the data must be passed into the neural network as a flat vector of length 240. The first layer in the network must reshape it to the original shape which was 80 x 3.
- **First 1D CNN layer:** The first layer defines a filter (or also called feature detector) of height 10 (also called kernel size). Only defining one filter would allow the neural network to learn one single feature in the first layer. This might not be sufficient, therefore we will define 100 filters. This allows us to train 100 different features on the first layer of the network. The output of the first neural network layer is a 71 x 100 neuron matrix. Each column of the output matrix holds the weights of one single filter. With the defined kernel size and considering the length of the input matrix, each filter will contain 71 weights.
- **Second 1D CNN layer:** The result from the first CNN will be fed into the second CNN layer. We will again define 100 different filters to be trained on this level. Following the same logic as the first layer, the output matrix will be of size 62 x 100.

- **Max pooling layer:** A pooling layer is often used after a CNN layer in order to reduce the complexity of the output and prevent overfitting of the data. In our example we chose a size of three. This means that the size of the output matrix of this layer is only a third of the input matrix.
- **Third and fourth 1D CNN layer:** Another sequence of 1D CNN layers follows in order to learn higher level features. The output matrix after those two layers is a 2 x 160 matrix.
- **Average pooling layer:** One more pooling layer to further avoid overfitting. This time not the maximum value is taken but instead the average value of two weights within the neural network. The output matrix has a size of 1 x 160 neurons. Per feature detector there is only one weight remaining in the neural network on this layer.
- **Dropout layer:** The dropout layer will randomly assign 0 weights to the neurons in the network. Since we chose a rate of 0.5, 50% of the neurons will receive a zero weight. With this operation, the network becomes less sensitive to react to smaller variations in the data. Therefore it should further increase our accuracy on unseen data. The output of this layer is still a 1 x 160 matrix of neurons.
- **Fully connected layer with Softmax activation:** The final layer will reduce the vector of height 160 to a vector of six since we have six classes that we want to predict (“Jogging”, “Sitting”, “Walking”, “Standing”, “Upstairs”, “Downstairs”). This reduction is done by another matrix multiplication. Softmax is used as the activation function. It forces all six outputs of the neural network to sum up to one. The output value will

therefore represent the probability for each of the six classes.

# Training and Testing the Neural Network

Here is the Python code to train the model with a batch size of 400 and a training and validation split of 80 to 20.

The model reaches an accuracy of 97% for the training data.

```
...
Epoch 9/50
16694/16694 [==============================] - 16s
973us/step - loss: 0.0975 - acc: 0.9683 - val_loss: 0.7468 - val_acc: 0.8031
Epoch 10/50
16694/16694 [==============================] - 17s
989us/step - loss: 0.0917 - acc: 0.9715 - val_loss: 0.7215 - val_acc: 0.8064
Epoch 11/50
16694/16694 [==============================] - 17s
1ms/step - loss: 0.0877 - acc: 0.9716 - val_loss: 0.7233 - val_acc: 0.8040
Epoch 12/50
16694/16694 [==============================] - 17s
1ms/step - loss: 0.0659 - acc: 0.9802 - val_loss: 0.7064 - val_acc: 0.8347
Epoch 13/50
16694/16694 [==============================] - 17s
1ms/step - loss: 0.0626 - acc: 0.9799 - val_loss: 0.7219 - val_acc: 0.8107
```

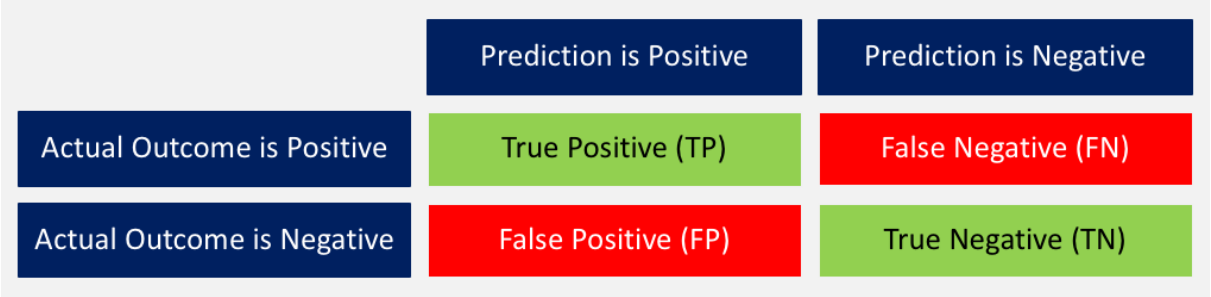
Running it against the test data reveals an accuracy of 92%.

```
Accuracy on test data: 0.92
Loss on test data: 0.39
```

This is a good number considering that we used one of the standard 1D CNN models. Our model also scores well on precision, recall, and the f1-score.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.76 | 0.78 | 0.77 | 650 |
| 1 | 0.98 | 0.96 | 0.97 | 1990 |
| 2 | 0.91 | 0.94 | 0.92 | 452 |
| 3 | 0.99 | 0.84 | 0.91 | 370 |
| 4 | 0.82 | 0.77 | 0.79 | 725 |
| 5 | 0.93 | 0.98 | 0.95 | 2397 |
| avg / total | 0.92 | 0.92 | 0.92 | 6584 |

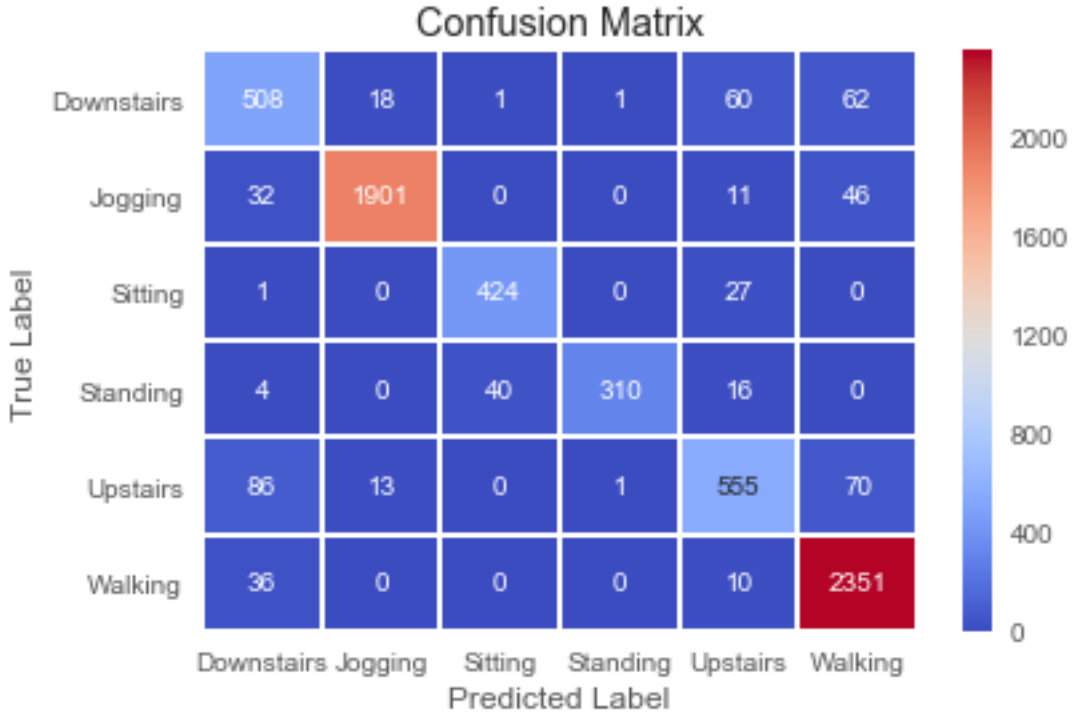
Here is a brief recap of what those scores mean:






- **Accuracy:** The ratio between correctly predicted outcomes and the sum of all predictions. ((TP + TN) / (TP + TN + FP + FN))
- **Precision:** When the model predicted positive, was it right? All true positives divided by all positive predictions. (TP / (TP + FP))
- **Recall:** How many positives did the model identify out of all possible positives? True positives divided by all actual positives. (TP / (TP + FN))
- **F1-score:** This is the weighted average of precision and recall. (2 x recall x precision / (recall + precision))

The associated confusion matrix against the test data looks as following.




# Summary

In this article you have seen an example on how to use a 1D CNN to train a network for predicting the user behaviour based on a given set of accelerometer data from smartphones. The full Python code is available on github.

# Links and References

- Keras documentation for 1D convolutional neural networks
- Keras examples for 1D convolutional neural networks

- A good article with an introduction to 1D CNNs for natural language processing problems

# Disclaimer

*The postings on this site are my own and do not necessarily represent the postings, strategies or opinions of my employer.*

## Getting started with the Keras Sequential model

The `Sequential` model is a linear stack of layers.

You can create a `Sequential` model by passing a list of layer instances to the constructor:

```
from keras.models import Sequential
from keras.layers import Dense, Activation

model = Sequential([
    Dense(32, input_shape=(784,)),
    Activation('relu'),
    Dense(10),
    Activation('softmax'),
])
```

You can also simply add layers via the `.add()` method:

```
model = Sequential()
model.add(Dense(32, input_dim=784))
```

```
model.add(Activation('relu'))
```

## Specifying the input shape

The model needs to know what input shape it should expect. For this reason, the first layer in a `Sequential` model (and only the first, because following layers can do automatic shape inference) needs to receive information about its input shape. There are several possible ways to do this:

- Pass an `input_shape` argument to the first layer. This is a shape tuple (a tuple of integers or `None` entries, where `None` indicates that any positive integer may be expected). In `input_shape`, the batch dimension is not included.
- Some 2D layers, such as `Dense`, support the specification of their input shape via the argument `input_dim`, and some 3D temporal layers support the arguments `input_dim` and `input_length`.
- If you ever need to specify a fixed batch size for your inputs (this is useful for stateful recurrent networks), you can pass a `batch_size` argument to a layer. If you pass both `batch_size=32` and `input_shape=(6, 8)` to a layer, it will then expect every batch of inputs to have the batch shape `(32, 6, 8)`.

As such, the following snippets are strictly equivalent:

```
model = Sequential()
model.add(Dense(32, input_shape=(784,)))
```

```
model = Sequential()
model.add(Dense(32, input_dim=784))
```

## Compilation

Before training a model, you need to configure the learning process, which is done via the `compile` method. It receives three arguments:

- An optimizer. This could be the string identifier of an existing optimizer (such as `rmsprop` or `adagrad`), or an instance of the `Optimizer` class. See: optimizers.
- A loss function. This is the objective that the model will try to minimize. It can be the string identifier of an existing loss function (such as `categorical_crossentropy` or `mse`), or it can be an objective function. See: losses.
- A list of metrics. For any classification problem you will want to set this to `metrics=['accuracy']`. A metric could be the string identifier of an existing metric or a custom metric function.

```
# For a multi-class classification problem
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# For a binary classification problem
model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# For a mean squared error regression problem
model.compile(optimizer='rmsprop',
              loss='mse')

# For custom metrics
import keras.backend as K
```

```
def mean_pred(y_true, y_pred):
    return K.mean(y_pred)

model.compile(optimizer= 'rmsprop',
              loss= 'binary_crossentropy',
              metrics=['accuracy', mean_pred])
```

# Training

Keras models are trained on Numpy arrays of input data and labels. For training a model, you will typically use the `fit` function. Read its documentation here.

```
# For a single-input model with 2 classes (binary classification):

model = Sequential()
model.add(Dense( 32, activation='relu', input_dim=100))
model.add(Dense( 1, activation='sigmoid'))
model.compile(optimizer= 'rmsprop',
              loss= 'binary_crossentropy',
              metrics=[ 'accuracy'])

# Generate dummy data
import numpy as np
data = np.random.random(( 1000, 100))
labels = np.random.randint( 2, size=(1000, 1))
```

```
# Train the model, iterating on the data in batches of 32 samples
model.fit(data, labels, epochs=10, batch_size=32)
# For a single-input model with 10 classes (categorical classification):
model = Sequential()
model.add(Dense(32, activation='relu', input_dim=100))
model.add(Dense(10, activation='softmax'))
model.compile(optimizer='rmsprop',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
# Generate dummy data
import numpy as np
data = np.random.random((1000, 100))
labels = np.random.randint(10, size=(1000, 1))
# Convert labels to categorical one-hot encoding
one_hot_labels = keras.utils.to_categorical(labels, num_classes=10)
# Train the model, iterating on the data in batches of 32 samples
model.fit(data, one_hot_labels, epochs=10, batch_size=32)
```

# Examples

Here are a few examples to get you started!

In the examples folder, you will also find example models for real datasets:

- CIFAR10 small images classification: Convolutional Neural Network (CNN) with realtime data augmentation
- IMDB movie review sentiment classification: LSTM over sequences of words
- Reuters newswires topic classification: Multilayer Perceptron (MLP)
- MNIST handwritten digits classification: MLP & CNN
- Character-level text generation with LSTM

...and more.

## Multilayer Perceptron (MLP) for multi-class softmax classification:

```
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation
from keras.optimizers import SGD

# Generate dummy data
import numpy as np
x_train = np.random.random(( 1000, 20))
y_train = keras.utils.to_categorical(np.random.randint( 10, size=(1000, 1)),
num_classes=10)
x_test = np.random.random(( 100, 20))
y_test = keras.utils.to_categorical(np.random.randint( 10, size=(100, 1)),
num_classes=10)

model = Sequential()
# Dense(64) is a fully-connected layer with 64 hidden units.
# in the first layer, you must specify the expected input data shape:
```

```
# here, 20-dimensional vectors.
model.add(Dense(64, activation='relu', input_dim=20))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))

sgd = SGD(lr=0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

## MLP for binary classification:

```
import numpy as np
from keras.models import Sequential
from keras.layers import Dense, Dropout

# Generate dummy data
x_train = np.random.random((1000, 20))
y_train = np.random.randint(2, size=(1000, 1))
x_test = np.random.random((100, 20))
```

```
y_test = np.random.randint(2, size=(100, 1))

model = Sequential()
model.add(Dense(64, input_dim=20, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train,
          epochs=20,
          batch_size=128)
score = model.evaluate(x_test, y_test, batch_size=128)
```

## VGG-like convnet:

```
import numpy as np
import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.optimizers import SGD
```

```
# Generate dummy data
x_train = np.random.random((100, 100, 100, 3))
y_train = keras.utils.to_categorical(np.random.randint(10, size=(100, 1)),
num_classes=10)
x_test = np.random.random((20, 100, 100, 3))
y_test = keras.utils.to_categorical(np.random.randint(10, size=(20, 1)),
num_classes=10)

model = Sequential()
# input: 100x100 images with 3 channels -> (100, 100, 3) tensors.
# this applies 32 convolution filters of size 3x3 each.
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(100, 100, 3)))
model.add(Conv2D(32, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(10, activation='softmax'))
```

```
sgd = SGD(lr= 0.01, decay=1e-6, momentum=0.9, nesterov=True)
model.compile(loss= 'categorical_crossentropy', optimizer=sgd)

model.fit(x_train, y_train, batch_size= 32, epochs=10)
score = model.evaluate(x_test, y_test, batch_size= 32)
```

## Sequence classification with LSTM:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import LSTM

max_features = 1024

model = Sequential()
model.add(Embedding(max_features, output_dim= 256))
model.add(LSTM( 128))
model.add(Dropout( 0.5))
model.add(Dense( 1, activation='sigmoid'))

model.compile(loss= 'binary_crossentropy',
              optimizer= 'rmsprop',
              metrics=[ 'accuracy'])

model.fit(x_train, y_train, batch_size= 16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size= 16)
```

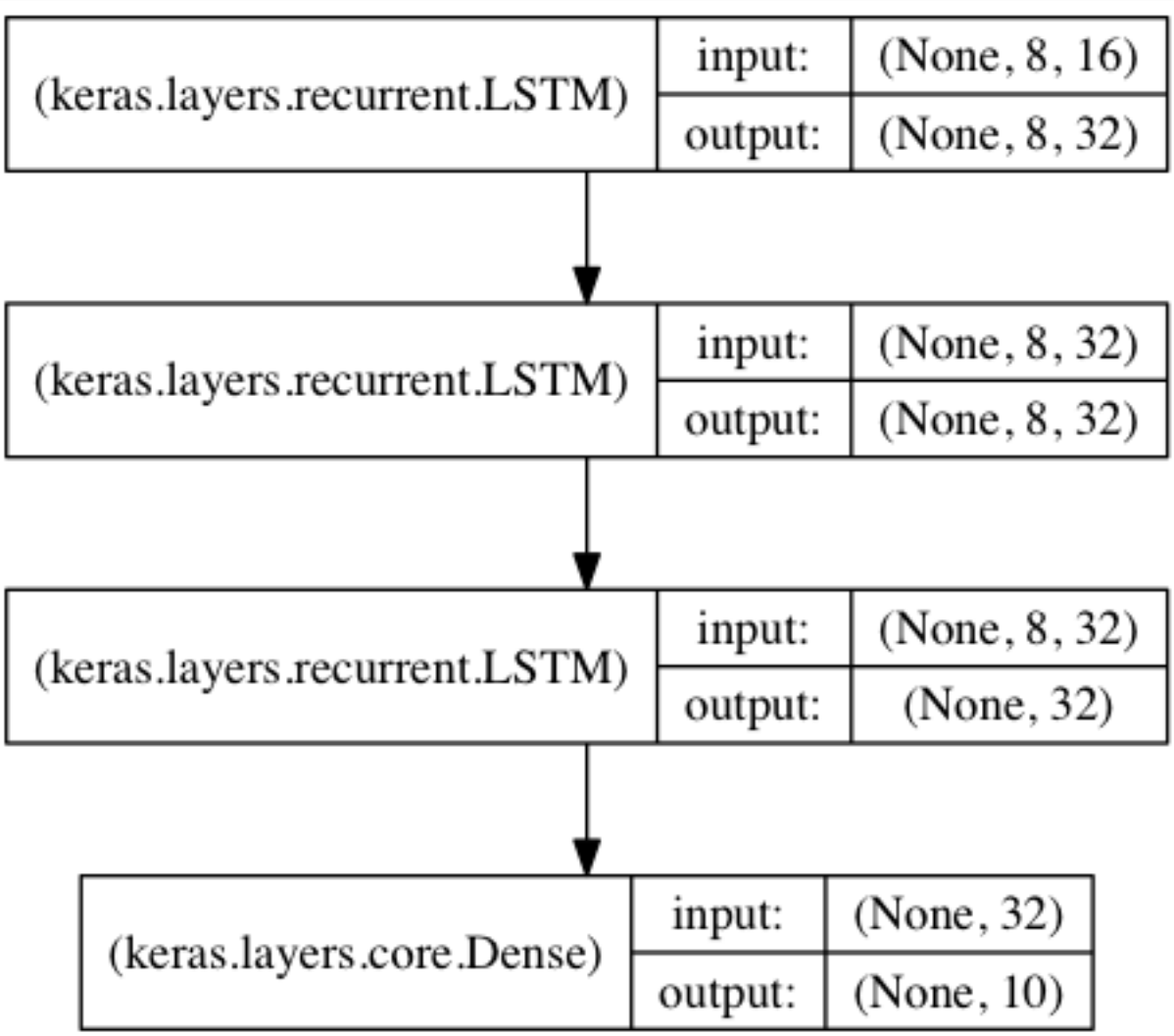
## Sequence classification with 1D convolutions:

```
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.layers import Embedding
from keras.layers import Conv1D, GlobalAveragePooling1D, MaxPooling1D

seq_length = 64

model = Sequential()
model.add(Conv1D(64, 3, activation='relu', input_shape=(seq_length, 100)))
model.add(Conv1D(64, 3, activation='relu'))
model.add(MaxPooling1D(3))
model.add(Conv1D(128, 3, activation='relu'))
model.add(Conv1D(128, 3, activation='relu'))
model.add(GlobalAveragePooling1D())
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))

model.compile(loss='binary_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])

model.fit(x_train, y_train, batch_size=16, epochs=10)
score = model.evaluate(x_test, y_test, batch_size=16)
```

## Stacked LSTM for sequence classification

In this model, we stack 3 LSTM layers on top of each other, making the model capable of learning higher-level temporal representations.

The first two LSTMs return their full output sequences, but the last one only returns the last step in its output sequence, thus dropping the temporal dimension (i.e. converting the input sequence into a single vector).




```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10
```

```
# expected input data shape: (batch_size, timesteps, data_dim)
model = Sequential()
model.add(LSTM(32, return_sequences=True,
               input_shape=(timesteps, data_dim)))  # returns a sequence of vectors of
dimension 32
model.add(LSTM(32, return_sequences=True))  # returns a sequence of vectors of
dimension 32
model.add(LSTM(32))  # return a single vector of dimension 32
model.add(Dense(10, activation='softmax'))

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

# Generate dummy training data
x_train = np.random.random((1000, timesteps, data_dim))
y_train = np.random.random((1000, num_classes))

# Generate dummy validation data
x_val = np.random.random((100, timesteps, data_dim))
y_val = np.random.random((100, num_classes))

model.fit(x_train, y_train,
          batch_size=64, epochs=5,
          validation_data=(x_val, y_val))
```

## Same stacked LSTM model, rendered "stateful"

A stateful recurrent model is one for which the internal states (memories) obtained after processing a batch of samples are reused as initial states for the samples of the next batch. This allows to process longer sequences while keeping computational complexity manageable.

You can read more about stateful RNNs in the FAQ.

```
from keras.models import Sequential
from keras.layers import LSTM, Dense
import numpy as np

data_dim = 16
timesteps = 8
num_classes = 10
batch_size = 32

# Expected input batch shape: (batch_size, timesteps, data_dim)
# Note that we have to provide the full batch_input_shape since the network is stateful.
# the sample of index i in batch k is the follow-up for the sample i in batch k-1.
model = Sequential()
model.add(LSTM(32, return_sequences=True, stateful=True,
               batch_input_shape=(batch_size, timesteps, data_dim)))
model.add(LSTM(32, return_sequences=True, stateful=True))
model.add(LSTM(32, stateful=True))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
```

```
              metrics=[ 'accuracy'])

# Generate dummy training data
x_train = np.random.random((batch_size * 10, timesteps, data_dim))
y_train = np.random.random((batch_size * 10, num_classes))

# Generate dummy validation data
x_val = np.random.random((batch_size * 3, timesteps, data_dim))
y_val = np.random.random((batch_size * 3, num_classes))

model.fit(x_train, y_train,
          batch_size=batch_size, epochs=5, shuffle=False,
          validation_data=(x_val, y_val))
```

Next  Previous

NOVEMBER 7, 2015 BY DENNY BRITZ

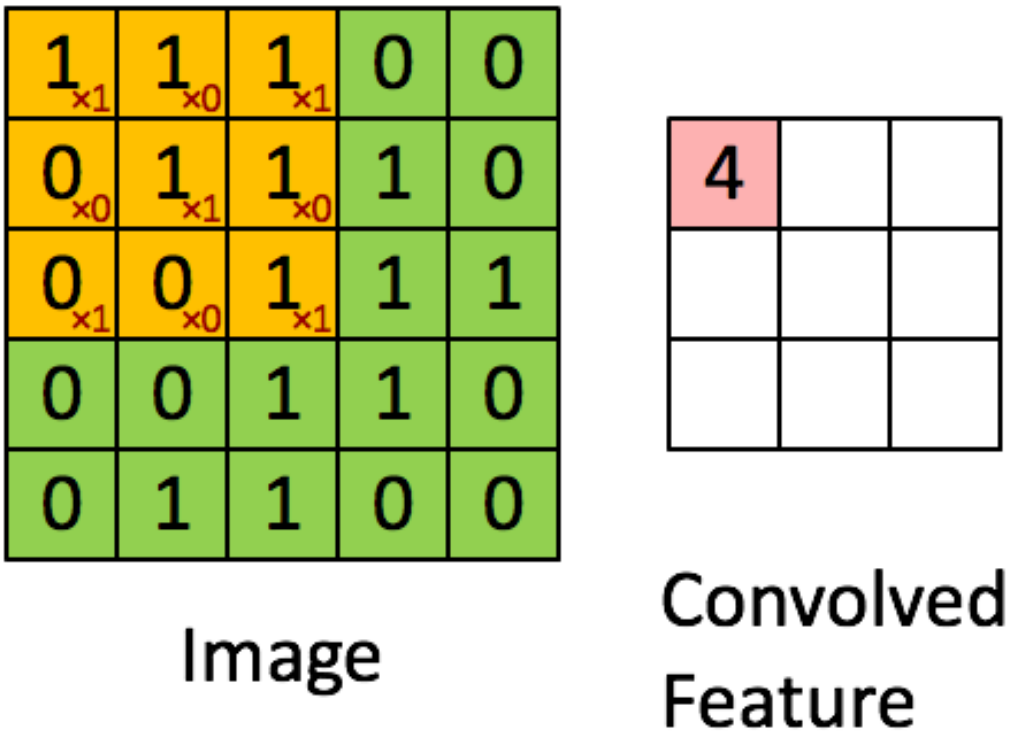
# Understanding Convolutional Neural Networks for NLP

When we hear about Convolutional Neural Network (CNNs), we typically think of Computer Vision. CNNs were responsible for major breakthroughs in Image Classification and are the core of most Computer Vision systems today, from Facebook's automated photo tagging to self-driving cars.

More recently we've also started to apply CNNs to problems in Natural Language Processing and gotten some interesting results. In this post I'll try to summarize what CNNs are, and how they're used in NLP. The intuitions behind CNNs are somewhat easier to understand for the Computer Vision use case, so I'll start there, and then slowly move towards NLP.

## What is Convolution?

The for me easiest way to understand a *convolution* is by thinking of it as a sliding window function applied to a matrix. That's a mouthful, but it becomes quite clear looking at a visualization:




Convolution with 3×3 Filter.
Source: http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution

Imagine that the matrix on the left represents an black and white image. Each entry corresponds to one pixel, 0 for black and 1 for white (typically it's between 0 and 255 for grayscale images). The sliding window is called a *kernel, filter,* or *feature detector.* Here we use a 3×3 filter, multiply its values element-wise with the original matrix, then sum them up. To get the full convolution we do this for each element by sliding the filter over the whole matrix.

