EECS 738 Lab 6

Step 0: Import required packages

In [8]:
```
from __future__ import print_function

import os
import sys
import csv

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import *
from tensorflow.keras.activations import relu
from tensorflow.keras.preprocessing import sequence
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras import backend as K

from tensorflow.keras.datasets import imdb

from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"