You may be wondering wonder what you can actually do with this. Here are some intuitive examples.

*Averaging each pixel with its neighboring values blurs an image:*

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 1 | 1 | 1 | 0 |
| 0 | 0 | 0 | 0 | 0 |

*Taking the difference between a pixel and its neighbors detects edges:*

(To understand this one intuitively, think about what happens in parts of the image that are smooth, where a pixel color equals that of its neighbors: The additions cancel and the resulting value is 0, or black. If there’s a sharp edge in intensity, a transition from white to black for example, you get a large difference and a resulting white value)

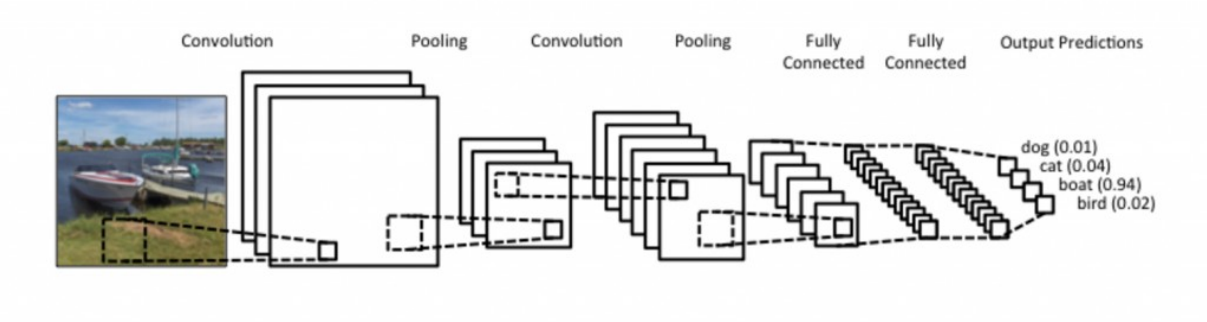
| | | | | |
|---|---|---|---|---|
| | 0 | 1 | 0 | |
| | 1 | -4 | 1 | |
| | 0 | 1 | 0 | |
| | | | | |

The GIMP manual has a few other examples. To understand more about how convolutions work I also recommend checking out Chris Olah's post on the topic.

## What are Convolutional Neural Networks?

Now you know what convolutions are. But what about CNNs? CNNs are basically just several layers of convolutions with *nonlinear activation functions* like ReLU or tanh applied to the results. In a traditional feedforward neural network we connect each input neuron to each output neuron in the next layer. That's also called a fully connected layer, or affine layer. In CNNs we don't do that. Instead, we use convolutions over the input layer to compute the output. This results in local connections, where each region of the input is connected to a neuron in the output. Each layer applies different filters, typically hundreds or thousands like the ones showed above, and combines their results. There's also something something called pooling (subsampling) layers, but I'll get into that later. During the training phase, **a CNN automatically learns the values of its filters** based on the task you want to perform. For example, in Image Classification a CNN may learn to detect edges from raw pixels in the first layer, then use the edges to detect simple shapes in the second layer, and then use these shapes to deter

higher-level features, such as facial shapes in higher layers. The last layer is then a classifier that uses these high-level features.




There are two aspects of this computation worth paying attention to: **Location Invariance** and **Compositionality**. Let's say you want to classify whether or not there's an elephant in an image. Because you are sliding your filters over the whole image you don't really care *where* the elephant occurs. In practice, *pooling* also gives you invariance to translation, rotation and scaling, but more on that later. The second key aspect is (local) compositionality. Each filter *composes* a local patch of lower-level features into higher-level representation. That's why CNNs are so powerful in Computer Vision. It makes intuitive sense that you build edges from pixels, shapes from edges, and more complex objects from shapes.

*So, how does any of this apply to NLP?*

Instead of image pixels, the input to most NLP tasks are sentences or documents represented as a matrix. Each row of the matrix corresponds to one token, typically a word, but it could be a character. That is, each row is vector that represents a word. Typically, these vectors are *word embeddings* (low-dimensional representations) like word2vec or GloVe, but they could also be one-hot vectors that index the word into a vocabulary. For a 10 word sentence using a 100-dimensional embedding we would have a 10×100 matrix as our input. That's our "image".

In vision, our filters slide over local patches of an image, but in NLP we typically use filters that slide over full rows of the matrix (words). Thus, the "width" of our filters is usually the same as the width of the input matrix. The height, or *region size*, may vary, but sliding windows over 2-5 words at a time is typical. Putting all the above together, a Convolutional Neural Network for NLP may look like this (take a few minutes and try understand this picture and how the dimensions are computed. You can ignore the pooling for now, we'll explain that later):

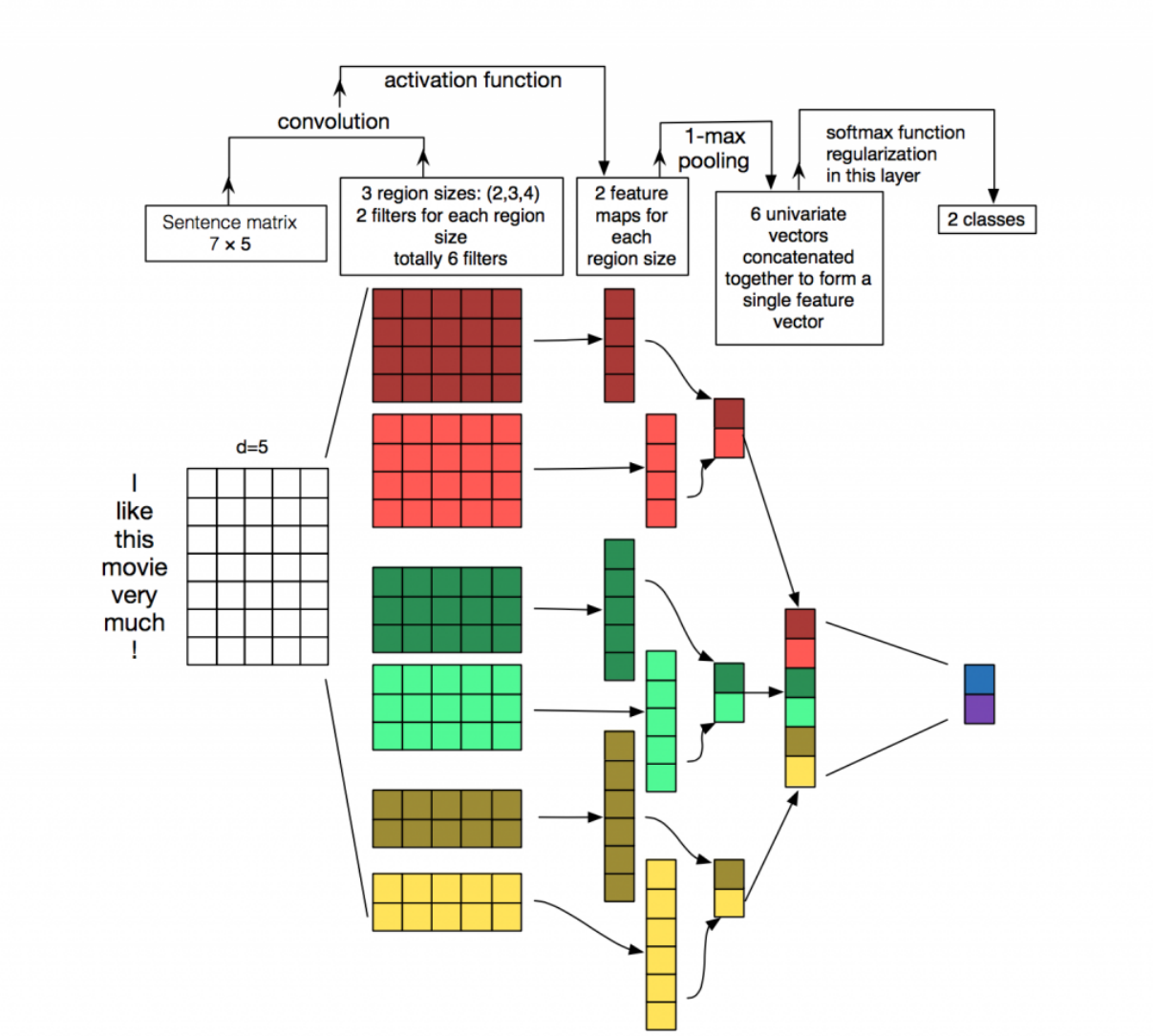



Illustration of a Convolutional Neural Network (CNN) architecture for sentence classification. Here we depict three filter region sizes: 2, 3 and 4, each of which has 2 filters. Every filter performs convolution on the sentence matrix and generates (variable-length) feature maps. Then 1-max pooling is performed over each map, i.e., the largest number from each feature map is recorded. Thus a univariate feature vector is generated from all six maps, and these 6 features are concatenated to form a feature vector for the penultimate layer. The final softmax layer then receives this feature vector as input and uses it to classify the sentence; here we assume binary classification and hence depict two possible output states. Source: Zhang, Y., & Wallace, B. (2015). A Sensitivity Analysis of (and Practitioners' Guide to) Convolutional Neural Networks for Sentence Classification.

What about the nice intuitions we had for Computer Vision? Location Invariance and local Compositionality made intuitive sense for images, but not so much for NLP. You probably do care a lot where in the sentence a word appears. Pixels close to each other are likely to be semantically related (part of the same object), but the same isn't always true for words. In many languages, parts of phrases could be separated by several other words. The compositional aspect isn't obvious either. Clearly, words compose in some ways, like an adjective modifying a noun, but how exactly this

works what higher level representations actually “mean” isn’t as obvious as in the Computer Vision case.

Given all this, it seems like CNNs wouldn’t be a good fit for NLP tasks. Recurrent Neural Networks make more intuitive sense. They resemble how we process language (or at least how we think we process language): Reading sequentially from left to right. Fortunately, this doesn’t mean that CNNs don’t work. All models are wrong, but some are useful. It turns out that CNNs applied to NLP problems perform quite well. The simple Bag of Words model is an obvious oversimplification with incorrect assumptions, but has nonetheless been the standard approach for years and lead to pretty good results.

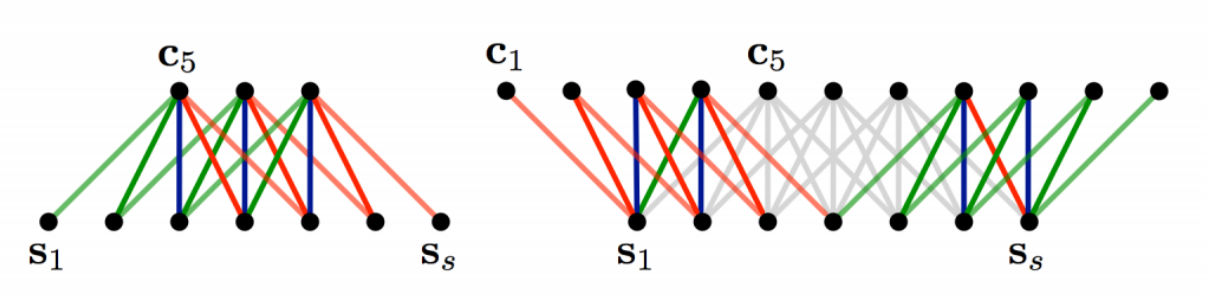
A big argument for CNNs is that they are fast. Very fast. Convolutions are a central part of computer graphics and implemented on a hardware level on GPUs. Compared to something like n-grams, CNNs are also *efficient* in terms of representation. With a large vocabulary, computing anything more than 3-grams can quickly become expensive. Even Google doesn’t provide anything beyond 5-grams. Convolutional Filters learn good representations automatically, without needing to represent the whole vocabulary. It’s completely reasonable to have filters of size larger than 5. I like to think that many of the learned filters in the first layer are capturing features quite similar (but not limited) to n-grams, but represent them in a more compact way.

## CNN Hyperparameters

Before explaining at how CNNs are applied to NLP tasks, let’s look at some of the choices you need to make when building a CNN. Hopefully this will help you better understand the literature in the field.

### *Narrow vs. Wide convolution*

When I explained convolutions above I neglected a little detail of how we apply the filter. Applying a 3×3 filter at the center of the matrix works fine, but what about the edges? How would you apply the filter to the first element of a matrix that doesn’t have any neighboring elements to the top and left? You can use *zero-padding*. All elements that would fall outside of the matrix are taken to be zero. By doing this you can apply the filter to every element of your input matrix, and get a larger or equally sized output. Adding zero-padding is also called *wide convolution*, and not using zero-padding would be a *narrow convolution*. An example in 1D looks like this:

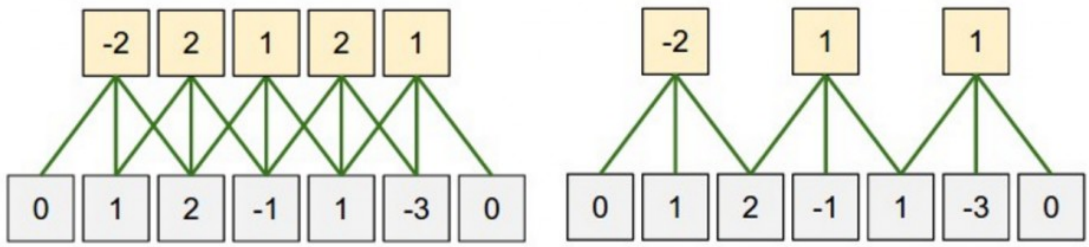



Narrow vs. Wide Convolution. Filter size 5, input size 7. Source: A Convolutional Neural Network for Modelling Sentences (2014)

You can see how wide convolution is useful, or even necessary, when you have a large filter relative to the input size. In the above, the narrow convolution yields an output of size $(7-5)+1=3$, and a wide convolution an output of size $(7+2*4-5)+1=11$. More generally, the formula for the output size is $n_{out}=(n_{in}+2*n_{padding}-n_{filter})+1$.

*Stride Size*

Another hyperparameter for your convolutions is the *stride size*, defining by how much you want to shift your filter at each step. In all the examples above the stride size was 1, and consecutive applications of the filter overlapped. A larger stride size leads to fewer applications of the filter and a smaller output size. The following from the Stanford cs231 website shows stride sizes of 1 and 2 applied to a one-dimensional input:



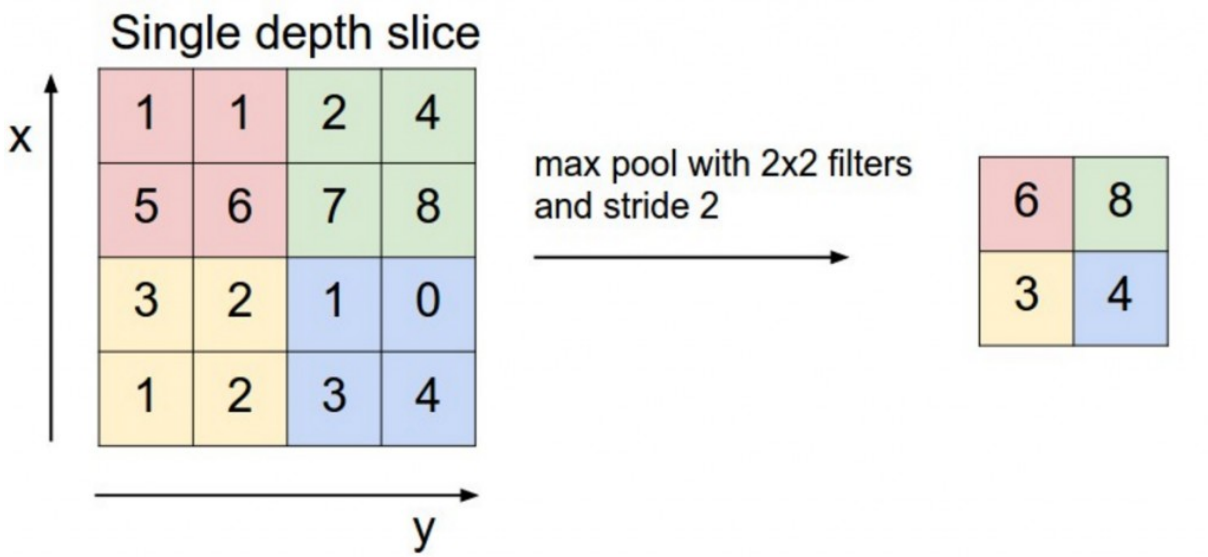

Convolution Stride Size. Left: Stride size 1. Right: Stride size 2.
Source: http://cs231n.github.io/convolutional-networks/

In the literature we typically see stride sizes of 1, but a larger stride size may allow you to build a model that behaves somewhat similarly to a Recursive Neural Network, i.e. looks like a tree.

*Pooling Layers*

A key aspect of Convolutional Neural Networks are *pooling layers,* typically applied after the convolutional layers. Pooling layers subsample their input. The most common way to do pooling it to

apply a $max$ operation to the result of each filter. You don't necessarily need to pool over the complete matrix, you could also pool over a window. For example, the following shows max pooling for a 2×2 window (in NLP we typically are apply pooling over the complete output, yielding just a single number for each filter):




Max pooling in CNN. Source: http://cs231n.github.io/convolutional-networks/#pool

Why pooling? There are a couple of reasons. One property of pooling is that it provides a fixed size output matrix, which typically is required for classification. For example, if you have 1,000 filters and you apply max pooling to each, you will get a 1000-dimensional output, regardless of the size of your filters, or the size of your input. This allows you to use variable size sentences, and variable size filters, but always get the same output dimensions to feed into a classifier.

Pooling also reduces the output dimensionality but (hopefully) keeps the most salient information. You can think of each filter as detecting a specific feature, such as detecting if the sentence contains a negation like "not amazing" for example. If this phrase occurs somewhere in the sentence, the result of applying the filter to that region will yield a large value, but a small value in other regions. By performing the max operation you are keeping information about whether or not the feature appeared in the sentence, but you are losing information about where exactly it appeared. But isn't this information about locality really useful? Yes, it is and it's a bit similar to what a bag of n-grams model is doing. You are losing global information about locality (where in a sentence something happens), but you are keeping local information captured

by your filters, like “not amazing” being very different from “amazing not”.

In imagine recognition, pooling also provides basic invariance to translating (shifting) and rotation. When you are pooling over a region, the output will stay approximately the same even if you shift/rotate the image by a few pixels, because the max operations will pick out the same value regardless.

*Channels*

The last concept we need to understand are *channels*. Channels are different “views” of your input data. For example, in image recognition you typically have RGB (red, green, blue) channels. You can apply convolutions across channels, either with different or equal weights. In NLP you could imagine having various channels as well: You could have a separate channels for different word embeddings (word2vec and GloVe for example), or you could have a channel for the same sentence represented in different languages, or phrased in different ways.

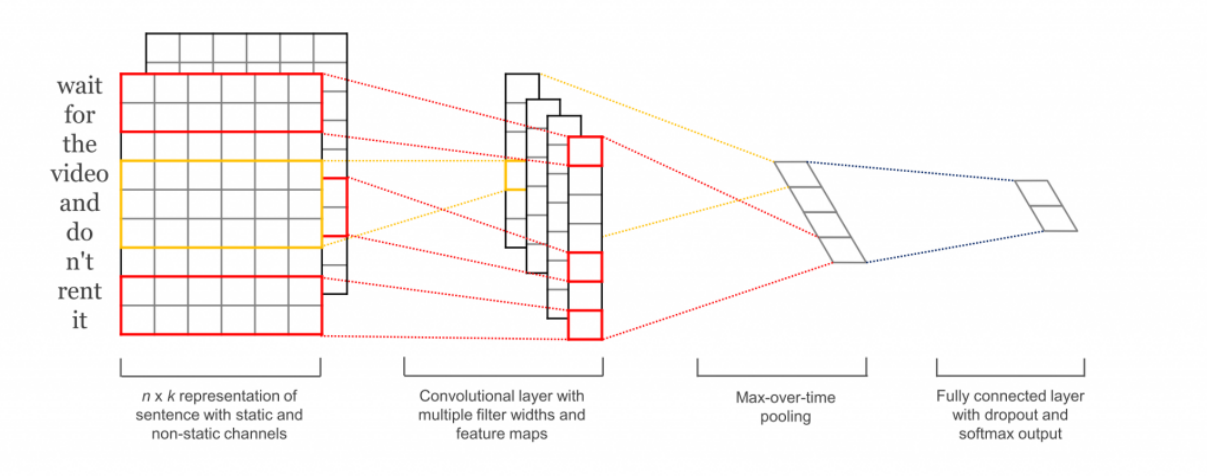
## Convolutional Neural Networks applied to NLP

Let’s now look at some of the applications of CNNs to Natural Language Processing. I’ll try it summarize some of the research results. Invariably I’ll miss many interesting applications (do let me know in the comments), but I hope to cover at least some of the more popular results.

The most natural fit for CNNs seem to be classifications tasks, such as Sentiment Analysis, Spam Detection or Topic Categorization. Convolutions and pooling operations lose information about the local order of words, so that sequence tagging as in PoS Tagging or Entity Extraction is a bit harder to fit into a pure CNN architecture (though not impossible, you can add positional features to the input).

[1] Evaluates a CNN architecture on various classification datasets, mostly comprised of Sentiment Analysis and Topic Categorization tasks. The CNN architecture achieves very good performance across datasets, and new state-of-the-art on a few. Surprisingly, the network used in this paper is quite simple, and that’s what makes it powerful.The input layer is a sentence comprised of concatenated word2vec word embeddings. That’s followed by a convolutional layer with multiple filters, then a max-pooling layer, and finally a softmax classifier.  The paper also experiments with two different channels in the form of static and dynamic word

embeddings, where one channel is adjusted during training and the other isn't. A similar, but somewhat more complex, architecture was previously proposed in [2]. [6] Adds an additional layer that performs "semantic clustering" to this network architecture.




Kim, Y. (2014). Convolutional Neural Networks for Sentence Classification

[4] Trains a CNN from scratch, without the need for for pre-trained word vectors like word2vec or GloVe. It applies convolutions directly to one-hot vectors. The author also proposes a space-efficient bag-of-words-like representation for the input data, reducing the number of parameters the network needs to learn. In [5] the author extends the model with an additional unsupervised "region embedding" that is learned using a CNN predicting the context of text regions. The approach in these papers seems to work well for long-form texts (like movie reviews), but their performance on short texts (like tweets) isn't clear. Intuitively, it makes sense that using pre-trained word embeddings for short texts would yield larger gains than using them for long texts.

Building a CNN architecture means that there are many hyperparameters to choose from, some of which I presented above: Input represenations (word2vec, GloVe, one-hot), number and sizes of convolution filters, pooling strategies (max, average), and activation functions (ReLU, tanh). [7] performs an empirical evaluation on the effect of varying hyperparameters in CNN architectures, investigating their impact on performance and variance over multiple runs. If you are looking to implement your own CNN for text classification, using the results of this paper as a starting point would be an excellent idea. A few results that stand out are that max-pooling always beat average pooling, that the ideal filter sizes are important but task-dependent, and that regularization

doesn’t seem to make a big different in the NLP tasks that were considered. A caveat of this research is that all the datasets were quite similar in terms of their document length, so the same guidelines may not apply to data that looks considerably different.

[8] explores CNNs for  Relation Extraction and Relation Classification tasks. In addition to the word vectors, the authors use the relative positions of words to the entities of interest as an input to the convolutional layer. This models assumes that the positions of the entities are given, and that each example input contains one relation. [9] and [10] have explored similar models.

Another interesting use case of CNNs in NLP can be found in [11] and [12], coming out of Microsoft Research. These papers describe how to learn semantically meaningful representations of sentences that can be used for Information Retrieval. The example given in the papers includes recommending potentially interesting documents to users based on what they are currently reading. The sentence representations are trained based on search engine log data.

Most CNN architectures learn embeddings (low-dimensional representations) for words and sentences in one way or another as part of their training procedure. Not all papers though focus on this aspect of training or investigate how meaningful the learned embeddings are. [13] presents a CNN architecture to predict hashtags for Facebook posts, while at the same time generating meaningful embeddings for words and sentences. These learned embeddings are then successfully applied to another task – recommending potentially interesting documents to users, trained based on clickstream data.

*Character-Level CNNs*

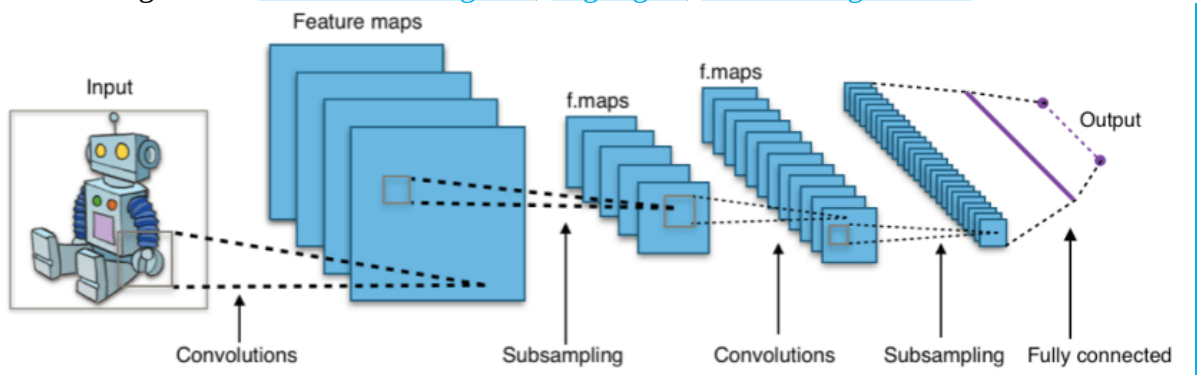
So far, all of the models presented were based on words. But there has also been research in applying CNNs directly to characters. [14] learns character-level embeddings, joins them with pre-trained word embeddings, and uses a CNN for Part of Speech tagging. [15][16] explores the use of CNNs to learn directly from characters, without the need for any pre-trained embeddings. Notably, the authors use a relatively deep network with a total of 9 layers, and apply it to Sentiment Analysis and Text Categorization tasks. Results show that learning directly from character-level input works very well on large datasets (millions of examples), but underperforms simpler models on smaller datasets (hundreds of thousands of examples). [17] explores to application of character-level convolutions to Language

Modeling, using the output of the character-level CNN as the input to an LSTM at each time step. The same model is applied to various languages.

What's amazing is that essentially all of the papers above were published in the past 1-2 years. Obviously there has been excellent work with CNNs on NLP before, as in Natural Language Processing (almost) from Scratch, but the pace of new results and state of the art systems being published is clearly accelerating.

# Explaining Tensorflow Code for a Convolutional Neural Network

Jessica Yung05.2017Artificial Intelligence, Highlights, Self-Driving Car ND




In this post, we will go through the code for a convolutional neural network. We will use Aymeric Damien's implementation. I recommend you have a skim before you read this post. I have included the key portions of the code below.

*If you're not familiar with TensorFlow or neural networks, you may find it useful to read my post on multilayer perceptrons (a simpler neural network) first.*

*Feature image credits: Aphex34 (Wikimedia Commons)*

## 1. Code

Here are the relevant network parameters and graph input for context (skim this, I'll explain it below). This network is applied to **MNIST data** – scans of handwritten digits from 0 to 9 we want to identify.

```
1  # Parameters
2  learning_rate = 0.001
3  training_iters = 200000
4  batch_size = 128
5  display_step = 10
```

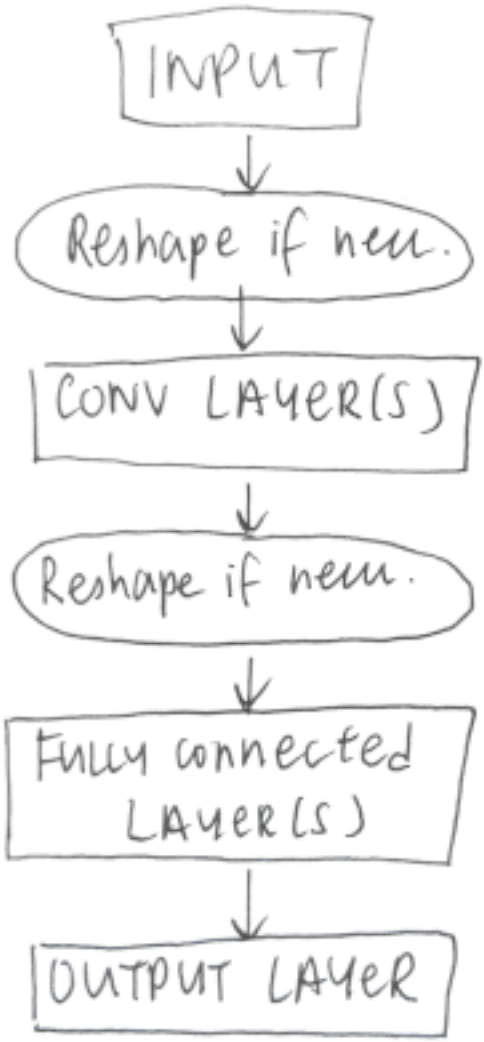
```
6
7 # Network Parameters
8 n_input = 784 # MNIST data input (img shape: 28*28)
9 n_classes = 10 # MNIST total classes (0-9 digits)
10 dropout = 0.75 # Dropout, probability to keep units
11
12 # tf Graph input
13 x = tf.placeholder(tf.float32, [None, n_input]) # input, i.e. pixels that constitute the image
14 y = tf.placeholder(tf.float32, [None, n_classes]) # labels, i.e which digit the image is
15 keep_prob = tf.placeholder(tf.float32) #dropout (keep probability)
```

Here is the model (I will explain this below):

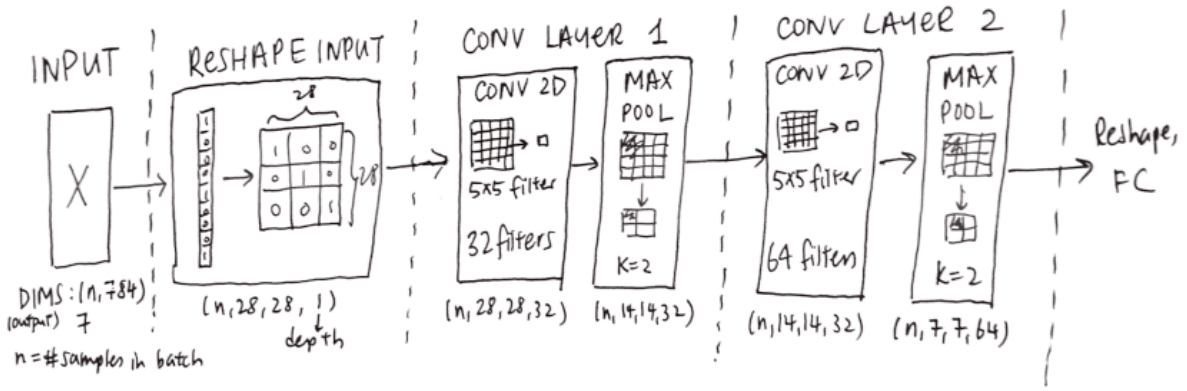
```
1 # Create model
2 def conv_net(x, weights, biases, dropout):
3 # Reshape input picture
4 x = tf.reshape(x, shape=[-1, 28, 28, 1])
5
6 # Convolution Layer
7 conv1 = conv2d(x, weights['wc1'], biases['bc1'])
8 # Max Pooling (down-sampling)
9 conv1 = maxpool2d(conv1, k=2)
10
11 # Convolution Layer
12 conv2 = conv2d(conv1, weights['wc2'], biases['bc2'])
13 # Max Pooling (down-sampling)
14 conv2 = maxpool2d(conv2, k=2)
15
16 # Reshape conv2 output to fit fully connected layer input
17 fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
18 # Fully connected layer
19 fc1 = tf.add(tf.matmul(fc1, weights['wd1']), biases['bd1'])
20 fc1 = tf.nn.relu(fc1)
21 # Apply Dropout
22 fc1 = tf.nn.dropout(fc1, dropout)
23
24 # Output, class prediction
25 out = tf.add(tf.matmul(fc1, weights['out']), biases['out'])
26 return out
27
28 # Store layers weight &amp;amp; bias
29 weights = {
30 # 5x5 conv, 1 input, 32 outputs
31 'wc1': tf.Variable(tf.random_normal([5, 5, 1, 32])),
32 # 5x5 conv, 32 inputs, 64 outputs
33 'wc2': tf.Variable(tf.random_normal([5, 5, 32, 64])),
34 # fully connected, 7*7*64 inputs, 1024 outputs
35 'wd1': tf.Variable(tf.random_normal([7*7*64, 1024])),
36 # 1024 inputs, 10 outputs (class prediction)
37 'out': tf.Variable(tf.random_normal([1024, n_classes]))
38 }
39
40 biases = {
41 'bc1': tf.Variable(tf.random_normal([32])),
42 'bc2': tf.Variable(tf.random_normal([64])),
43 'bd1': tf.Variable(tf.random_normal([1024])),
44 'out': tf.Variable(tf.random_normal([n_classes]))
45 }
46
47 # Construct model
48 pred = conv_net(x, weights, biases, keep_prob)
```

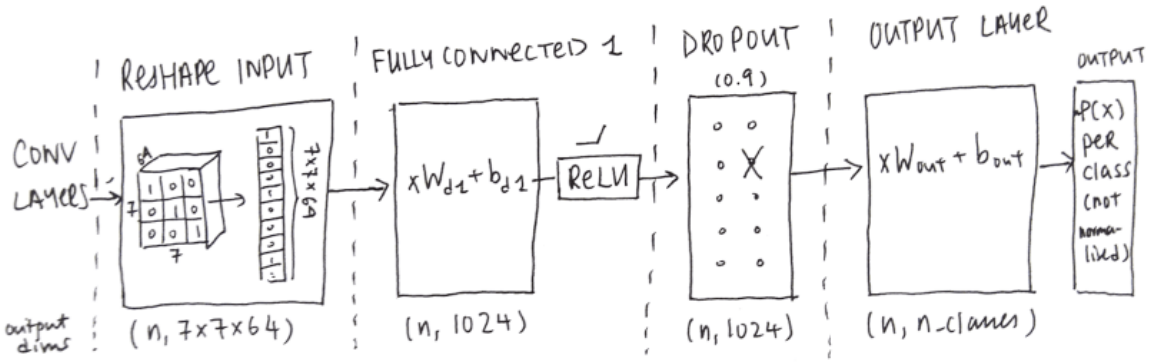
# *2. Translating the code*

Let’s draw the model the function conv_net represents. The **batch size** given is 128. That means that each time, at most 128 images are fed into our model.
The big picture:




In more detail:



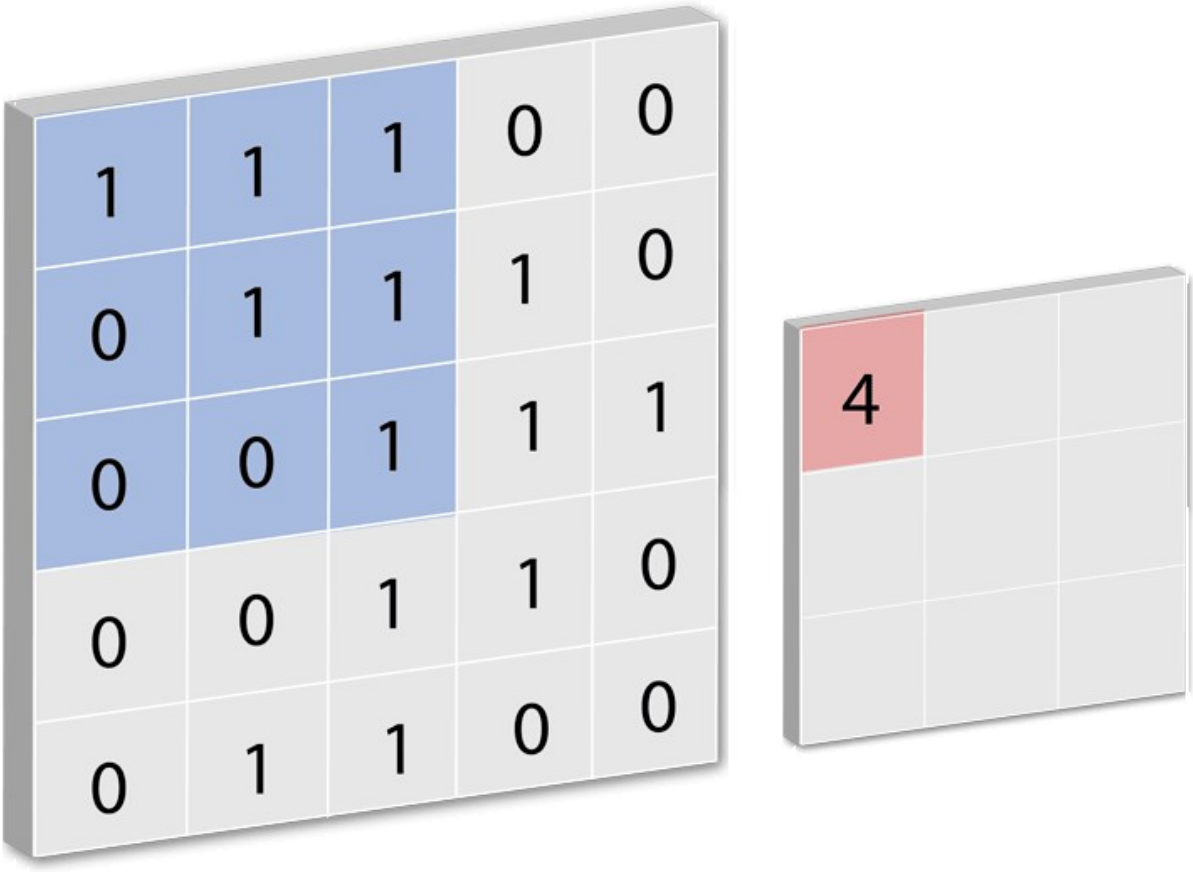




We can see that there are fives types of layers here:

- convolution layers,
- max pooling layers,
- layers for reshaping input,
- fully-connected layers and
- dropout layers.

### 2.1 What is conv2d (convolution layer)?

A convolution layer tries to extract higher-level features by replacing data for each (one) pixel with a value computed from the pixels covered by the e.g. 5×5 filter centered on that pixel(all the pixels in that region).



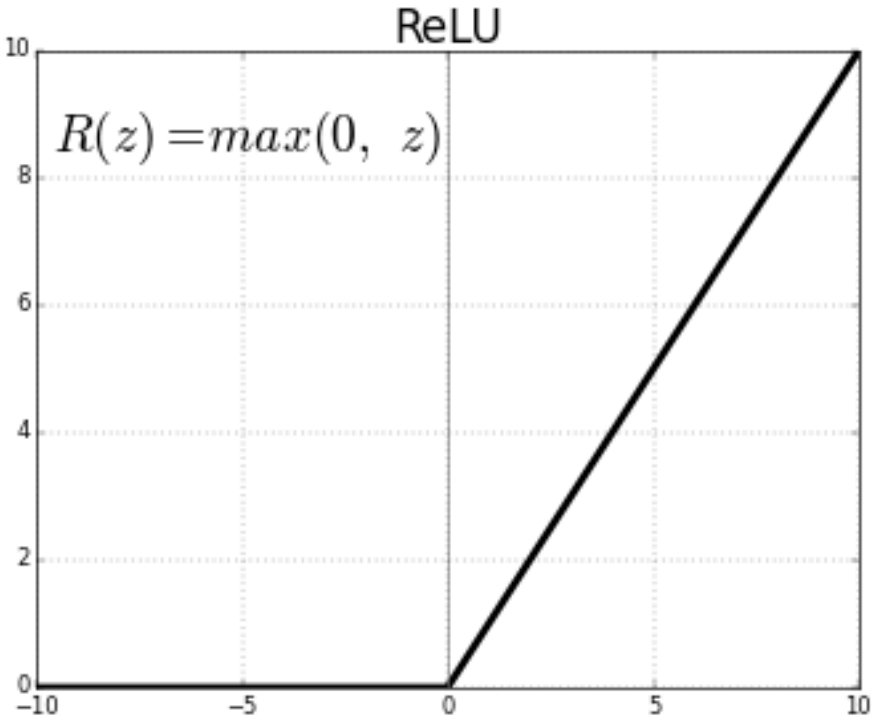

CREDITS: FLETCHER BACH

We slide the filter across the width and height of the input and compute the dot products between the entries of the filter and input at each position. I explain this further when discussing tf.nn.conv2d() below. Stanford's CS231n course provides an excellent explanation of how convolution layers work (complete with diagrams). Here we will focus on the code.

```
1 def conv2d(x, W, b, strides=1):
2 # Conv2D wrapper, with bias and relu activation
3 x = tf.nn.conv2d(x, W, strides=[1, strides, strides, 1], padding='SAME')
4 x = tf.nn.bias_add(x, b)
5 return tf.nn.relu(x)
```

This function comprises three parts:

1. Conv2D layer from Tensorflow
    1. tf.nn.conv2d()
    2. This is analogous to xW (multiplying input by weights) in a fully connected layer.
2. Add bias
3. ReLU activation
    1. This transforms the output like so: $ReLU(x) = max(0, x)$. (See previous post for more details).

2. 

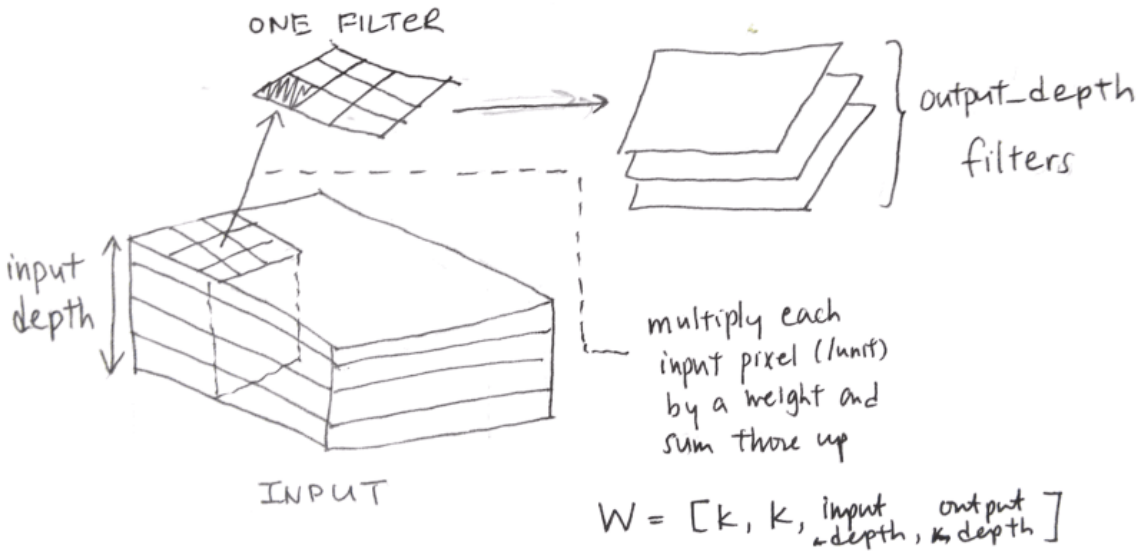

X AXIS: INPUT, Y AXIS: OUTPUT. CREDITS: ML4A ON GITHUB

You can see it is structurally the same as a fully connected layer, except we multiply the input with weights in a different way.

**Conv2D layer**

The key part here is tf.nn.conv2d(). Let's look at each of its arguments.

- x is the input.
- W are the weights.
  - The weights have four dimensions: [filter_height, filter_width, input_depth,output_depth].
  - What this means is that **we have** output_depth **filters in this layer**.
    - Each filter considers information with dimensions [filter_height, filter_width,input_depth] at a time. Yes, **each filter goes through ALL the input depth layers**.




- This is like how, in a fully connected layer, we may have ten neurons, each of which interacts with all the neurons in the previous layer.

- **stride** is the number of units the filter shifts each time.
  - Why are there four dimensions? This is because the input tensor has four dimensions: [number_of_samples, height, width, colour_channels].
  - strides = [1, strides, strides, 1] thus applies the filter to every image and every colour channel and to every strides image patch in the height and width dimensions.
  - You don't usually skip entire images or entire colour channels, so those positions are hardcoded as 1 here.
  - E.g. strides=[1, 2, 2, 1] would apply the filter to every other image patch in each dimension. (Image below has width stride 1.)
- "SAME" **padding**: the output size is the same as the input size. This requires the filter window to shift out of the input map. The portions where the filter window is outside of the input map is the padding.

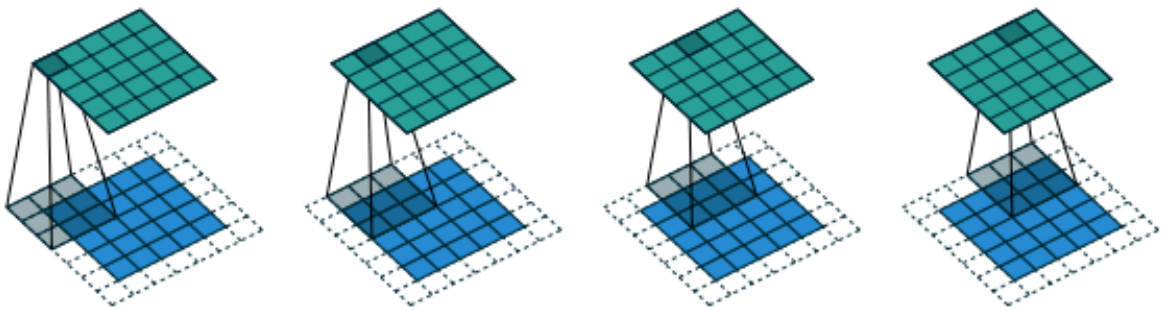
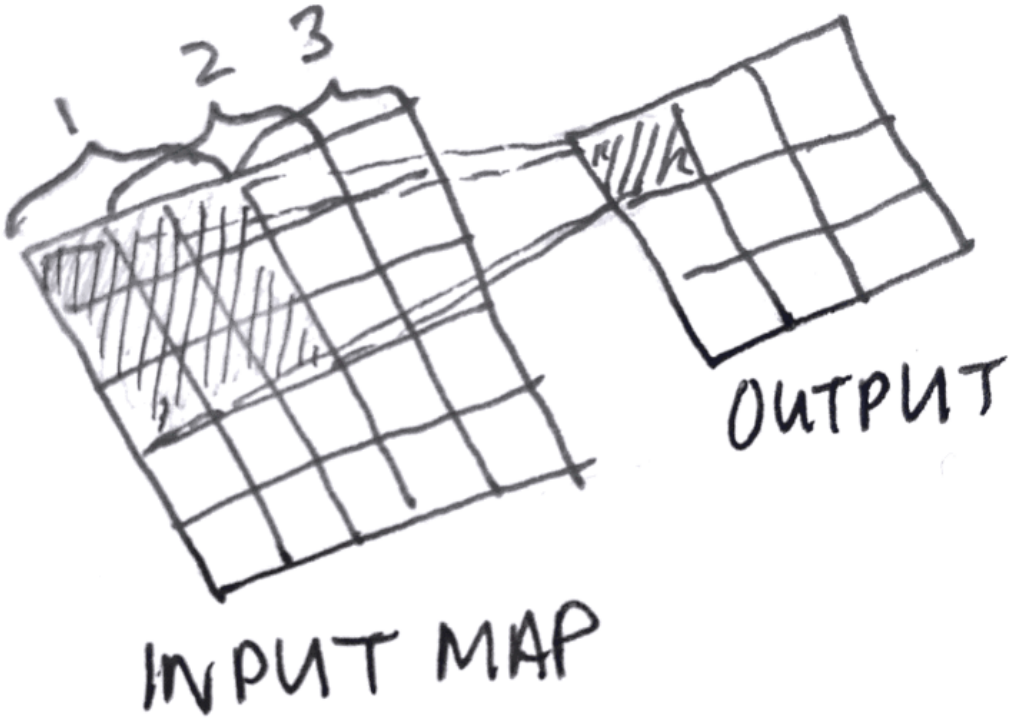


Figure 2.3: (Half padding, no strides) Convolving a $3 \times 3$ kernel over a $5 \times 5$ input using half padding and unit strides (i.e., $i = 5$, $k = 3$, $s = 1$ and $p = 1$).

"SAME" PADDING: OUTPUT (GREEN) IS THE SAME SIZE AS THE INPUT (BLUE). STRIDE = 1.

- The alternative is "VALID" padding, where there is no padding. The filter window stays inside the input map the whole time (in *valid positions*), so the output size is smaller than the input.



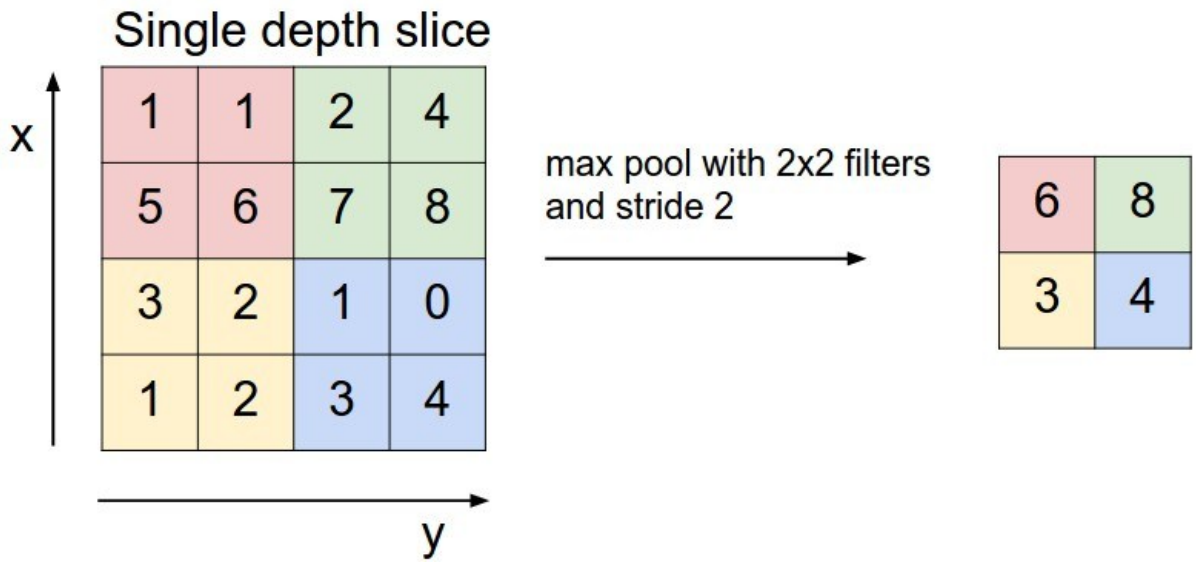

‘VALID’ PADDING.

# *2.2 What is maxpool2d (max_pool)?*

Pooling layers reduce the spatial size of the output by replacing values in the kernel by a function of those values. E.g. in **max** pooling, you take the maximum out of every pool (kernel) as the new value for that pool.

```
1 def maxpool2d(x, k=2):
2 # MaxPool2D wrapper
3 return tf.nn.max_pool(x, ksize=[1, k, k, 1], strides=[1, k, k, 1],
4 padding='SAME')
```

Here the kernel is square and the kernel size is set to be the same as the stride. It resizes the input as shown in the diagram below:




MAX POOLING WITH K = 2. CREDITS: CS231N

## *2.3 Layers for reshaping input*

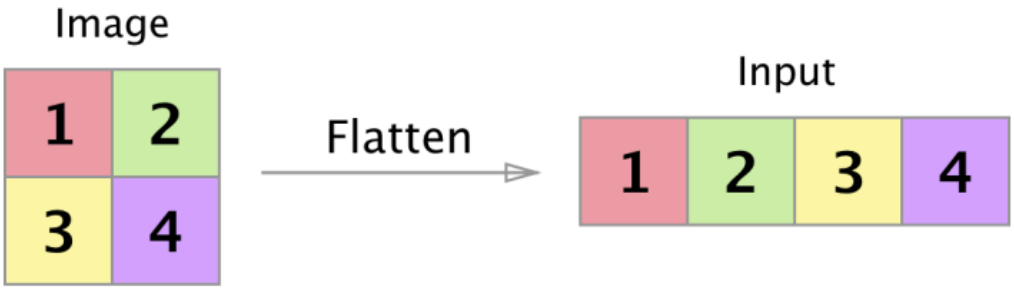
We reshape input twice in this model. The first time is at the beginning:

```
1 x = tf.reshape(x, shape=[-1, 28, 28, 1])
```

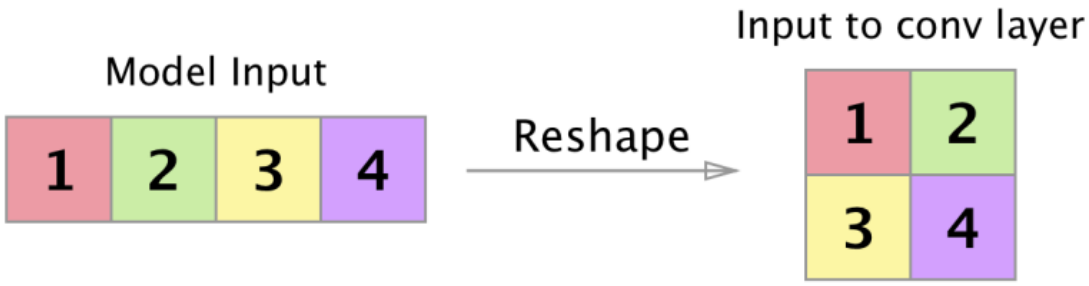
Recall the input was

```
1 x = tf.placeholder(tf.float32, [None, n_input]) # input, i.e. pixels that constitute the image
```

That is, each sample inputted to the model was a one-dimensional array: an image flattened into a list of pixels. That is, the person who was preprocessing the MNIST dataset did this with each image:



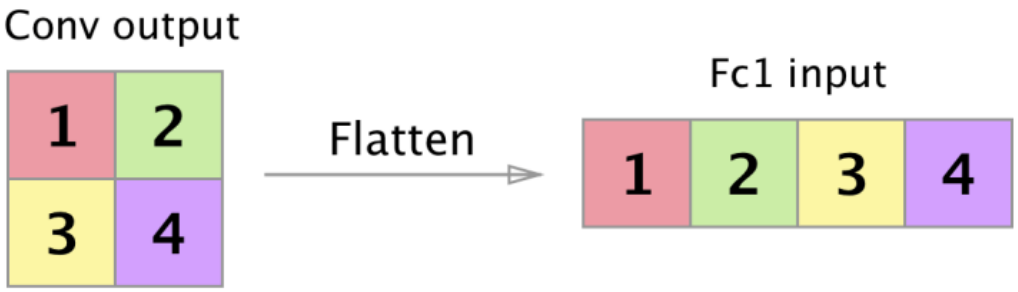

And now we're reversing the process:



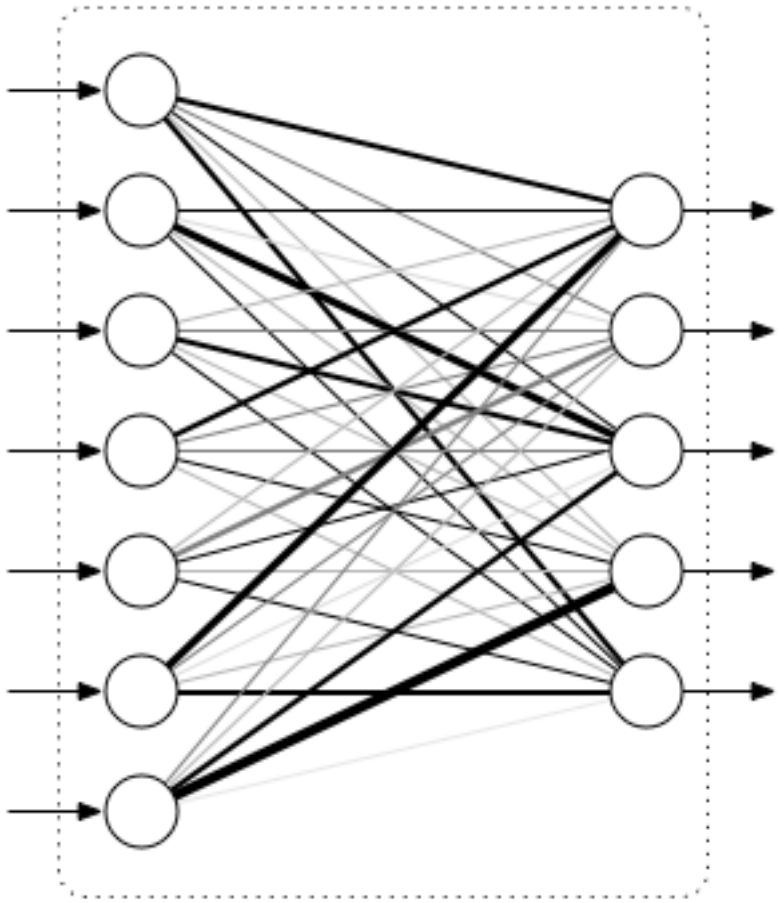

The second time we reshape input is right after the convolutional layer before the first fully connected layer:

```
# Reshape conv2 output to fit fully connected layer input

# latter part gets the first dimension of the shape of weights['wd1'], i.e. the number of rows it has.
fc1 = tf.reshape(conv2, [-1, weights['wd1'].get_shape().as_list()[0]])
```

We're doing this again to prepare 1D input for the fully connected layer:



## 2.4 What is a fully connected layer?



Do a Wx + b: For each neuron (number of neurons = number of outputs), multiply each input by a weight, sum all those products up and then add a bias to get your output for that neuron.

See post Explaining TensorFlow code for a Multilayer Perceptron.

## 2.5 What is Dropout?

Just before the output layer, we apply Dropout:

```
1 fc1 = tf.nn.dropout(fc1, dropout)
```

Dropout sets a proportion 1-dropout of activations (neuron outputs) passed on to the next layer to zero. The zeroed-out outputs are chosen randomly.

- What happens if we set the dropout parameter to 0?

This reduces overfitting by checking that the network can provide the right output even if some activations are dropped out.

And that’s a wrap – hope you found this useful! If you enjoyed this or have any suggestions, do leave a comment.

# TensorFlow: Shapes and dynamic dimensions

Here is a simple HowTo to understand the concept of shapes in TensorFlow and hopefully avoid losing hours of debugging them.

## *What is a tensor?*

Very briefly, a tensor is an N-dimensional array containing the same type of data (int32, bool, etc.): All you need to describe a tensor fully is its data type and the value of each of the N dimension.

That’s why we describe a tensor with what we call a shape: it is a list, tuple or TensorShape of numbers containing the size of each dimension of our tensor, for example:

- For a tensor of n dimensions: **(D0, D1, …, Dn-1)**
- For a tensor of size **W x H** (usually called a matrix): **(W, H)**
- For a tensor of size **W** (usually called a vector): **(W,)**
- For a simple scalar (those are equivalent): **() or (1,)**

*Note: (**D***, **W** and **H** are integers)*

*Note on the vector (1-D tensor): it is impossible to determine if a vector is a row or column vector by looking at the vector shape in TensorFlow, and in fact, it doesn't matter. For more information please look at this stack overflow answer about NumPy notation ( which is roughly the same as TensorFlow notation):* http://stackoverflow.com/questions/22053050/difference-between-numpy-array-shape-r-1-and-r

A tensor looks like this in TensorFlow:

We can see we have a **Tensor** object:

- It has a **name** used in a key-value store to retrieve it later: **Const:0**
- It has a **shape** describing the size of each dimension: **(6, 3, 7)**
- It has a **type**: **float32**

That's it!

Now, here is the most important piece of this article: **Tensors in TensorFlow have 2 shapes: The static shape AND the dynamic shape!**

*Tensor in TensorFlow has 2 shapes! The static shape AND the dynamic shape*

# The static shape

The static shape is the shape you provided when creating a tensor OR the shape inferred by TensorFlow

when you define an operation resulting in a new tensor. It is a tuple or a list.

TensorFlow will do its best to guess the shape of your different tensors (between your different operations) but it won't always be able to do it. Especially if you start to do operations with placeholder defined with unknown dimensions (like when you want to use a dynamic batch size).

*The static shape is a tuple or a list.*

To use the static shape (Accessing/changing) in your code, you will use the different functions which are **attached to the Tensor itself and have an underscore in their names:**

*Note: The static shape is very useful to debug your code with* `print` *so you can check your tensors have the right shapes.*

# The dynamic shape

The dynamic shape is the actual one used when you `run` your graph. It is itself a tensor describing the shape of the original tensor.

If you defined a placeholder with undefined dimensions (with the `None` type as a dimension),
those `None` dimensions will only have a real value when you feed an input to your placeholder and so forth, any variable depending on this placeholder.

*The dynamic shape is itself a tensor describing the shape of the original tensor*

To use the dynamic shape(Accessing/changing) in your code, you will use the different **functions which are attached to the main scope and don’t have an underscore in their names**:

*The dynamic shape is very handy for dealing with dimensions that you want to keep dynamic.*

# A real use case: the RNN

We like dynamic inputs because we want to build a dynamic RNN which should be able to handle any different length of inputs.

In the training phase we will define a placeholder with a dynamic batch_size, and then we will use the TensorFlow API to create an LSTM. You will end up with something like this:
And now you need to initialize the init_state with `init_state = cell.zero_state(batch_size, tf.float32)` **…**

But what the “batch_size” input should be equal to? Remember, you want it to be dynamic so what are our options? TensorFlow allows different types here, if you read the source code you will find:

```
Args:
    batch_size: int, float, or unit Tensor representing the batch size.
```

**int** and **float** can’t be used because when you define your graph, you actually don’t know what the **batch_size** will be (that’s the point).

The interesting piece is the last type: **“unit Tensor representing the batch size”.** If you dig the doc up from there, you will find that a unit Tensor is a “0-d Tensor” which is just a scalar. So how do you get that scalar-tensor anyway?

If you try with the **static shape**:
`batch_size` will be the `Dimension(None)` type (printed as ‘?’). This type can only be used as a dimension for placeholders. What you actually want to do is to keep the dynamic `batch_size` “flow” through the graph, so you must use the **dynamic shape**:
`batch_size` will be a TensorFlow `0-d Tensor (Scalar Tensor)` type describing the batch dimension, hooray!

# Conclusion

- Use the static shape for debugging
- Use the dynamic shape everywhere else especially when you have undefined dimensions

## Convolutional neural networks for language tasks

Though they are typically applied to vision problems, convolution neural networks can be very effective for some language tasks.

When approaching problems with sequential data, such as natural language tasks, recurrent neural networks (RNNs) typically top the choices. While the temporal nature of RNNs are a natural fit for these problems with text data, convolutional neural networks (CNNs), which are tremendously successful when applied to vision tasks, have also demonstrated efficacy in this space.

In our LSTM tutorial, we took an in-depth look at how long short-term memory (LSTM) networks work and used TensorFlow to build a multi-layered LSTM network to model stock market sentiment from social media content. In this post, we will briefly discuss how CNNs are applied to text data while providing some sample TensorFlow code to build a CNN that can perform binary classification tasks similar to our stock market sentiment model.

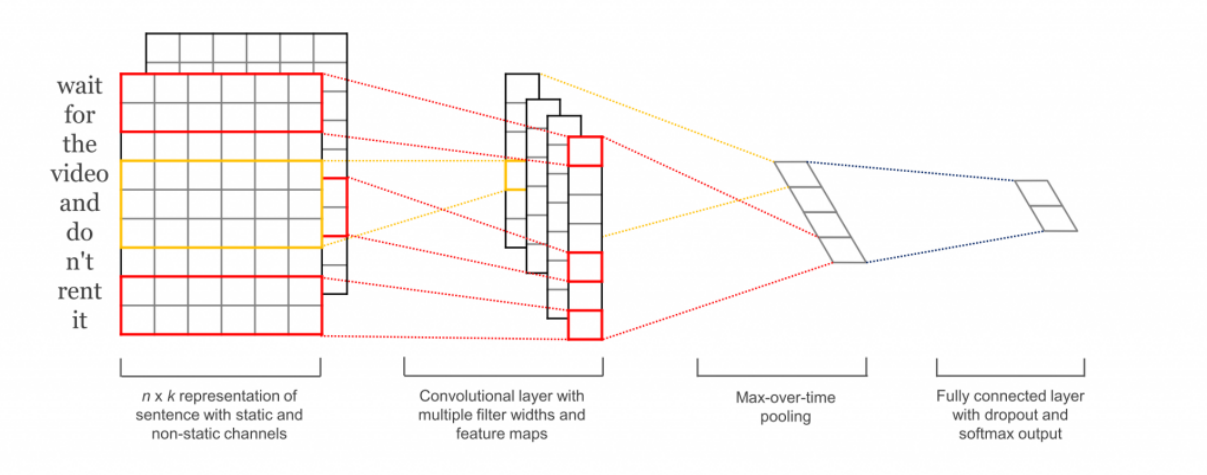



Figure 1. Sample CNN model architecture for text classification. Image by Garrett Hoffman, based on a figure from "Convolutional Neural Networks for Sentence Classification."

We see a sample CNN architecture for text classification in Figure 1. First, we start with our input sentence (of length *seq_len*), represented as a matrix in which the rows are our words vectors and the columns are the dimensions of the distributed word embedding. In computer vision problems, we typically see three input channels for RGB; however, for text we have only a single input channel. When we implement our model in TensorFlow, we first define placeholders for our inputs and then build the embedding matrix and embedding lookup.

```
# Define Inputs
inputs_ = tf.placeholder(tf.int32, [None, seq_len], name='inputs')
labels_ = tf.placeholder(tf.float32, [None, 1], name='labels—)
training_ = tf.placeholder(tf.bool, name='training')

# Define Embeddings
embedding = tf.Variable(tf.random_uniform((vocab_size, embed_size), -1, 1))
embed = tf.nn.embedding_lookup(embedding, inputs_)
```

Notice how the CNN processes the input as a complete sentence, rather than word by word as we did with the LSTM. For our CNN, we pass a tensor with all word indices in our sentence to our embedding lookup and get back the matrix for our sentence that will be used as the input to our network.

Now that we have our embedded representation of our input sentence, we build our convolutional layers. In our CNN, we will use one-dimensional convolutions, as opposed to the two-dimensional convolutions typically used on vision tasks. Instead of defining a height and a width for our filters, we will only define a height, and the width will always be the embedding dimension. This makes sense intuitively, when compared to how images are represented in CNNs. When we deal with images, each pixel is a unit for analysis, and these pixels exist in both dimensions of our input image. For our sentence, each word is a unit for analysis and is represented by the dimension of our embeddings (the width of our input matrix), so words exist only in the single dimension of our rows.

We can include as many one-dimensional kernels as we like with different sizes. Figure 1 shows a kernel size of two (red box over input) and a kernel size of three (yellow box over input). We also define a uniform number of filters (in the same fashion as we would for a two-dimensional convolutional layer) for each of our layers, which will be the output dimension of our convolution. We apply a relu activation and add a max-over-time pooling to our output that takes the maximum output for each filter of each convolution—resulting in the extraction of a single model feature from each filter.

```
# Define Convolutional Layers with Max Pooling
convs = []
for filter_size in filter_sizes:
   conv = tf.layers.conv1d(inputs=embed, filters=128, kernel_size=filter_size,
activation=tf.nn.relu)
   pool = tf.layers.max_pooling1d(inputs=conv, pool_size=seq_len-filter_size+1,
strides=1)
   convs.append(pool)
```

We can think of these layers as “parallel”—i.e., one convolution layer doesn’t feed into the next, but rather they are all functions on the input that result in a unique output. We concatenate and flatten these outputs to combine the results.

```
# Concat Pooling Outputs and Flatten
pool_concat = tf.concat(convs, axis=-1)
pool_flat = tf.layers.Flatten(pool_concat)
```

Finally, we now build a single fully connected layer with a sigmoid activation to make predictions from our concatenated convolutional outputs. Note that we can use a `tf.nn.softmax` activation function here as well if the problem has more than two classes. We also include a dropout layer here to regularize our model for better out-of-sample performance.

```
dense = tf.layers.Dense(inputs=drop, num_outputs=1,
activation_fn=tf.nn.sigmoid)
```

Finally, we can wrap this code into a custom `tf.Estimator` using the `model_fn` for a simple API for training, evaluating and making future predictions.

And there we have it: a convolutional neural network architecture for text classification.

As with any model comparison, there are some trade offs between CNNs and RNNs for text classification. Even though RNNs seem like a more natural choice for language, CNNs have been shown to train up to 5x faster than RNNs and perform well on text where feature detection is important. However, when long-term dependency over the input sequence is an important factor, RNN variants typically outperform CNNs.

Ultimately, language problems in various domains behave differently, so it is important to have multiple techniques in your arsenal. This is just one example of a trend we are seeing in applying techniques successfully across different areas of research. While convolutional neural networks have traditionally been the star of the computer vision world, we are starting to see more breakthroughs in applying them to sequential data.

# Text Classification with TensorFlow Estimators

- Loading data using Datasets.
- Building baselines using pre-canned estimators.
- Using word embeddings.

- Building custom estimators with convolution and LSTM layers.
- Loading pre-trained word vectors.
- Evaluating and comparing models using TensorBoard.

---

Welcome to Part 4 of a blog series that introduces TensorFlow Datasets and Estimators. You don't need to read all of the previous material, but take a look if you want to refresh any of the following concepts. Part 1 focused on pre-made Estimators, Part 2 discussed feature columns, and Part 3 how to create custom Estimators.
Here in Part 4, we will build on top of all the above to tackle a different family of problems in Natural Language Processing (NLP). In particular, this article demonstrates how to solve a text classification task using custom TensorFlow estimators, embeddings, and the tf.layers module. Along the way, we'll learn about word2vec and transfer learning as a technique to bootstrap model performance when labeled data is a scarce resource.
We will show you relevant code snippets. Here's the complete Jupyter Notebook that you can run locally or on Google Colaboratory. The plain .py source file is also available here. Note that the code was written to demonstrate how Estimators work functionally and was not optimized for maximum performance.

**The task**

The dataset we will be using is the IMDB Large Movie Review Dataset, which consists of 25,00025,00025,000 highly polar movie reviews for training, and 25,00025,00025,000 for testing. We will use this dataset to train a binary

classification model, able to predict whether a review is positive or negative.
For illustration, here's a piece of a negative review (with 222 stars) in the dataset:
*Now, I LOVE Italian horror films. The cheesier they are, the better. However, this is not cheesy Italian. This is week-old spaghetti sauce with rotting meatballs. It is amateur hour on every level. There is no suspense, no horror, with just a few drops of blood scattered around to remind you that you are in fact watching a horror film.*
*Keras* provides a convenient handler for importing the dataset which is also available as a serialized numpy array .npz file to download here. For text classification, it is standard to limit the size of the vocabulary to prevent the dataset from becoming too sparse and high dimensional, causing potential overfitting. For this reason, each review consists of a series of word indexes that go from 444 (the most frequent word in the dataset **the**) to 499949994999, which corresponds to **orange**. Index 111 represents the beginning of the sentence and the index 222 is assigned to all unknown (also known as *out-of-vocabulary* or *OOV*) tokens. These indexes have been obtained by pre-processing the text data in a pipeline that cleans, normalizes and tokenizes each sentence first and then builds a dictionary indexing each of the tokens by frequency.
After we've loaded the data in memory we pad each of the sentences with 00 so that we have two 25000×20025000×200 arrays for training and testing respectively.

```
vocab_size = 5000
sentence_size = 200
(x_train_variable, y_train), (x_test_variable, y_test) =
imdb.load_data(num_words=vocab_size)
x_train = sequence.pad_sequences(
```

```
    x_train_variable,
    maxlen=sentence_size,
    padding='post',
    value=0)
x_test = sequence.pad_sequences(
    x_test_variable,
    maxlen=sentence_size,
    padding='post',
    value=0)
```

### Input Functions

The Estimator framework uses *input functions* to split the data pipeline from the model itself. Several helper methods are available to create them, whether your data is in a `.csv` file, or in a `pandas.DataFrame`, whether it fits in memory or not. In our case, we can use `Dataset.from_tensor_slices` for both the train and test sets.

```
x_len_train = np.array([min(len(x), sentence_size) for x in x_train_variable])
x_len_test = np.array([min(len(x), sentence_size) for x in x_test_variable])

def parser(x, length, y):
    features = {"x": x, "len": length}
    return features, y

def train_input_fn():
    dataset = tf.data.Dataset.from_tensor_slices((x_train, x_len_train, y_train))
    dataset = dataset.shuffle(buffer_size=len(x_train_variable))
    dataset = dataset.batch(100)
    dataset = dataset.map(parser)
    dataset = dataset.repeat()
    iterator = dataset.make_one_shot_iterator()
    return iterator.get_next()

def eval_input_fn():
    dataset = tf.data.Dataset.from_tensor_slices((x_test, x_len_test, y_test))
    dataset = dataset.batch(100)
    dataset = dataset.map(parser)
    iterator = dataset.make_one_shot_iterator()
    return iterator.get_next()
```

We shuffle the training data and do not predefine the number of epochs we want to train, while we only need one epoch of the test data for evaluation. We also add an additional `"len"` key that captures the length of the original, unpadded sequence, which we will use later.

### Building a baseline

It’s good practice to start any machine learning project trying basic baselines. The simpler the better as having a simple and robust baseline is key to understanding exactly how much we are gaining in terms of performance by adding extra complexity. It may very well be the case that a simple solution is good enough for our requirements.

With that in mind, let us start by trying out one of the simplest models for text classification. That would be a sparse linear model that gives a weight to each token and adds up all of the results, regardless of the order. As this model does not care about the order of words in a sentence, we normally refer to it as a *Bag-of-Words*approach. Let’s see how we can implement this model using an Estimator.
We start out by defining the feature column that is used as input to our classifier. As we have seen in Part 2, categorical_column_with_identity is the right choice for this pre-processed text input. If we were feeding raw text tokens other feature_columnscould do a lot of the pre-processing for us. We can now use the pre-made LinearClassifier.

```
column = tf.feature_column.categorical_column_with_identity('x', vocab_size)
classifier = tf.estimator.LinearClassifier(
    feature_columns=[column],
    model_dir=os.path.join(model_dir, 'bow_sparse'))
```

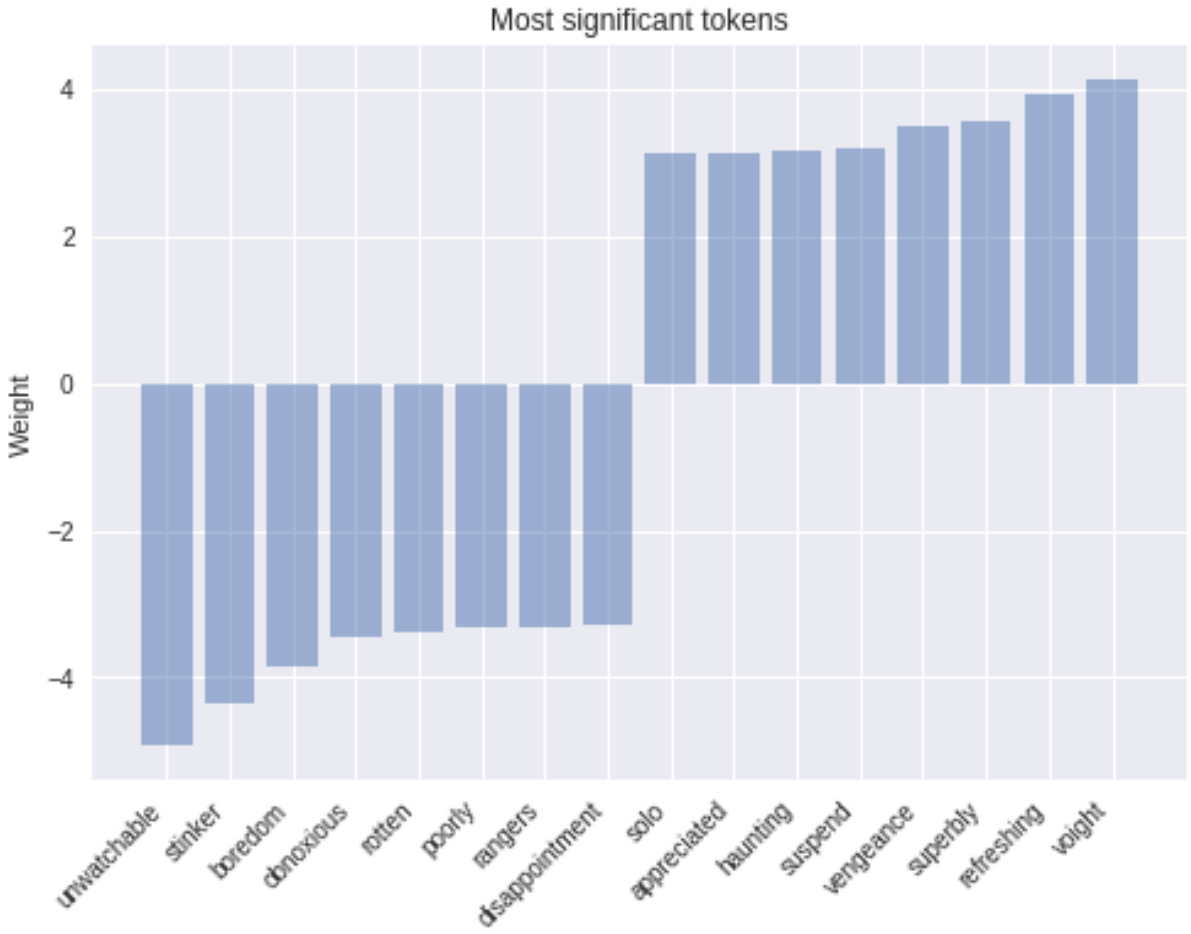
Finally, we create a simple function that trains the classifier and additionally creates a precision-recall curve. As we do not aim to maximize performance in this blog post, we only train our models for 25,000 steps.

```
def train_and_evaluate(classifier):
    classifier.train(input_fn=train_input_fn, steps=25000)
    eval_results = classifier.evaluate(input_fn=eval_input_fn)
    predictions = np.array([p['logistic'][0] for p in
classifier.predict(input_fn=eval_input_fn)])
    tf.reset_default_graph()
    # Add a PR summary in addition to the summaries that the classifier writes
```

```
  pr = summary_lib.pr_curve('precision_recall', predictions=predictions,
labels=y_test.astype(bool), num_thresholds=21)
  with tf.Session() as sess:
    writer = tf.summary.FileWriter(os.path.join(classifier.model_dir, 'eval'), sess.graph)
    writer.add_summary(sess.run(pr), global_step=0)
    writer.close()
train_and_evaluate(classifier)
```

One of the benefits of choosing a simple model is that it is much more interpretable. The more complex a model, the harder it is to inspect and the more it tends to work like a black box. In this example, we can load the weights from our model's last checkpoint and take a look at what tokens correspond to the biggest weights in absolute value. The results look like what we would expect.

```
# Load the tensor with the model weights
weights = classifier.get_variable_value('linear/linear_model/x/weights').flatten()
# Find biggest weights in absolute value
extremes = np.concatenate((sorted_indexes[-8:], sorted_indexes[:8]))
# word_inverted_index is a dictionary that maps from indexes back to tokens
extreme_weights = sorted(
  [(weights[i], word_inverted_index[i - index_offset]) for i in extremes])
# Create plot
y_pos = np.arange(len(extreme_weights))
plt.bar(y_pos, [pair[0] for pair in extreme_weights], align='center', alpha=0.5)
plt.xticks(y_pos, [pair[1] for pair in extreme_weights], rotation=45, ha='right')
plt.ylabel('Weight')
plt.title('Most significant tokens')
plt.show()
```




As we can see, tokens with the most positive weight such as ‘refreshing’ are clearly associated with positive sentiment, while tokens that have a large negative weight unarguably evoke negative emotions. A simple but powerful modification that one can do to improve this model is weighting the tokens by their tf-idf scores.

**Embeddings**

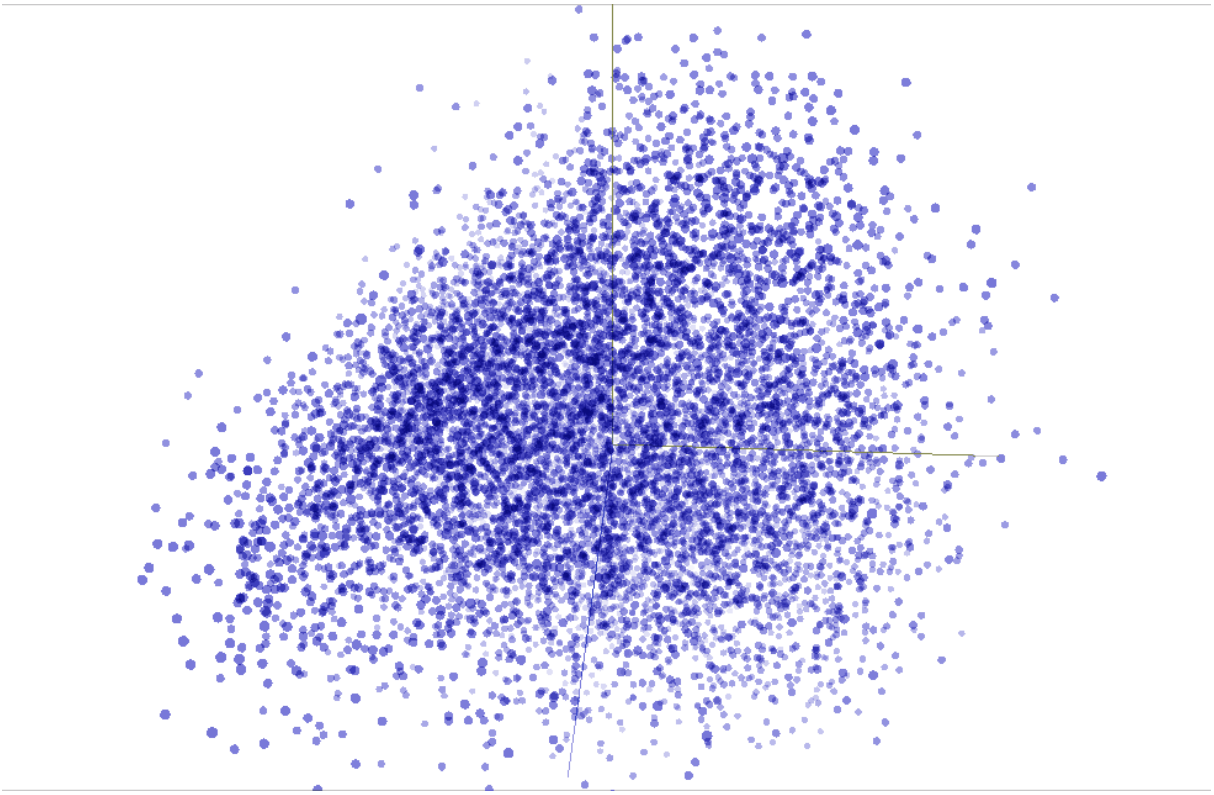
The next step of complexity we can add are word embeddings. Embeddings are a dense low-dimensional representation of sparse high-dimensional data. This allows our model to learn a more meaningful representation of each token, rather than just an index. While an individual dimension is not meaningful, the low-dimensional space—when learned from a large enough corpus—has been shown

to capture relations such as tense, plural, gender, thematic relatedness, and many more. We can add word embeddings by converting our existing feature column into an embedding_column. The representation seen by the model is the mean of the embeddings for each token (see the combiner argument in the docs). We can plug in the embedded features into a pre-canned DNNClassifier.

A note for the keen observer: an embedding_column is just an efficient way of applying a fully connected layer to the sparse binary feature vector of tokens, which is multiplied by a constant depending of the chosen combiner. A direct consequence of this is that it wouldn't make sense to use an embedding_column directly in a LinearClassifier because two consecutive linear layers without non-linearities in between add no prediction power to the model, unless of course the embeddings are pre-trained.

```
embedding_size = 50
word_embedding_column = tf.feature_column.embedding_column(
  column, dimension=embedding_size)
classifier = tf.estimator.DNNClassifier(
  hidden_units=[100],
  feature_columns=[word_embedding_column],
  model_dir=os.path.join(model_dir, 'bow_embeddings'))
train_and_evaluate(classifier)
```

We can use TensorBoard to visualize our 5050 dimensional word vectors projected into R3R3 using t-SNE. We expect similar words to be close to each other. This can be a useful way to inspect our model weights and find unexpected behaviours.

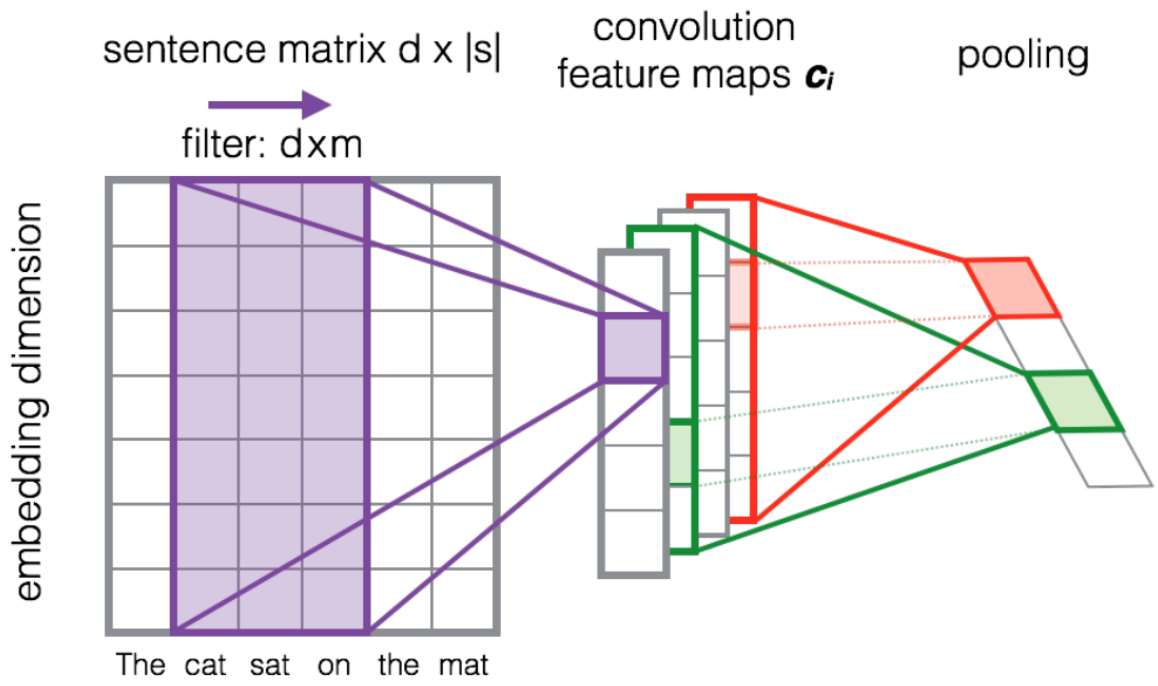


**Convolutions**

At this point one possible approach would be to go deeper, further adding more fully connected layers and playing around with layer sizes and training functions. However, by doing that we would add extra complexity and ignore important structure in our sentences. Words do not live in a vacuum and meaning is compositional, formed by words and its neighbors.

Convolutions are one way to take advantage of this structure, similar to how we can model salient clusters of pixels for image classification. The intuition is that certain sequences of words, or *n-grams*, usually have the same meaning regardless of their overall position in the sentence. Introducing a structural prior via the convolution operation allows us to model the interaction between neighboring words and consequently gives us a better way to represent such meaning.

The following image shows how a filter matrix $F \in R^{d \times m}$ $F \in R^{d \times m}$ tri-gram window of tokens to build a new feature map. Afterwards a *pooling* layer is usually applied to combine adjacent results.




Source: Learning to Rank Short Text Pairs with Convolutional Deep Neural Networks by **Severyn** et al. [2015]

Let us look at the full model architecture. The use of dropout layers is a regularization technique that makes the model less likely to overfit.

Embedding Layer
Dropout
Convolution1D
GlobalMaxPooling1D
Hidden Dense Layer
Dropout
Output Layer

**Creating a custom estimator**

As seen in previous blog posts, the tf.estimator framework provides a high-level API for training machine learning models, defining train(), evaluate() and predict()operations, handling checkpointing, loading, initializing, serving, building the graph and the session out of the box. There is a small family of pre-made estimators, like the ones we used earlier, but it's most likely that you will need to build your own.
Writing a custom estimator means writing a model_fn(features, labels, mode, params)that returns an EstimatorSpec. The first step will be mapping the features into our embedding layer:

```
input_layer = tf.contrib.layers.embed_sequence(
   features['x'],
   vocab_size,
   embedding_size,
   initializer=params['embedding_initializer'])
```

Then we use tf.layers to process each output sequentially.

```
training = (mode == tf.estimator.ModeKeys.TRAIN)
dropout_emb = tf.layers.dropout(inputs=input_layer,
                                rate=0.2,
                                training=training)
conv = tf.layers.conv1d(
   inputs=dropout_emb,
   filters=32,
   kernel_size=3,
   padding="same",
   activation=tf.nn.relu)
pool = tf.reduce_max(input_tensor=conv, axis=1)
hidden = tf.layers.dense(inputs=pool, units=250, activation=tf.nn.relu)
dropout = tf.layers.dropout(inputs=hidden, rate=0.2, training=training)
logits = tf.layers.dense(inputs=dropout_hidden, units=1)
```

Finally, we will use a Head to simplify the writing of our last part of the model_fn. The head already knows how to compute predictions, loss, train_op, metrics and export outputs, and can be reused across models. This is also used in the pre-made estimators and provides us with the benefit of a uniform evaluation function across all of our models. We will use binary_classification_head, which is a head for single label binary classification that

uses sigmoid_cross_entropy_with_logits as the loss function under the hood.

```
head = tf.contrib.estimator.binary_classification_head()
optimizer = tf.train.AdamOptimizer()
def _train_op_fn(loss):
   tf.summary.scalar('loss', loss)
   return optimizer.minimize(
      loss=loss,
      global_step=tf.train.get_global_step())

return head.create_estimator_spec(
   features=features,
   labels=labels,
   mode=mode,
   logits=logits,
   train_op_fn=_train_op_fn)
```

Running this model is just as easy as before:

```
initializer = tf.random_uniform([vocab_size, embedding_size], -1.0, 1.0))
params = {'embedding_initializer': initializer}
cnn_classifier = tf.estimator.Estimator(model_fn=model_fn,
                                        model_dir=os.path.join(model_dir, 'cnn'),
                                        params=params)
train_and_evaluate(cnn_classifier)
```

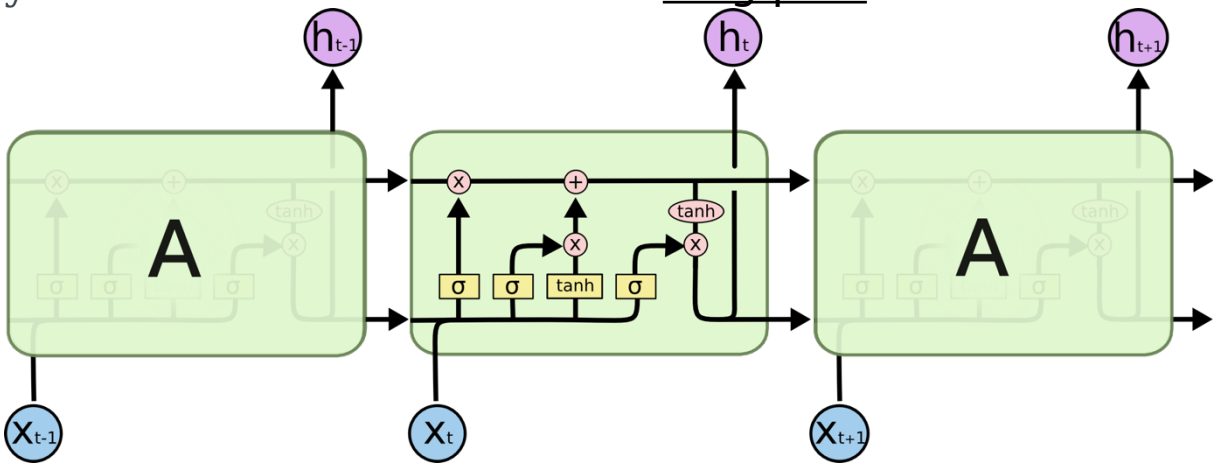
**LSTM Networks**

Using the Estimator API and the same model head, we can also create a classifier that uses a *Long Short-Term Memory* (*LSTM*) cell instead of convolutions. Recurrent models such as this are some of the most successful building blocks for NLP applications. An LSTM processes the entire document sequentially, recursing over the sequence with its cell while storing the current state of the sequence in its memory.

One of the drawbacks of recurrent models compared to CNNs is that, because of the nature of recursion, models turn out deeper and more complex, which usually produces slower training time and worse convergence. LSTMs (and RNNs in general) can suffer convergence issues like vanishing or exploding gradients, that said, with sufficient tuning they can obtain state-of-the-art results for many problems. As a rule of thumb CNNs are good at feature extraction,

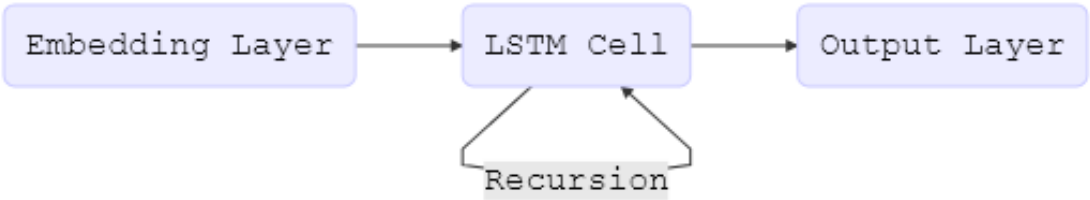
while RNNs excel at tasks that depend on the meaning of the whole sentence, like question answering or machine translation.

Each cell processes one token embedding at a time updating its internal state based on a differentiable computation that depends on both the embedding vector $x_t$ and the previous state $h_{t-1}$. In order to get a better understanding of how LSTMs work, you can refer to Chris Olah's blog post.




Source: Understanding LSTM Networks by **Chris Olah**

The complete LSTM model can be expressed by the following simple flowchart:




In the beginning of this post, we padded all documents up to 200200200 tokens, which is necessary to build a proper tensor. However, when a

document contains fewer than 200200200 words, we don't want the LSTM to continue processing padding tokens as it does not add information and degrades performance. For this reason, we additionally want to provide our network with the length of the original sequence before it was padded. Internally, the model then copies the last state through to the sequence's end. We can do this by using the "len" feature in our input functions. We can now use the same logic as above and simply replace the convolutional, pooling, and flatten layers with our LSTM cell.

```
lstm_cell = tf.nn.rnn_cell.BasicLSTMCell(100)
_, final_states = tf.nn.dynamic_rnn(
      lstm_cell, inputs, sequence_length=features['len'], dtype=tf.float32)
logits = tf.layers.dense(inputs=final_states.h, units=1)
```

**Pre-trained vectors**

Most of the models that we have shown before rely on word embeddings as a first layer. So far, we have initialized this embedding layer randomly. However, muchprevious work has shown that using embeddings pre-trained on a large unlabeled corpus as initialization is beneficial, particularly when training on only a small number of labeled examples. The most popular pre-trained embedding is word2vec. Leveraging knowledge from unlabeled data via pre-trained embeddings is an instance of *transfer learning*.

To this end, we will show you how to use them in an Estimator. We will use the pre-trained vectors from another popular model, GloVe.

```
embeddings = {}
with open('glove.6B.50d.txt', 'r', encoding='utf-8') as f:
   for line in f:
      values = line.strip().split()
      w = values[0]
      vectors = np.asarray(values[1:], dtype='float32')
      embeddings[w] = vectors
```

After loading the vectors into memory from a file we store them as a numpy.array using the same indexes as

our vocabulary. The created array is of shape (5000, 50). At every row index, it contains the 50-dimensional vector representing the word at the same index in our vocabulary.

```
embedding_matrix = np.random.uniform(-1, 1, size=(vocab_size, embedding_size))
for w, i in word_index.items():
  v = embeddings.get(w)
  if v is not None and i < vocab_size:
    embedding_matrix[i] = v
```

Finally, we can use a custom initializer function and pass it in the params object to our cnn_model_fn , without any modifications.

```
def my_initializer(shape=None, dtype=tf.float32, partition_info=None):
  assert dtype is tf.float32
  return embedding_matrix
params = {'embedding_initializer': my_initializer}
cnn_pretrained_classifier = tf.estimator.Estimator(
  model_fn=cnn_model_fn,
  model_dir=os.path.join(model_dir, 'cnn_pretrained'),
  params=params)
train_and_evaluate(cnn_pretrained_classifier)
```

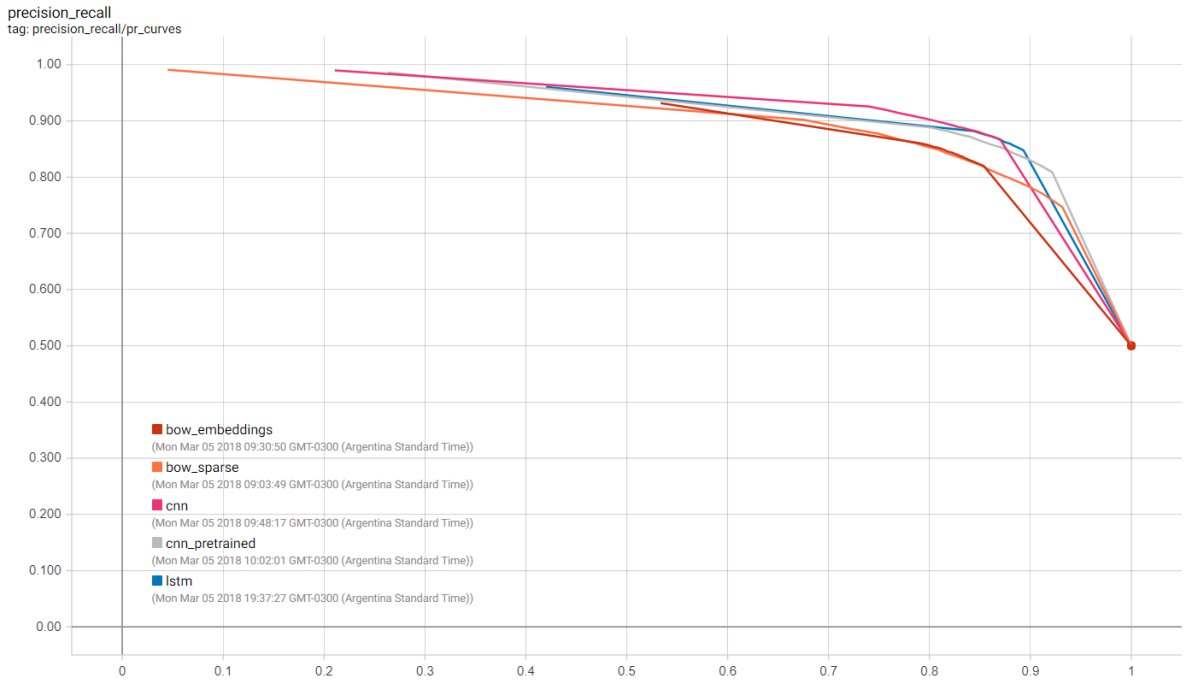
**Running TensorBoard**

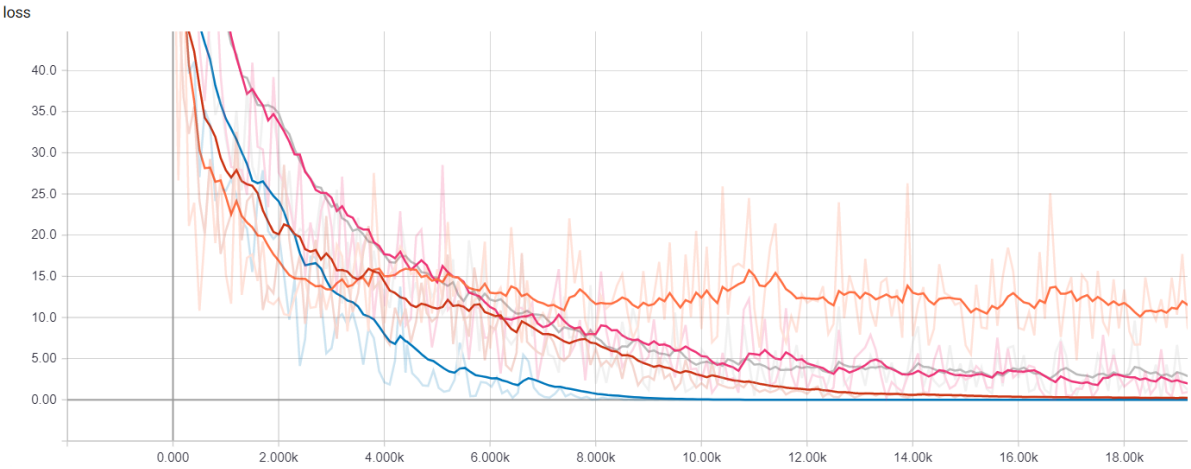
Now we can launch TensorBoard and see how the different models we've trained compare against each other in terms of training time and performance.

In a terminal, we run

```
> tensorboard --logdir={model_dir}
```

We can visualize many metrics collected while training and testing, including the loss function values of each model at each training step, and the precision-recall curves. This is of course most useful to select which model works best for our use-case as well as how to choose classification thresholds.




## Getting Predictions

To obtain predictions on new sentences we can use the predict method in the Estimator instances, which will load the latest checkpoint for each model and evaluate on the unseen examples. But before passing the data into the model we have to clean up, tokenize and map each token to the corresponding index as we see below.

```
def text_to_index(sentence):
    # Remove punctuation characters except for the apostrophe
    translator = str.maketrans('', '', string.punctuation.replace("'", ''))
    tokens = sentence.translate(translator).lower().split()
```

```
    return np.array([1] + [word_index[t] + index_offset if t in word_index else 2 for t in
tokens])

def print_predictions(sentences, classifier):
    indexes = [text_to_index(sentence) for sentence in sentences]
    x = sequence.pad_sequences(indexes,
                               maxlen=sentence_size,
                               padding='post',
                               value=-1)
    length = np.array([min(len(x), sentence_size) for x in indexes])
    predict_input_fn = tf.estimator.inputs.numpy_input_fn(x={"x": x, "len": length},
shuffle=False)
    predictions = [p['logistic'][0] for p in classifier.predict(input_fn=predict_input_fn)]
    print(predictions)
```

It is worth noting that the checkpoint itself is not sufficient to make predictions; the actual code used to build the estimator is necessary as well in order to map the saved weights to the corresponding tensors. It's a good practice to associate saved checkpoints with the branch of code with which they were created.

If you are interested in exporting the models to disk in a fully recoverable way, you might want to look into the SavedModel class, which is especially useful for serving your model through an API using TensorFlow Serving.

**Summary**

In this blog post, we explored how to use estimators for text classification, in particular for the IMDB Reviews Dataset. We trained and visualized our own embeddings, as well as loaded pre-trained ones. We started from a simple baseline and made our way to convolutional neural networks and LSTMs.

# How to Develop 1D Convolutional Neural Network Models for Human Activity Recognition

by **Jason Brownlee** on September 21, 2018 in **Deep Learning for Time Series**

Human activity recognition is the problem of classifying sequences of accelerometer data recorded by specialized harnesses or smart phones into known well-defined movements.

Classical approaches to the problem involve hand crafting features from the time series data based on fixed-sized windows and training machine learning models, such as ensembles of decision trees. The difficulty is that this feature engineering requires deep expertise in the field.

Recently, deep learning methods such as recurrent neural networks and one-dimensional convolutional neural networks, or CNNs, have been shown to provide state-of-the-art results on challenging activity recognition tasks with little or no data feature engineering, instead using feature learning on raw data.

In this tutorial, you will discover how to develop one-dimensional convolutional neural networks for time series classification on the problem of human activity recognition.

After completing this tutorial, you will know:

- How to load and prepare the data for a standard human activity recognition dataset and develop a single 1D CNN model that achieves excellent performance on the raw data.
- How to further tune the performance of the model, including data transformation, filter maps, and kernel sizes.
- How to develop a sophisticated multi-headed one-dimensional convolutional neural network model that provides an ensemble-like result.

## Tutorial Overview

This tutorial is divided into four parts; they are:

1. Activity Recognition Using Smartphones Dataset
2. Develop 1D Convolutional Neural Network
3. Tuned 1D Convolutional Neural Network
4. Multi-Headed 1D Convolutional Neural Network

## Activity Recognition Using Smartphones Dataset

Human Activity Recognition, or HAR for short, is the problem of predicting what a person is doing based on a trace of their movement using sensors. A standard human activity recognition dataset is the ‘Activity Recognition Using Smart Phones Dataset’ made available in 2012.

It was prepared and made available by Davide Anguita, et al. from the University of Genova, Italy and is described in full in their 2013 paper "A Public Domain Dataset for Human Activity Recognition Using Smartphones."
The dataset was modeled with machine learning algorithms in their 2012 paper titled "Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine."
The dataset was made available and can be downloaded for free from the UCI Machine Learning Repository:

- Human Activity Recognition Using Smartphones Data Set, UCI Machine Learning Repository

The data was collected from 30 subjects aged between 19 and 48 years old performing one of six standard activities while wearing a waist-mounted smartphone that recorded the movement data. Video was recorded of each subject performing the activities and the movement data was labeled manually from these videos.

Below is an example video of a subject performing the activities while their movement data is being recorded.

The six activities performed were as follows:

1. Walking
2. Walking Upstairs
3. Walking Downstairs
4. Sitting
5. Standing
6. Laying

The movement data recorded was the x, y, and z accelerometer data (linear acceleration) and gyroscopic data (angular velocity) from the smart phone, specifically a Samsung Galaxy S II. Observations were recorded at 50 Hz (i.e. 50 data points per second). Each subject performed the sequence of activities twice, once with the device on their left-hand-side and once with the device on their right-hand side.

The raw data is not available. Instead, a pre-processed version of the dataset was made available. The pre-processing steps included:

- Pre-processing accelerometer and gyroscope using noise filters.
- Splitting data into fixed windows of 2.56 seconds (128 data points) with 50% overlap.
- Splitting of accelerometer data into gravitational (total) and body motion components.

Feature engineering was applied to the window data, and a copy of the data with these engineered features was made available.

A number of time and frequency features commonly used in the field of human activity recognition were extracted from each window. The result was a 561 element vector of features.

The dataset was split into train (70%) and test (30%) sets based on data for subjects, e.g. 21 subjects for train and nine for test.

Experiment results with a support vector machine intended for use on a smartphone (e.g. fixed-point arithmetic) resulted in a predictive accuracy of 89% on the test dataset, achieving similar results as an unmodified SVM implementation.

The dataset is freely available and can be downloaded from the UCI Machine Learning repository.

The data is provided as a single zip file that is about 58 megabytes in size. The direct

# Develop 1D Convolutional Neural Network

In this section, we will develop a one-dimensional convolutional neural network model (1D CNN) for the human activity recognition dataset.

Convolutional neural network models were developed for image classification problems, where the model learns an internal representation of a two-dimensional input, in a process referred to as feature learning.
This same process can be harnessed on one-dimensional sequences of data, such as in the case of acceleration and gyroscopic data for human activity recognition. The model learns to extract features from sequences of observations and how to map the internal features to different activity types.

The benefit of using CNNs for sequence classification is that they can learn from the raw time series data directly, and in turn do not require domain expertise to manually engineer input features. The model can learn an internal representation of the time series data and ideally achieve comparable performance to models fit on a version of the dataset with engineered features.

This section is divided into 4 parts; they are:

1. Load Data

2. Fit and Evaluate Model
3. Summarize Results
4. Complete Example

## Load Data

The first step is to load the raw dataset into memory.

There are three main signal types in the raw data: total acceleration, body acceleration, and body gyroscope. Each has three axes of data. This means that there are a total of nine variables for each time step.

Further, each series of data has been partitioned into overlapping windows of 2.65 seconds of data, or 128 time steps. These windows of data correspond to the windows of engineered features (rows) in the previous section.

This means that one row of data has (128 * 9), or 1,152, elements. This is a little less than double the size of the 561 element vectors in the previous section and it is likely that there is some redundant data.

The signals are stored in the */Inertial Signals/* directory under the train and test subdirectories. Each axis of each signal is stored in a separate file, meaning that each of the train and test datasets have nine input files to load and one output file to load. We can batch the loading of these files into groups given the consistent directory structures and file naming conventions. The input data is in CSV format where columns are separated by whitespace. Each of these files can be loaded as a NumPy array. The *load_file()* function below loads a dataset given the file path to the file and returns the loaded data as a NumPy array.

```
1 # load a single file as a numpy array
2 def load_file(filepath):
3         dataframe = read_csv(filepath, header=None, delim_whitespace=True)
4         return dataframe.values
```

We can then load all data for a given group (train or test) into a single three-dimensional NumPy array, where the dimensions of the array are [*samples, time steps, features*].
To make this clearer, there are 128 time steps and nine features, where the number of samples is the number of rows in any given raw signal data file.

The *load_group()* function below implements this behavior. The dstack() NumPy functionallows us to stack each of the loaded 3D arrays into a single 3D array where the variables are separated on the third dimension (features).

```
1 # load a list of files into a 3D array of [samples, timesteps, features]
2 def load_group(filenames, prefix=''):
3         loaded = list()
```

```
        for name in filenames:
                data = load_file(prefix + name)
                loaded.append(data)
        # stack group so that features are the 3rd dimension
        loaded = dstack(loaded)
        return loaded
```

We can use this function to load all input signal data for a given group, such as train or test.

The *load_dataset_group()* function below loads all input signal data and the output data for a single group using the consistent naming conventions between the train and test directories.

```
# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
        filepath = prefix + group + '/Inertial Signals/'
        # load all 9 files as a single array
        filenames = list()
        # total acceleration
        filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
'total_acc_z_'+group+'.txt']
        # body acceleration
        filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
'body_acc_z_'+group+'.txt']
        # body gyroscope
        filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
'body_gyro_z_'+group+'.txt']
        # load input data
        X = load_group(filenames, filepath)
        # load class output
        y = load_file(prefix + group + '/y_'+group+'.txt')
        return X, y
```

Finally, we can load each of the train and test datasets.

The output data is defined as an integer for the class number. We must one hot encode these class integers so that the data is suitable for fitting a neural network multi-class classification model. We can do this by calling the to_categorical() Keras function.

The *load_dataset()* function below implements this behavior and returns the train and test *X*and *y* elements ready for fitting and evaluating the defined models.

```
# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
        # load all train
        trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
        print(trainX.shape, trainy.shape)
        # load all test
        testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
        print(testX.shape, testy.shape)
        # zero-offset class values
        trainy = trainy - 1
        testy = testy - 1
        # one hot encode y
        trainy = to_categorical(trainy)
        testy = to_categorical(testy)
        print(trainX.shape, trainy.shape, testX.shape, testy.shape)
        return trainX, trainy, testX, testy
```

## Fit and Evaluate Model

Now that we have the data loaded into memory ready for modeling, we can define, fit, and evaluate a 1D CNN model.

We can define a function named *evaluate_model()* that takes the train and test dataset, fits a model on the training dataset, evaluates it on the test dataset, and returns an estimate of the models performance.
First, we must define the CNN model using the Keras deep learning library. The model requires a three-dimensional input with [*samples, time steps, features*].
This is exactly how we have loaded the data, where one sample is one window of the time series data, each window has 128 time steps, and a time step has nine variables or features.

The output for the model will be a six-element vector containing the probability of a given window belonging to each of the six activity types.

These input and output dimensions are required when fitting the model, and we can extract them from the provided training dataset.

```
1 n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
```

The model is defined as a Sequential Keras model, for simplicity.

We will define the model as having two 1D CNN layers, followed by a dropout layer for regularization, then a pooling layer. It is common to define CNN layers in groups of two in order to give the model a good chance of learning features from the input data. CNNs learn very quickly, so the dropout layer is intended to help slow down the learning process and hopefully result in a better final model. The pooling layer reduces the learned features to 1/4 their size, consolidating them to only the most essential elements.

After the CNN and pooling, the learned features are flattened to one long vector and pass through a fully connected layer before the output layer used to make a prediction. The fully connected layer ideally provides a buffer between the learned features and the output with the intent of interpreting the learned features before making a prediction.

For this model, we will use a standard configuration of 64 parallel feature maps and a kernel size of 3. The feature maps are the number of times the input is processed or interpreted, whereas the kernel size is the number of

input time steps considered as the input sequence is read or processed onto the feature maps.

The efficient Adam version of stochastic gradient descent will be used to optimize the network, and the categorical cross entropy loss function will be used given that we are learning a multi-class classification problem.
The definition of the model is listed below.

```
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(n_timesteps,n_features)))
model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
model.add(Dropout(0.5))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

The model is fit for a fixed number of epochs, in this case 10, and a batch size of 32 samples will be used, where 32 windows of data will be exposed to the model before the weights of the model are updated.

Once the model is fit, it is evaluated on the test dataset and the accuracy of the fit model on the test dataset is returned.

The complete *evaluate_model()* function is listed below.

```
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
	verbose, epochs, batch_size = 0, 10, 32
	n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
	model = Sequential()
	model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(n_timesteps,n_features)))
	model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
	model.add(Dropout(0.5))
	model.add(MaxPooling1D(pool_size=2))
	model.add(Flatten())
	model.add(Dense(100, activation='relu'))
	model.add(Dense(n_outputs, activation='softmax'))
	model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
	# fit network
	model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
	# evaluate model
	_, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
	return accuracy
```

There is nothing special about the network structure or chosen hyperparameters; they are just a starting point for this problem.

## Summarize Results

We cannot judge the skill of the model from a single evaluation.

The reason for this is that neural networks are stochastic, meaning that a different specific model will result when training the same model configuration on the same data.

This is a feature of the network in that it gives the model its adaptive ability, but requires a slightly more complicated evaluation of the model.

We will repeat the evaluation of the model multiple times, then summarize the performance of the model across each of those runs. For example, we can call *evaluate_model()* a total of 10 times. This will result in a population of model evaluation scores that must be summarized.

```
# repeat experiment
scores = list()
for r in range(repeats):
	score = evaluate_model(trainX, trainy, testX, testy)
	score = score * 100.0
	print('>#%d: %.3f' % (r+1, score))
	scores.append(score)
```

We can summarize the sample of scores by calculating and reporting the mean and standard deviation of the performance. The mean gives the average accuracy of the model on the dataset, whereas the standard deviation gives the average variance of the accuracy from the mean.

The function *summarize_results()* below summarizes the results of a run.

```
# summarize scores
def summarize_results(scores):
	print(scores)
	m, s = mean(scores), std(scores)
	print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))
```

We can bundle up the repeated evaluation, gathering of results, and summarization of results into a main function for the experiment, called *run_experiment()*, listed below.
By default, the model is evaluated 10 times before the performance of the model is reported.

```
# run an experiment
def run_experiment(repeats=10):
	# load data
	trainX, trainy, testX, testy = load_dataset()
	# repeat experiment
	scores = list()
	for r in range(repeats):
		score = evaluate_model(trainX, trainy, testX, testy)
		score = score * 100.0
		print('>#%d: %.3f' % (r+1, score))
		scores.append(score)
	# summarize results
	summarize_results(scores)
```

## Complete Example

Now that we have all of the pieces, we can tie them together into a worked example.

The complete code listing is provided below.

```
# cnn model
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
	dataframe = read_csv(filepath, header=None, delim_whitespace=True)
	return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
	loaded = list()
	for name in filenames:
		data = load_file(prefix + name)
		loaded.append(data)
	# stack group so that features are the 3rd dimension
	loaded = dstack(loaded)
	return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
	filepath = prefix + group + '/Inertial Signals/'
	# load all 9 files as a single array
	filenames = list()
	# total acceleration
	filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
'total_acc_z_'+group+'.txt']
	# body acceleration
	filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
'body_acc_z_'+group+'.txt']
	# body gyroscope
	filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
'body_gyro_z_'+group+'.txt']
	# load input data
	X = load_group(filenames, filepath)
	# load class output
	y = load_file(prefix + group + '/y_'+group+'.txt')
	return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
	# load all train
	trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
	print(trainX.shape, trainy.shape)
	# load all test
	testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
	print(testX.shape, testy.shape)
	# zero-offset class values
```

```
        trainy = trainy - 1
        testy = testy - 1
        # one hot encode y
        trainy = to_categorical(trainy)
        testy = to_categorical(testy)
        print(trainX.shape, trainy.shape, testX.shape, testy.shape)
        return trainX, trainy, testX, testy

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
        verbose, epochs, batch_size = 0, 10, 32
        n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
        model = Sequential()
        model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(n_timesteps,n_features)))
        model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
        model.add(Dropout(0.5))
        model.add(MaxPooling1D(pool_size=2))
        model.add(Flatten())
        model.add(Dense(100, activation='relu'))
        model.add(Dense(n_outputs, activation='softmax'))
        model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
        # fit network
        model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
        # evaluate model
        _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
        return accuracy

# summarize scores
def summarize_results(scores):
        print(scores)
        m, s = mean(scores), std(scores)
        print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))

# run an experiment
def run_experiment(repeats=10):
        # load data
        trainX, trainy, testX, testy = load_dataset()
        # repeat experiment
        scores = list()
        for r in range(repeats):
                score = evaluate_model(trainX, trainy, testX, testy)
                score = score * 100.0
                print('>#%d: %.3f' % (r+1, score))
                scores.append(score)
        # summarize results
        summarize_results(scores)

# run the experiment
run_experiment()
```

Running the example first prints the shape of the loaded dataset, then the shape of the train and test sets and the input and output elements. This confirms the number of samples, time steps, and variables, as well as the number of classes.

Next, models are created and evaluated and a debug message is printed for each.

Finally, the sample of scores is printed followed by the mean and standard deviation. We can see that the model performed well achieving a classification accuracy of about 90.9% trained on the raw dataset, with a standard deviation of about 1.3.

This is a good result, considering that the original paper published a result of 89%, trained on the dataset with heavy domain-specific feature engineering, not the raw dataset.

Note: Given the stochastic nature of the algorithm, your specific results may vary.

```
   (7352, 128, 9) (7352, 1)
1  (2947, 128, 9) (2947, 1)
2  (7352, 128, 9) (7352, 6) (2947, 128, 9) (2947, 6)
3
4  >#1: 91.347
5  >#2: 91.551
6  >#3: 90.804
7  >#4: 90.058
8  >#5: 89.752
9  >#6: 90.940
10 >#7: 91.347
11 >#8: 87.547
12 >#9: 92.637
13 >#10: 91.890
14
15 [91.34713267729894, 91.55072955548015, 90.80420766881574, 90.05768578215134,
16 89.75229046487954, 90.93993892093654, 91.34713267729894, 87.54665761791652,
17 92.63657957244655, 91.89005768578215]
18
   Accuracy: 90.787% (+/-1.341)
```

Now that we have seen how to load the data and fit a 1D CNN model, we can investigate whether we can further lift the skill of the model with some hyperparameter tuning.

# Tuned 1D Convolutional Neural Network

In this section, we will tune the model in an effort to further improve performance on the problem.

We will look at three main areas:

1. Data Preparation
2. Number of Filters
3. Size of Kernel

## Data Preparation

In the previous section, we did not perform any data preparation. We used the data as-is.

Each of the main sets of data (body acceleration, body gyroscopic, and total acceleration) have been scaled to the range -1, 1. It is not clear if the data was scaled per-subject or across all subjects.

One possible transform that may result in an improvement is to standardize the observations prior to fitting a model.

Standardization refers to shifting the distribution of each variable such that it has a mean of zero and a standard deviation of 1. It really only makes sense if the distribution of each variable is Gaussian.

We can quickly check the distribution of each variable by plotting a histogram of each variable in the training dataset.

A minor difficulty in this is that the data has been split into windows of 128 time steps, with a 50% overlap. Therefore, in order to get a fair idea of the data distribution, we must first remove the duplicated observations (the overlap), then remove the windowing of the data.

We can do this using NumPy, first slicing the array and only keeping the second half of each window, then flattening the windows into a long vector for each variable. This is quick and dirty and does mean that we lose the data in the first half of the first window.

```
# remove overlap
cut = int(trainX.shape[1] / 2)
longX = trainX[:, -cut:, :]
# flatten windows
longX = longX.reshape((longX.shape[0] * longX.shape[1], longX.shape[2]))
```

The complete example of loading the data, flattening it, and plotting a histogram for each of the nine variables is listed below.

```
# plot distributions
from numpy import dstack
from pandas import read_csv
from keras.utils import to_categorical
from matplotlib import pyplot

# load a single file as a numpy array
def load_file(filepath):
	dataframe = read_csv(filepath, header=None, delim_whitespace=True)
	return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
	loaded = list()
	for name in filenames:
		data = load_file(prefix + name)
		loaded.append(data)
```

```
		# stack group so that features are the 3rd dimension
		loaded = dstack(loaded)
		return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
		filepath = prefix + group + '/Inertial Signals/'
		# load all 9 files as a single array
		filenames = list()
		# total acceleration
		filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt', 'total_acc_z_'+group+'.txt']
		# body acceleration
		filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt', 'body_acc_z_'+group+'.txt']
		# body gyroscope
		filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt', 'body_gyro_z_'+group+'.txt']
		# load input data
		X = load_group(filenames, filepath)
		# load class output
		y = load_file(prefix + group + '/y_'+group+'.txt')
		return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
		# load all train
		trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
		print(trainX.shape, trainy.shape)
		# load all test
		testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
		print(testX.shape, testy.shape)
		# zero-offset class values
		trainy = trainy - 1
		testy = testy - 1
		# one hot encode y
		trainy = to_categorical(trainy)
		testy = to_categorical(testy)
		print(trainX.shape, trainy.shape, testX.shape, testy.shape)
		return trainX, trainy, testX, testy

# plot a histogram of each variable in the dataset
def plot_variable_distributions(trainX):
		# remove overlap
		cut = int(trainX.shape[1] / 2)
		longX = trainX[:, -cut:, :]
		# flatten windows
		longX = longX.reshape((longX.shape[0] * longX.shape[1], longX.shape[2]))
		print(longX.shape)
		pyplot.figure()
		xaxis = None
		for i in range(longX.shape[1]):
			ax = pyplot.subplot(longX.shape[1], 1, i+1, sharex=xaxis)
			ax.set_xlim(-1, 1)
			if i == 0:
				xaxis = ax
			pyplot.hist(longX[:, i], bins=100)
		pyplot.show()

# load data
trainX, trainy, testX, testy = load_dataset()
# plot histograms
plot_variable_distributions(trainX)
```

Running the example creates a figure with nine histogram plots, one for each variable in the training dataset.

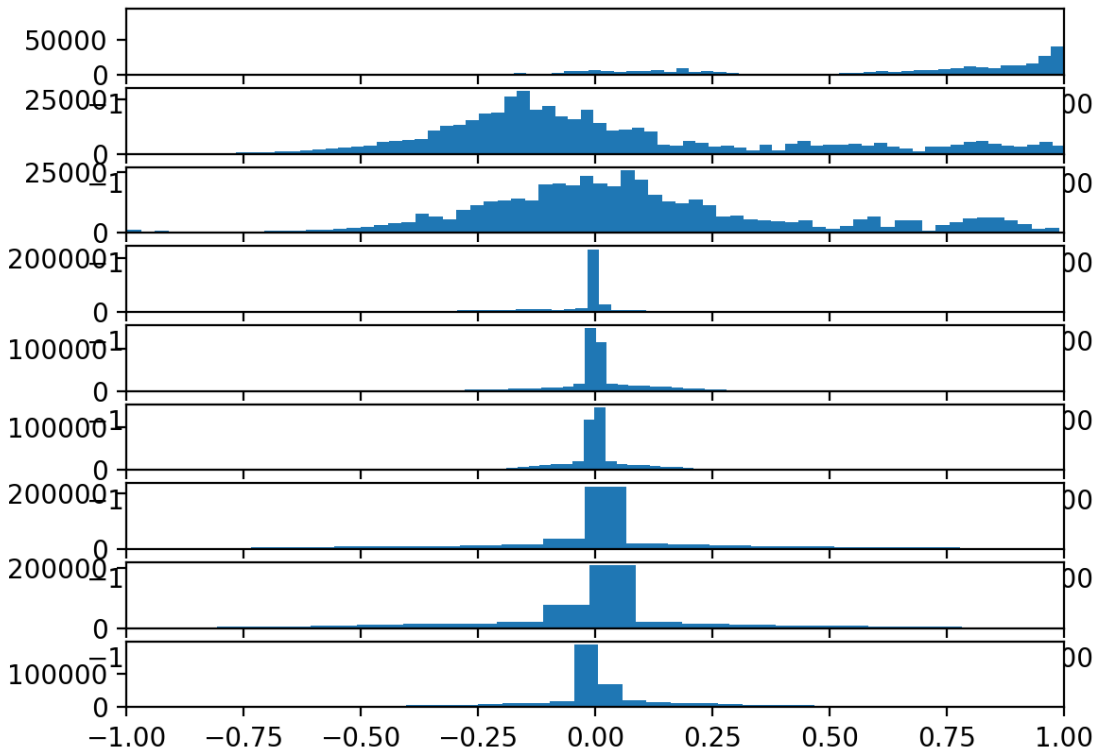
The order of the plots matches the order in which the data was loaded, specifically:

1. Total Acceleration x
2. Total Acceleration y
3. Total Acceleration z
4. Body Acceleration x
5. Body Acceleration y
6. Body Acceleration z
7. Body Gyroscope x
8. Body Gyroscope y
9. Body Gyroscope z

We can see that each variable has a Gaussian-like distribution, except perhaps the first variable (Total Acceleration x).

The distributions of total acceleration data is flatter than the body data, which is more pointed.

We could explore using a power transform on the data to make the distributions more Gaussian, although this is left as an exercise.




Histograms of each variable in the training data set

The data is sufficiently Gaussian-like to explore whether a standardization transform will help the model extract salient signal from the raw observations.

The function below named *scale_data()* can be used to standardize the data prior to fitting and evaluating the model. The StandardScaler scikit-learn class will be used to perform the transform. It is first fit on the training data (e.g. to find the mean and standard deviation for each variable), then applied to the train and test sets.
The standardization is optional, so we can apply the process and compare the results to the same code path without the standardization in a controlled experiment.

```
# standardize data
def scale_data(trainX, testX, standardize):
	# remove overlap
	cut = int(trainX.shape[1] / 2)
	longX = trainX[:, -cut:, :]
	# flatten windows
	longX = longX.reshape((longX.shape[0] * longX.shape[1], longX.shape[2]))
	# flatten train and test
	flatTrainX = trainX.reshape((trainX.shape[0] * trainX.shape[1], trainX.shape[2]))
	flatTestX = testX.reshape((testX.shape[0] * testX.shape[1], testX.shape[2]))
	# standardize
	if standardize:
		s = StandardScaler()
		# fit on training data
		s.fit(longX)
		# apply to training and test data
		longX = s.transform(longX)
		flatTrainX = s.transform(flatTrainX)
		flatTestX = s.transform(flatTestX)
	# reshape
	flatTrainX = flatTrainX.reshape((trainX.shape))
	flatTestX = flatTestX.reshape((testX.shape))
	return flatTrainX, flatTestX
```

We can update the *evaluate_model()* function to take a parameter, then use this parameter to decide whether or not to perform the standardization.

```
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy, param):
	verbose, epochs, batch_size = 0, 10, 32
	n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
	# scale data
	trainX, testX = scale_data(trainX, testX, param)
	model = Sequential()
	model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(n_timesteps,n_features)))
	model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
	model.add(Dropout(0.5))
	model.add(MaxPooling1D(pool_size=2))
	model.add(Flatten())
	model.add(Dense(100, activation='relu'))
	model.add(Dense(n_outputs, activation='softmax'))
	model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
	# fit network
	model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
	# evaluate model
	_, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
```

```
	return accuracy
```

We can also update the *run_experiment()* to repeat the experiment 10 times for each parameter; in this case, only two parameters will be evaluated [*False, True*] for no standardization and standardization respectively.

```
# run an experiment
def run_experiment(params, repeats=10):
	# load data
	trainX, trainy, testX, testy = load_dataset()
	# test each parameter
	all_scores = list()
	for p in params:
		# repeat experiment
		scores = list()
		for r in range(repeats):
			score = evaluate_model(trainX, trainy, testX, testy, p)
			score = score * 100.0
			print('>p=%d #%d: %.3f' % (p, r+1, score))
			scores.append(score)
		all_scores.append(scores)
	# summarize results
	summarize_results(all_scores, params)
```

This will result in two samples of results that can be compared.

We will update the *summarize_results()* function to summarize the sample of results for each configuration parameter and to create a boxplot to compare each sample of results.

```
# summarize scores
def summarize_results(scores, params):
	print(scores, params)
	# summarize mean and standard deviation
	for i in range(len(scores)):
		m, s = mean(scores[i]), std(scores[i])
		print('Param=%d: %.3f%% (+/-%.3f)' % (params[i], m, s))
	# boxplot of scores
	pyplot.boxplot(scores, labels=params)
	pyplot.savefig('exp_cnn_standardize.png')
```

These updates will allow us to directly compare the results of a model fit as before and a model fit on the dataset after it has been standardized.

It is also a generic change that will allow us to evaluate and compare the results of other sets of parameters in the following sections.

The complete code listing is provided below.

```
# cnn model with standardization
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
```

```
from keras.layers.convolutional import MaxPooling1D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
	dataframe = read_csv(filepath, header=None, delim_whitespace=True)
	return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
	loaded = list()
	for name in filenames:
		data = load_file(prefix + name)
		loaded.append(data)
	# stack group so that features are the 3rd dimension
	loaded = dstack(loaded)
	return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
	filepath = prefix + group + '/Inertial Signals/'
	# load all 9 files as a single array
	filenames = list()
	# total acceleration
	filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
'total_acc_z_'+group+'.txt']
	# body acceleration
	filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
'body_acc_z_'+group+'.txt']
	# body gyroscope
	filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
'body_gyro_z_'+group+'.txt']
	# load input data
	X = load_group(filenames, filepath)
	# load class output
	y = load_file(prefix + group + '/y_'+group+'.txt')
	return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
	# load all train
	trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
	print(trainX.shape, trainy.shape)
	# load all test
	testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
	print(testX.shape, testy.shape)
	# zero-offset class values
	trainy = trainy - 1
	testy = testy - 1
	# one hot encode y
	trainy = to_categorical(trainy)
	testy = to_categorical(testy)
	print(trainX.shape, trainy.shape, testX.shape, testy.shape)
	return trainX, trainy, testX, testy

# standardize data
def scale_data(trainX, testX, standardize):
	# remove overlap
	cut = int(trainX.shape[1] / 2)
	longX = trainX[:, -cut:, :]
	# flatten windows
	longX = longX.reshape((longX.shape[0] * longX.shape[1], longX.shape[2]))
	# flatten train and test
	flatTrainX = trainX.reshape((trainX.shape[0] * trainX.shape[1], trainX.shape[2]))
	flatTestX = testX.reshape((testX.shape[0] * testX.shape[1], testX.shape[2]))
	# standardize
```

```
        if standardize:
                s = StandardScaler()
                # fit on training data
                s.fit(longX)
                # apply to training and test data
                longX = s.transform(longX)
                flatTrainX = s.transform(flatTrainX)
                flatTestX = s.transform(flatTestX)
        # reshape
        flatTrainX = flatTrainX.reshape((trainX.shape))
        flatTestX = flatTestX.reshape((testX.shape))
        return flatTrainX, flatTestX

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy, param):
        verbose, epochs, batch_size = 0, 10, 32
        n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
        # scale data
        trainX, testX = scale_data(trainX, testX, param)
        model = Sequential()
        model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
input_shape=(n_timesteps,n_features)))
        model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
        model.add(Dropout(0.5))
        model.add(MaxPooling1D(pool_size=2))
        model.add(Flatten())
        model.add(Dense(100, activation='relu'))
        model.add(Dense(n_outputs, activation='softmax'))
        model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
        # fit network
        model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
        # evaluate model
        _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
        return accuracy

# summarize scores
def summarize_results(scores, params):
        print(scores, params)
        # summarize mean and standard deviation
        for i in range(len(scores)):
                m, s = mean(scores[i]), std(scores[i])
                print('Param=%s: %.3f%% (+/-%.3f)' % (params[i], m, s))
        # boxplot of scores
        pyplot.boxplot(scores, labels=params)
        pyplot.savefig('exp_cnn_standardize.png')

# run an experiment
def run_experiment(params, repeats=10):
        # load data
        trainX, trainy, testX, testy = load_dataset()
        # test each parameter
        all_scores = list()
        for p in params:
                # repeat experiment
                scores = list()
                for r in range(repeats):
                        score = evaluate_model(trainX, trainy, testX, testy, p)
                        score = score * 100.0
                        print('>p=%s #%d: %.3f' % (p, r+1, score))
                        scores.append(score)
                all_scores.append(scores)
        # summarize results
        summarize_results(all_scores, params)

    # run the experiment
```

```
n_params = [False, True]
run_experiment(n_params)
```

Running the example may take a few minutes, depending on your hardware.

The performance is printed for each evaluated model. At the end of the run, the performance of each of the tested configurations is summarized showing the mean and the standard deviation.

We can see that it does look like standardizing the dataset prior to modeling does result in a small lift in performance from about 90.4% accuracy (close to what we saw in the previous section) to about 91.5% accuracy.

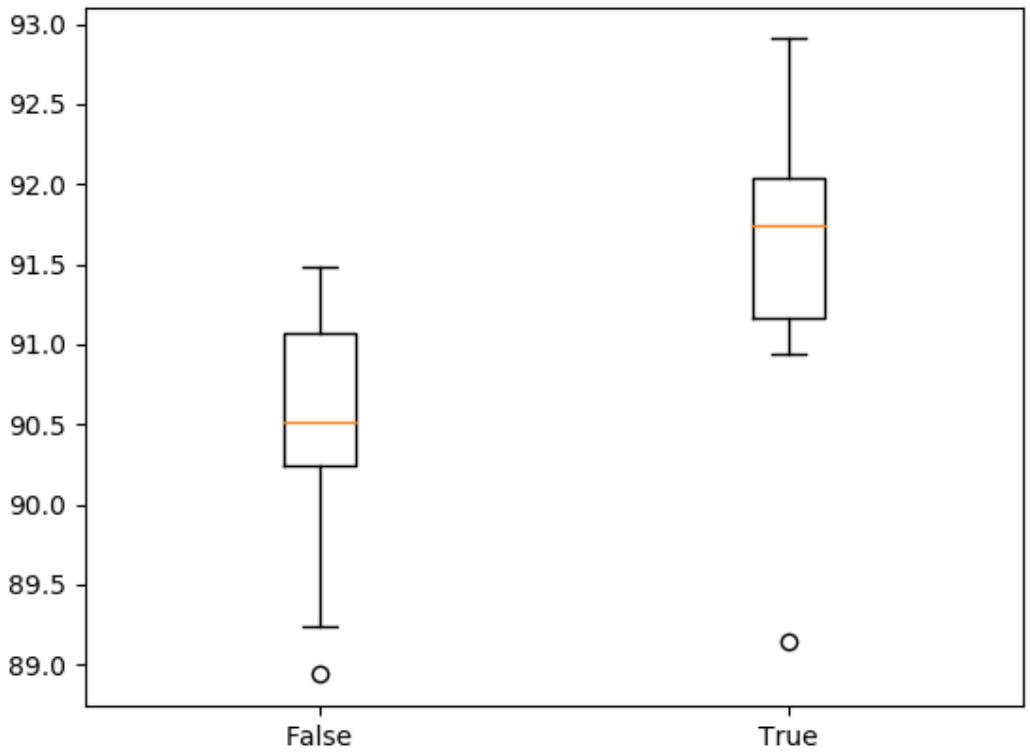
Note: Given the stochastic nature of the algorithm, your specific results may vary.

```
(7352, 128, 9) (7352, 1)
(2947, 128, 9) (2947, 1)
(7352, 128, 9) (7352, 6) (2947, 128, 9) (2947, 6)

>p=False #1: 91.483
>p=False #2: 91.245
>p=False #3: 90.838
>p=False #4: 89.243
>p=False #5: 90.193
>p=False #6: 90.465
>p=False #7: 90.397
>p=False #8: 90.567
>p=False #9: 88.938
>p=False #10: 91.144
>p=True #1: 92.908
>p=True #2: 90.940
>p=True #3: 92.297
>p=True #4: 91.822
>p=True #5: 92.094
>p=True #6: 91.313
>p=True #7: 91.653
>p=True #8: 89.141
>p=True #9: 91.110
>p=True #10: 91.890

[[91.48286392941975, 91.24533423820834, 90.83814048184594, 89.24329826942655,
90.19341703427214, 90.46487953851374, 90.39701391245333, 90.56667797760434,
88.93790295215473, 91.14353579911774], [92.90804207668816, 90.93993892093654,
92.29725144214456, 91.82219205972176, 92.09365456396336, 91.31319986426874,
91.65252799457076, 89.14149983033593, 91.10960298608755, 91.89005768578215]] [False,
True]

Param=False: 90.451% (+/-0.785)
Param=True: 91.517% (+/-0.965)
```

A box and whisker plot of the results is also created.

This allows the two samples of results to be compared in a nonparametric way, showing the median and the middle 50% of each sample.

We can see that the distribution of results with standardization is quite different from the distribution of results without standardization. This is likely a real effect.




Box and whisker plot of 1D CNN with and without standardization

## Number of Filters

Now that we have an experimental framework, we can explore varying other hyperparameters of the model.

An important hyperparameter for the CNN is the number of filter maps. We can experiment with a range of different values, from less to many more than the 64 used in the first model that we developed.

Specifically, we will try the following numbers of feature maps:

```
1 n_params = [8, 16, 32, 64, 128, 256]
```

We can use the same code from the previous section and update the *evaluate_model()*function to use the provided parameter as the number of filters in the Conv1D layers. We can also update the *summarize_results()* function to save the boxplot as *exp_cnn_filters.png*.

The complete code example is listed below.

```
# cnn model with filters
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
	dataframe = read_csv(filepath, header=None, delim_whitespace=True)
	return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
	loaded = list()
	for name in filenames:
		data = load_file(prefix + name)
		loaded.append(data)
	# stack group so that features are the 3rd dimension
	loaded = dstack(loaded)
	return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
	filepath = prefix + group + '/Inertial Signals/'
	# load all 9 files as a single array
	filenames = list()
	# total acceleration
	filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
'total_acc_z_'+group+'.txt']
	# body acceleration
	filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
'body_acc_z_'+group+'.txt']
	# body gyroscope
	filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
'body_gyro_z_'+group+'.txt']
	# load input data
	X = load_group(filenames, filepath)
	# load class output
	y = load_file(prefix + group + '/y_'+group+'.txt')
	return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
	# load all train
	trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
	print(trainX.shape, trainy.shape)
	# load all test
	testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
	print(testX.shape, testy.shape)
	# zero-offset class values
	trainy = trainy - 1
	testy = testy - 1
	# one hot encode y
	trainy = to_categorical(trainy)
	testy = to_categorical(testy)
```

```
        print(trainX.shape, trainy.shape, testX.shape, testy.shape)
        return trainX, trainy, testX, testy

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy, n_filters):
        verbose, epochs, batch_size = 0, 10, 32
        n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
        model = Sequential()
        model.add(Conv1D(filters=n_filters, kernel_size=3, activation='relu',
input_shape=(n_timesteps,n_features)))
        model.add(Conv1D(filters=n_filters, kernel_size=3, activation='relu'))
        model.add(Dropout(0.5))
        model.add(MaxPooling1D(pool_size=2))
        model.add(Flatten())
        model.add(Dense(100, activation='relu'))
        model.add(Dense(n_outputs, activation='softmax'))
        model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
        # fit network
        model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
        # evaluate model
        _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
        return accuracy

# summarize scores
def summarize_results(scores, params):
        print(scores, params)
        # summarize mean and standard deviation
        for i in range(len(scores)):
                m, s = mean(scores[i]), std(scores[i])
                print('Param=%d: %.3f%% (+/-%.3f)' % (params[i], m, s))
        # boxplot of scores
        pyplot.boxplot(scores, labels=params)
        pyplot.savefig('exp_cnn_filters.png')

# run an experiment
def run_experiment(params, repeats=10):
        # load data
        trainX, trainy, testX, testy = load_dataset()
        # test each parameter
        all_scores = list()
        for p in params:
                # repeat experiment
                scores = list()
                for r in range(repeats):
                        score = evaluate_model(trainX, trainy, testX, testy, p)
                        score = score * 100.0
                        print('>p=%d #%d: %.3f' % (p, r+1, score))
                        scores.append(score)
                all_scores.append(scores)
        # summarize results
        summarize_results(all_scores, params)

# run the experiment
n_params = [8, 16, 32, 64, 128, 256]
run_experiment(n_params)
```

Running the example repeats the experiment for each of the specified number of filters.

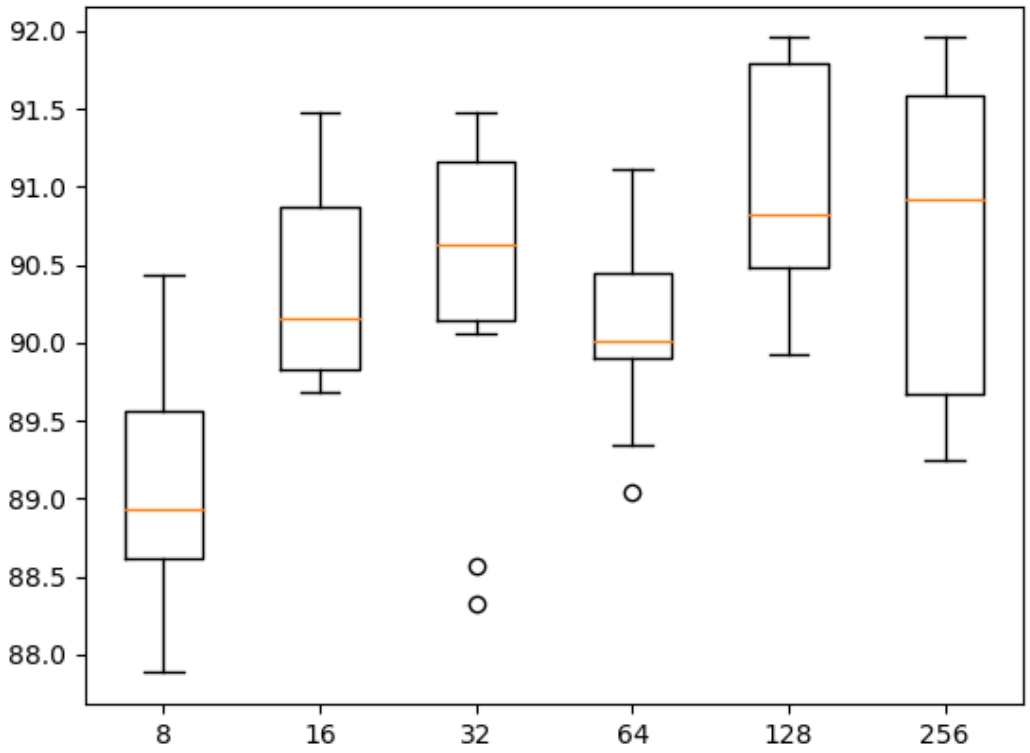
At the end of the run, a summary of the results with each number of filters is presented.

We can see perhaps a trend of increasing average performance with the increase in the number of filter maps. The variance stays pretty constant, and perhaps 128 feature maps might be a good configuration for the network.

```
...
Param=8: 89.148% (+/-0.790)
Param=16: 90.383% (+/-0.613)
Param=32: 90.356% (+/-1.039)
Param=64: 90.098% (+/-0.615)
Param=128: 91.032% (+/-0.702)
Param=256: 90.706% (+/-0.997)
```

A box and whisker plot of the results is also created, allowing the distribution of results with each number of filters to be compared.

From the plot, we can see the trend upward in terms of median classification accuracy (orange line on the box) with the increase in the number of feature maps. We do see a dip at 64 feature maps (the default or baseline in our experiments), which is surprising, and perhaps a plateau in accuracy across 32, 128, and 256 filter maps. Perhaps 32 would be a more stable configuration.




Box and whisker plot of 1D CNN with different numbers of filter maps

## Size of Kernel

The size of the kernel is another important hyperparameter of the 1D CNN to tune.

The kernel size controls the number of time steps consider in each "*read*" of the input sequence, that is then projected onto the feature map (via the convolutional process).
A large kernel size means a less rigorous reading of the data, but may result in a more generalized snapshot of the input.

We can use the same experimental set-up and test a suite of different kernel sizes in addition to the default of three time steps. The full list of values is as follows:

```
n_params = [2, 3, 5, 7, 11]
```

The complete code listing is provided below:

```
# cnn model vary kernel size
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
	dataframe = read_csv(filepath, header=None, delim_whitespace=True)
	return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
	loaded = list()
	for name in filenames:
		data = load_file(prefix + name)
		loaded.append(data)
	# stack group so that features are the 3rd dimension
	loaded = dstack(loaded)
	return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
	filepath = prefix + group + '/Inertial Signals/'
	# load all 9 files as a single array
	filenames = list()
	# total acceleration
	filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
'total_acc_z_'+group+'.txt']
	# body acceleration
	filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
'body_acc_z_'+group+'.txt']
	# body gyroscope
```

```
	filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
'body_gyro_z_'+group+'.txt']
	# load input data
	X = load_group(filenames, filepath)
	# load class output
	y = load_file(prefix + group + '/y_'+group+'.txt')
	return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
	# load all train
	trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
	print(trainX.shape, trainy.shape)
	# load all test
	testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
	print(testX.shape, testy.shape)
	# zero-offset class values
	trainy = trainy - 1
	testy = testy - 1
	# one hot encode y
	trainy = to_categorical(trainy)
	testy = to_categorical(testy)
	print(trainX.shape, trainy.shape, testX.shape, testy.shape)
	return trainX, trainy, testX, testy

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy, n_kernel):
	verbose, epochs, batch_size = 0, 15, 32
	n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
	model = Sequential()
	model.add(Conv1D(filters=64, kernel_size=n_kernel, activation='relu',
input_shape=(n_timesteps,n_features)))
	model.add(Conv1D(filters=64, kernel_size=n_kernel, activation='relu'))
	model.add(Dropout(0.5))
	model.add(MaxPooling1D(pool_size=2))
	model.add(Flatten())
	model.add(Dense(100, activation='relu'))
	model.add(Dense(n_outputs, activation='softmax'))
	model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
	# fit network
	model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
	# evaluate model
	_, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
	return accuracy

# summarize scores
def summarize_results(scores, params):
	print(scores, params)
	# summarize mean and standard deviation
	for i in range(len(scores)):
		m, s = mean(scores[i]), std(scores[i])
		print('Param=%d: %.3f%% (+/-%.3f)' % (params[i], m, s))
	# boxplot of scores
	pyplot.boxplot(scores, labels=params)
	pyplot.savefig('exp_cnn_kernel.png')

# run an experiment
def run_experiment(params, repeats=10):
	# load data
	trainX, trainy, testX, testy = load_dataset()
	# test each parameter
	all_scores = list()
	for p in params:
		# repeat experiment
		scores = list()
```

```
                for r in range(repeats):
                        score = evaluate_model(trainX, trainy, testX, testy, p)
                        score = score * 100.0
                        print('>p=%d #%d: %.3f' % (p, r+1, score))
                        scores.append(score)
                all_scores.append(scores)
        # summarize results
        summarize_results(all_scores, params)

# run the experiment
n_params = [2, 3, 5, 7, 11]
run_experiment(n_params)
```

Running the example tests each kernel size in turn.

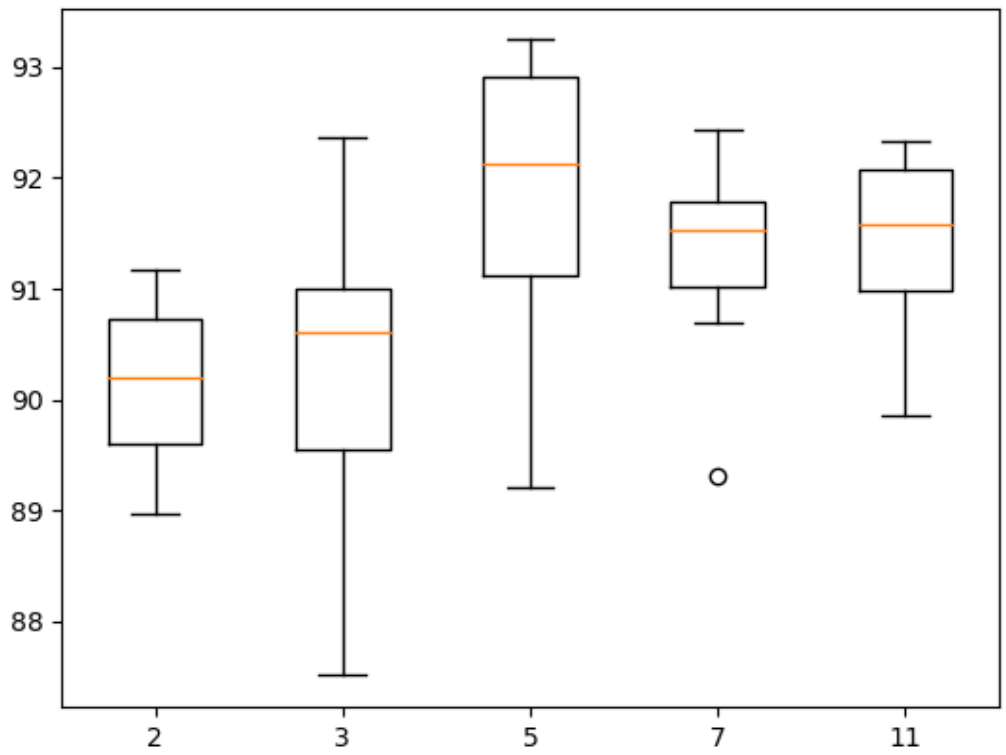
The results are summarized at the end of the run. We can see a general increase in model performance with the increase in kernel size.

The results suggest a kernel size of 5 might be good with a mean skill of about 91.8%, but perhaps a size of 7 or 11 may also be just as good with a smaller standard deviation.

```
...
Param=2: 90.176% (+/-0.724)
Param=3: 90.275% (+/-1.277)
Param=5: 91.853% (+/-1.249)
Param=7: 91.347% (+/-0.852)
Param=11: 91.456% (+/-0.743)
```

A box and whisker plot of the results is also created.

The results suggest that a larger kernel size does appear to result in better accuracy and that perhaps a kernel size of 7 provides a good balance between good performance and low variance.




Box and whisker plot of 1D CNN with different numbers of kernel sizes

This is just the beginning of tuning the model, although we have focused on perhaps the more important elements. It might be interesting to explore combinations of some of the above findings to see if performance can be lifted even further.

It may also be interesting to increase the number of repeats from 10 to 30 or more to see if it results in more stable findings.

# Multi-Headed Convolutional Neural Network

Another popular approach with CNNs is to have a multi-headed model, where each head of the model reads the input time steps using a different sized kernel.

For example, a three-headed model may have three different kernel sizes of 3, 5, 11, allowing the model to read and interpret the sequence data at three different resolutions. The interpretations from all three heads are then concatenated within the model and interpreted by a fully-connected layer before a prediction is made.

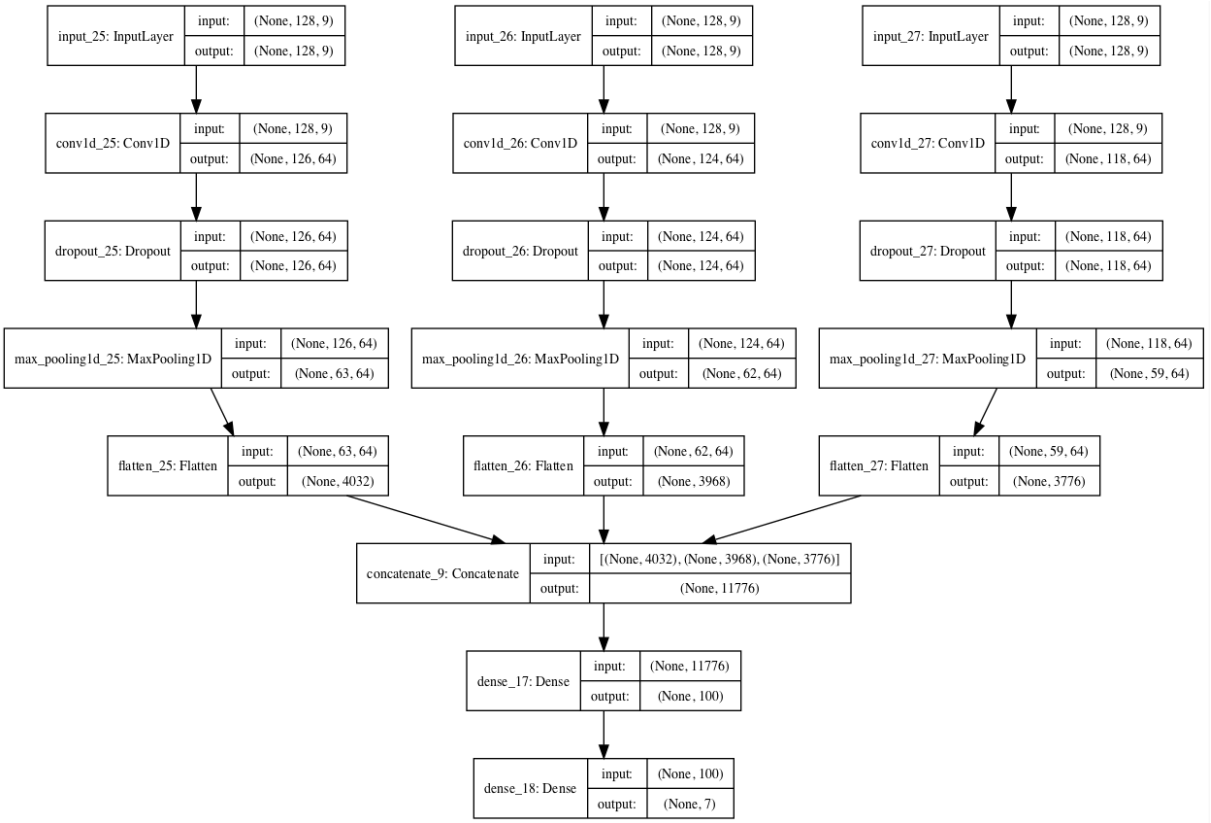
We can implement a multi-headed 1D CNN using the Keras functional API. For a gentle introduction to this API, see the post:

- How to Use the Keras Functional API for Deep Learning

The updated version of the *evaluate_model()* function is listed below that creates a three-headed CNN model.
We can see that each head of the model is the same structure, although the kernel size is varied. The three heads then feed into a single merge layer before being interpreted prior to making a prediction.

```
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
	verbose, epochs, batch_size = 0, 10, 32
	n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
	# head 1
	inputs1 = Input(shape=(n_timesteps,n_features))
	conv1 = Conv1D(filters=64, kernel_size=3, activation='relu')(inputs1)
	drop1 = Dropout(0.5)(conv1)
	pool1 = MaxPooling1D(pool_size=2)(drop1)
	flat1 = Flatten()(pool1)
	# head 2
	inputs2 = Input(shape=(n_timesteps,n_features))
	conv2 = Conv1D(filters=64, kernel_size=5, activation='relu')(inputs2)
	drop2 = Dropout(0.5)(conv2)
	pool2 = MaxPooling1D(pool_size=2)(drop2)
	flat2 = Flatten()(pool2)
	# head 3
	inputs3 = Input(shape=(n_timesteps,n_features))
	conv3 = Conv1D(filters=64, kernel_size=11, activation='relu')(inputs3)
	drop3 = Dropout(0.5)(conv3)
	pool3 = MaxPooling1D(pool_size=2)(drop3)
	flat3 = Flatten()(pool3)
	# merge
	merged = concatenate([flat1, flat2, flat3])
	# interpretation
	dense1 = Dense(100, activation='relu')(merged)
	outputs = Dense(n_outputs, activation='softmax')(dense1)
	model = Model(inputs=[inputs1, inputs2, inputs3], outputs=outputs)
	# save a plot of the model
	plot_model(model, show_shapes=True, to_file='multichannel.png')
	model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
	# fit network
	model.fit([trainX,trainX,trainX], trainy, epochs=epochs, batch_size=batch_size,
verbose=verbose)
	# evaluate model
	_, accuracy = model.evaluate([testX,testX,testX], testy, batch_size=batch_size,
verbose=0)
	return accuracy
```

When the model is created, a plot of the network architecture is created; provided below, it gives a clear idea of how the constructed model fits together.




Plot of the Multi-Headed 1D Convolutional Neural Network

Other aspects of the model could be varied across the heads, such as the number of filters or even the preparation of the data itself.

The complete code example with the multi-headed 1D CNN is listed below.

```
# multi-headed cnn model
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from matplotlib import pyplot
from keras.utils import to_categorical
from keras.utils.vis_utils import plot_model
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.merge import concatenate

# load a single file as a numpy array
def load_file(filepath):
	dataframe = read_csv(filepath, header=None, delim_whitespace=True)
	return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
	loaded = list()
	for name in filenames:
		data = load_file(prefix + name)
```

```
                loaded.append(data)
        # stack group so that features are the 3rd dimension
        loaded = dstack(loaded)
        return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
        filepath = prefix + group + '/Inertial Signals/'
        # load all 9 files as a single array
        filenames = list()
        # total acceleration
        filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
'total_acc_z_'+group+'.txt']
        # body acceleration
        filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
'body_acc_z_'+group+'.txt']
        # body gyroscope
        filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
'body_gyro_z_'+group+'.txt']
        # load input data
        X = load_group(filenames, filepath)
        # load class output
        y = load_file(prefix + group + '/y_'+group+'.txt')
        return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
        # load all train
        trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
        print(trainX.shape, trainy.shape)
        # load all test
        testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
        print(testX.shape, testy.shape)
        # zero-offset class values
        trainy = trainy - 1
        testy = testy - 1
        # one hot encode y
        trainy = to_categorical(trainy)
        testy = to_categorical(testy)
        print(trainX.shape, trainy.shape, testX.shape, testy.shape)
        return trainX, trainy, testX, testy

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
        verbose, epochs, batch_size = 0, 10, 32
        n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
        # head 1
        inputs1 = Input(shape=(n_timesteps,n_features))
        conv1 = Conv1D(filters=64, kernel_size=3, activation='relu')(inputs1)
        drop1 = Dropout(0.5)(conv1)
        pool1 = MaxPooling1D(pool_size=2)(drop1)
        flat1 = Flatten()(pool1)
        # head 2
        inputs2 = Input(shape=(n_timesteps,n_features))
        conv2 = Conv1D(filters=64, kernel_size=5, activation='relu')(inputs2)
        drop2 = Dropout(0.5)(conv2)
        pool2 = MaxPooling1D(pool_size=2)(drop2)
        flat2 = Flatten()(pool2)
        # head 3
        inputs3 = Input(shape=(n_timesteps,n_features))
        conv3 = Conv1D(filters=64, kernel_size=11, activation='relu')(inputs3)
        drop3 = Dropout(0.5)(conv3)
        pool3 = MaxPooling1D(pool_size=2)(drop3)
        flat3 = Flatten()(pool3)
        # merge
        merged = concatenate([flat1, flat2, flat3])
```

```
        # interpretation
        dense1 = Dense(100, activation='relu')(merged)
        outputs = Dense(n_outputs, activation='softmax')(dense1)
        model = Model(inputs=[inputs1, inputs2, inputs3], outputs=outputs)
        # save a plot of the model
        plot_model(model, show_shapes=True, to_file='multichannel.png')
        model.compile(loss='categorical_crossentropy', optimizer='adam',
metrics=['accuracy'])
        # fit network
        model.fit([trainX,trainX,trainX], trainy, epochs=epochs, batch_size=batch_size,
verbose=verbose)
        # evaluate model
        _, accuracy = model.evaluate([testX,testX,testX], testy, batch_size=batch_size,
verbose=0)
        return accuracy

# summarize scores
def summarize_results(scores):
        print(scores)
        m, s = mean(scores), std(scores)
        print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))

# run an experiment
def run_experiment(repeats=10):
        # load data
        trainX, trainy, testX, testy = load_dataset()
        # repeat experiment
        scores = list()
        for r in range(repeats):
                score = evaluate_model(trainX, trainy, testX, testy)
                score = score * 100.0
                print('>#%d: %.3f' % (r+1, score))
                scores.append(score)
        # summarize results
        summarize_results(scores)

# run the experiment
run_experiment()
```

Running the example prints the performance of the model each repeat of the experiment and then summarizes the estimated score as the mean and standard deviation, as we did in the first case with the simple 1D CNN.

We can see that the average performance of the model is about 91.6% classification accuracy with a standard deviation of about 0.8.

This example may be used as the basis for exploring a variety of other models that vary different model hyperparameters and even different data preparation schemes across the input heads.

It would not be an apples-to-apples comparison to compare this result with a single-headed CNN given the relative tripling of the resources in this model. Perhaps an apples-to-apples comparison would be a model with the same architecture and the same number of filters across each input head of the model.

```
>#1: 91.788
>#2: 92.942
>#3: 91.551
>#4: 91.415
>#5: 90.974
>#6: 91.992
>#7: 92.162
>#8: 89.888
>#9: 92.671
>#10: 91.415

[91.78825924669155, 92.94197488971835, 91.55072955548015, 91.41499830335935,
90.97387173396675, 91.99185612487275, 92.16152019002375, 89.88802171700034,
92.67051238547675, 91.41499830335935]

Accuracy: 91.680% (+/-0.823)
```

# Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Date Preparation**. Explore other data preparation schemes such as data normalization and perhaps normalization after standardization.
- **Network Architecture**. Explore other network architectures, such as deeper CNN architectures and deeper fully-connected layers for interpreting the CNN input features.
- **Diagnostics**. Use simple learning curve diagnostics to interpret how the model is learning over the epochs and whether more regularization, different learning rates, or different batch sizes or numbers of epochs may result in a better performing or more stable model.

If you explore any of these extensions, I'd love to know.

# Further Reading

This section provides more resources on the topic if you are looking to go deeper.

## Papers

- A Public Domain Dataset for Human Activity Recognition Using Smartphones, 2013.
- Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine, 2012.

## Articles

- Human Activity Recognition Using Smartphones Data Set, UCI Machine Learning Repository
- Activity recognition, Wikipedia
- Activity Recognition Experiment Using Smartphone Sensors, Video.

# Summary

In this tutorial, you discovered how to develop one-dimensional convolutional neural networks for time series classification on the problem of human activity recognition.

Specifically, you learned:

- How to load and prepare the data for a standard human activity recognition dataset and develop a single 1D CNN model that achieves excellent performance on the raw data.
- How to further tune the performance of the model, including data transformation, filter maps, and kernel sizes.
- How to develop a sophisticated multi-headed one-dimensional convolutional neural network model that provides an ensemble-like result.

# Time series classification with Tensorflow

Time-series data arise in many fields including finance, signal processing, speech recognition and medicine. A standard approach to time-series problems usually requires manual engineering of features which can then be fed into a machine learning algorithm. Engineering of features generally requires some domain knowledge of the discipline where the data has originated from. For example, if one is dealing with signals (i.e. classification of EEG signals), then possible features would involve power spectra at various frequency bands, Hjorth parameters and several other specialized statistical properties.

A similar situation arises in image classification, where manually engineered features (obtained by applying a number of filters) could be used in classification algorithms. However, with the advent of deep learning, it has been shown that convolutional neural networks (CNN) can outperform this strategy. A CNN does not require any manual engineering of features. During training, the CNN learns lots of “filters” with increasing complexity as the layers get deeper, and uses them in a final classifier.

In this blog post, I will discuss the use of deep leaning methods to classify time-series data, without the need to manually engineer features. The example I will consider is the classic Human Activity Recognition (HAR) dataset from the UCI repository. The dataset contains the raw time-series data, as well as a pre-processed one with 561 engineered features. I will compare the performance of typical machine learning algorithms which use engineered features with two deep learning methods (convolutional and recurrent neural networks) and show that deep learning can approach the performance of the former.

I have used Tensorflow for the implementation and training of the models discussed in this post. In the discussion below, code snippets are provided to explain the implementation. For the complete code, please see my Github repository.

# Convolutional Neural Networks (CNN)

The first step is to cast the data in a numpy array with shape (`batch_size, seq_len, n_channels`) where `batch_size` is the number of examples in a batch during training. `seq_len` is the length of the sequence in time-series (128 in our case) and `n_channels` is the number of channels where measurements are made. There are 9 channels in this case, which include 3 different acceleration measurements for each 3

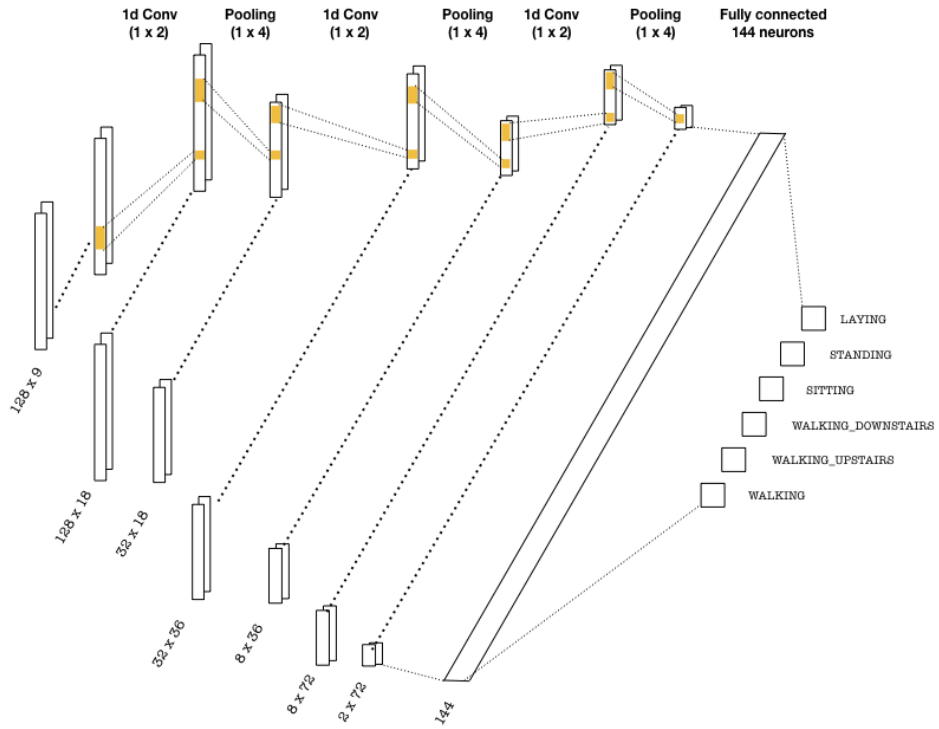
coordinate axes. There are 6 classes of activities where each observation belong to: LAYING, STANDING, SITTING, WALKING_DOWNSTAIRS, WALKING_UPSTAIRS, WALKING.

First, we construct placeholders for the inputs to our computational graph:

```
graph = tf.Graph()

with graph.as_default():
    inputs_ = tf.placeholder(tf.float32, [None, seq_len, n_channels],
        name = 'inputs')
    labels_ = tf.placeholder(tf.float32, [None, n_classes], name = 'labels')
    keep_prob_ = tf.placeholder(tf.float32, name = 'keep')
    learning_rate_ = tf.placeholder(tf.float32, name = 'learning_rate')
```

where inputs_ are input tensors to be fed into the graph whose first dimension is kept at None to allow for variable batch sizes. labels_ are the one-hot encoded labels to be predicted, keep_prob_ is the keep probability used in dropout regularization to prevent overfitting, and learning_rate_ is the learning rate used in Adam optimizer.

The convolutional layers are constructed using one-dimensional kernels that move through the sequence (unlike images where 2d convolutions are used). These kernels act as filters which are being learned during training. As in many CNN architectures, the deeper the layers get, the higher the number of filters become. Each convolution is followed by pooling layers to reduce the sequence length. Below is a simple picture of a possible CNN architecture that can be used:



The convolutional layers that are slightly deeper than the ones depicted above are implemented as follows:

```
with graph.as_default():
    # (batch, 128, 9) -> (batch, 64, 18)
    conv1 = tf.layers.conv1d(inputs=inputs_, filters=18, kernel_size=2, strides=1,
        padding='same', activation = tf.nn.relu)
    max_pool_1 = tf.layers.max_pooling1d(inputs=conv1, pool_size=2, strides=2, pad

    # (batch, 64, 18) -> (batch, 32, 36)
    conv2 = tf.layers.conv1d(inputs=max_pool_1, filters=36, kernel_size=2, strides
    padding='same', activation = tf.nn.relu)
    max_pool_2 = tf.layers.max_pooling1d(inputs=conv2, pool_size=2, strides=2, pad

    # (batch, 32, 36) -> (batch, 16, 72)
    conv3 = tf.layers.conv1d(inputs=max_pool_2, filters=72, kernel_size=2, strides
    padding='same', activation = tf.nn.relu)
    max_pool_3 = tf.layers.max_pooling1d(inputs=conv3, pool_size=2, strides=2, pad
# (batch, 16, 72) -> (batch, 8, 144)
    conv4 =tf.layers.conv1d(inputs=max_pool_3, filters=144, kernel_size=2, stride
    padding='same', activation = tf.nn.relu)
    max_pool_4 = tf.layers.max_pooling1d(inputs=conv4, pool_size=2, strides=2, pad

```

Once the last layer is reached, we need to flatten the tensor and feed it to a classifier with the right number of neurons (144 in the picture, 8×144 in the code snippet). Then, the classifier outputs logits, which are used in two instances:

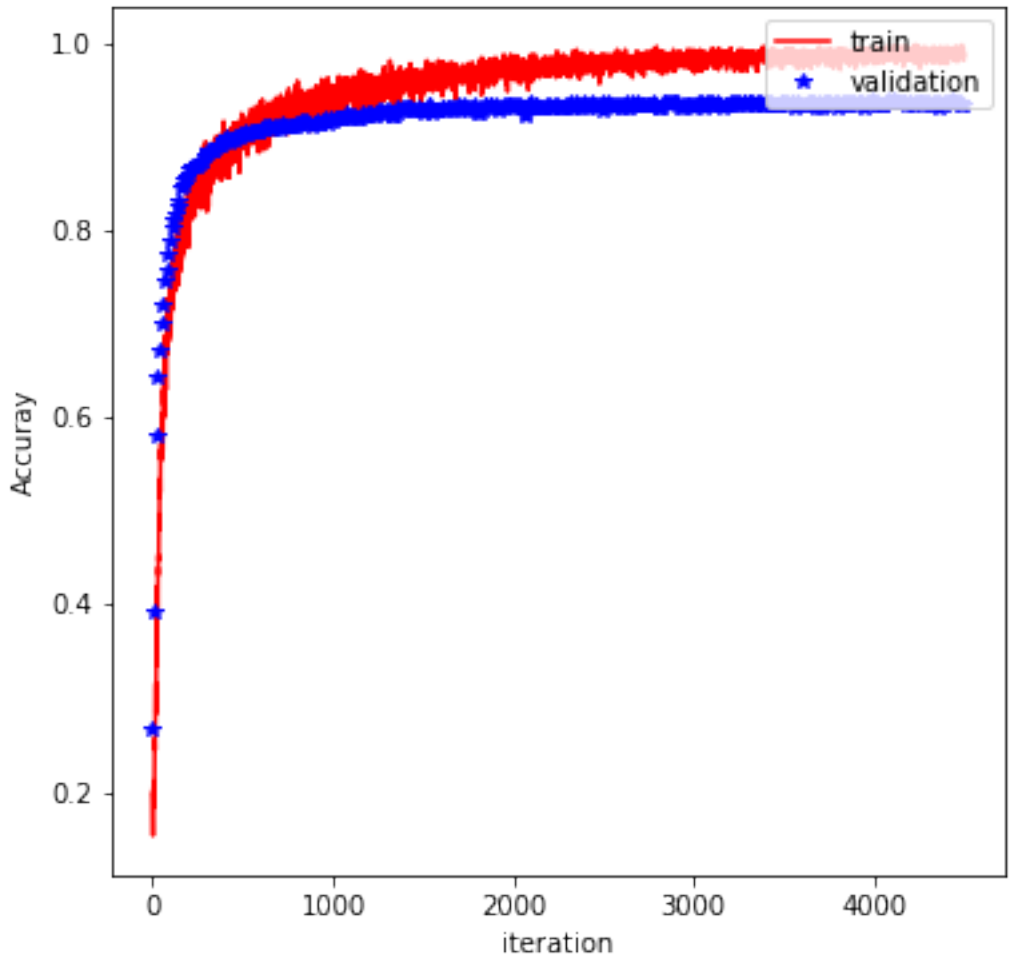
1. Computing the softmax cross entropy, which is a standard loss measure used in multi-class problems.
2. Predicting class labels from the maximum probability as well as the accuracy.

These are implemented as follows:

```
with graph.as_default():
    # Flatten and add dropout
    flat = tf.reshape(max_pool_4, (-1, 8*144))
    flat = tf.nn.dropout(flat, keep_prob=keep_prob_)

    # Predictions
    logits = tf.layers.dense(flat, n_classes)

    # Cost function and optimizer
    cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits,
        labels=labels_))
    optimizer = tf.train.AdamOptimizer(learning_rate_).minimize(cost)

    # Accuracy
    correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(labels_, 1))
    accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32), name='accuracy')

```

The rest of the implementation is pretty typical, and involve feeding the graph with batches of training data and evaluating the performance on a validation set. Finally, the trained model is

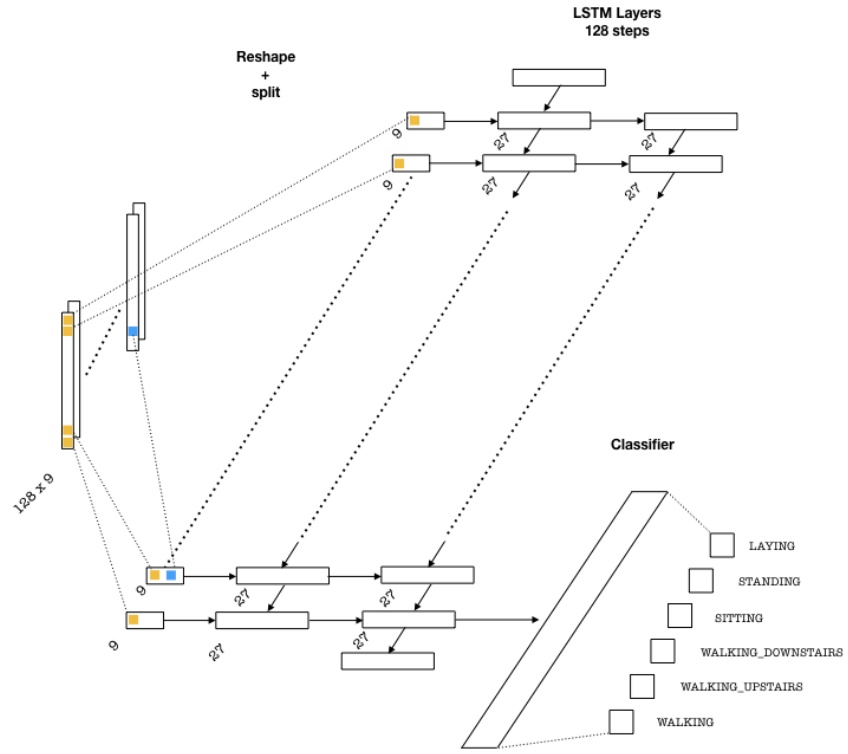
evaluated on the test set. With the above architecture and a batch_size of 600, learning_rate of 0.0001, keep_prob of 0.5, and at 750 epochs, we obtain a test accuracy of 92%. The plot below shows how the training/validation accuracy evolves through the epochs:




# Long-Short-Term Memory Networks (LSTM)

LSTMs are quite popular in dealing with text based data, and has been quite successful in sentiment analysis, language translation and text generation. Since this problem also involves a sequence of similar sorts, an LSTM is a great candidate to be tried.

Below is an example architecture which can be used in our problem:




To feed the data into the network, we need to split our array into 128 pieces (one for each entry of the sequence that goes into an LSTM cell) each of shape (batch_size, n_channels). Then, a single layer of neurons will transform these inputs to be fed into the LSTM cells, each with the dimension lstm_size. This size parameter is chosen to be larger than the number of channels. This is in a way similar to embedding layers in text applications where words are embedded as vectors from a given vocabulary. Then, one needs to pick the number of LSTM layers (lstm_layers), which I have set to 2.

For the implementation, the placeholders are the same as above. The below code snippet implements the LSTM layers:

```
    with graph.as_default():
        # Construct the LSTM inputs and LSTM cells
        lstm_in = tf.transpose(inputs_, [1,0,2]) # reshape into (seq_len, N, channels)
        lstm_in = tf.reshape(lstm_in, [-1, n_channels]) # Now (seq_len*N, n_channels)

        # To cells
        lstm_in = tf.layers.dense(lstm_in, lstm_size, activation=None)

        # Open up the tensor into a list of seq_len pieces
        lstm_in = tf.split(lstm_in, seq_len, 0)

        # Add LSTM layers
        lstm = tf.contrib.rnn.BasicLSTMCell(lstm_size)
        drop = tf.contrib.rnn.DropoutWrapper(lstm, output_keep_prob=keep_prob_)
        cell = tf.contrib.rnn.MultiRNNCell([drop] * lstm_layers)
        initial_state = cell.zero_state(batch_size, tf.float32)
```

```
15
16
```

There is an important technical detail in the above snippet. I reshaped the array from (batch_size, seq_len, n_channels) to (seq_len, batch_size, n_channels) first, so that tf.split would properly split the data (by the zeroth index) into a list of (batch_size, lstm_size) arrays at each step. The rest is pretty standard for LSTM implementations, involving construction of layers (including dropout for regularization) and then an initial state.

The next step is to implement the forward pass through the network and the cost function. One important technical aspect is that I included gradient clipping since it improves training by preventing exploding gradients during back propagation. Here is what the code looks like

```
1
2     with graph.as_default():
3         outputs, final_state = tf.contrib.rnn.static_rnn(cell, lstm_in, dtype=tf.float
4             initial_state = initial_state)
5
6         # We only need the last output tensor to pass into a classifier
7         logits = tf.layers.dense(outputs[-1], n_classes, name='logits')
8
9         # Cost function and optimizer
10        cost = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(logits=logits, l
11
12        # Grad clipping
13        train_op = tf.train.AdamOptimizer(learning_rate_)
14
15        gradients = train_op.compute_gradients(cost)
16        capped_gradients = [(tf.clip_by_value(grad, -1., 1.), var) for grad, var in gra
17        optimizer = train_op.apply_gradients(capped_gradients)
18
19        # Accuracy
20        correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(labels_, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32), name='accuracy')
```

Notice that only the last last member of the sequence at the top of the LSTM outputs are used, since we are trying to predict one number per sequence (the class probability). The rest is similar to CNNs and we just need to feed the data into the graph to train.
With lstm_size=27, lstm_layers=2, batch_size=600, learning_rate=0.0005, and keep_prob=0.5, I obtained around 8**5%** accuracy on the test set. This is worse than the CNN result, but still quite good. It is possible that better choices of these hyperparameters would lead to improved results.

# Comparison with engineered features

Previously, I have tested a few machine learning methods on this problem using the 561 pre-engineered features. One of the best performing models was a gradient booster (tree or linear), which results in an accuracy of **%96** (you can read more about it from this notebook ).

# Final Words

In this blog post, I have illustrated the use of CNNs and LSTMs for time-series classification and shown that a deep architecture can approach the performance of a model trained on pre-engineered features. Deep learning models “engineer” their own features during training. This

is highly desirable, since one does not need to have domain expertise from where the data has originated from, to be able to train an accurate model. With more data, and further hyperparameter tuning, it is possible that deep learning methods can surpass the models trained on pre-engineered features.

The sequence we used in this post was fairly small (128 steps). One may wonder what would happen if the number of steps were much larger and worry about the trainability of these architectures I discussed. One possible architecture would involve a combination of LSTM and CNN, which could work better for larger sequences (i.e. > 1000, which is problematic for LSTMs). In this case, several convolutions with pooling can effectively reduce the number of steps in the first few layers and the resulting shorter sequences can be fed into LSTM layers. An example of such an architecture has recently been used in atrial fibrillation detection from mobile device recordings.

Note: In the previous version of this post, the reported test accuracies were higher. This was due to a bug in the code during reading the train/test sets.

# Links

1. Also see G. Chevalier's repo and A. Saeed's blog where I got lots of inspiration.
2. Repository containing LSTM and CNN codes
3. Repository containing several models trained on 561 manually generated features (in R)
4. A notebook on the analysis of the data and predictions with gradient boosters (in R).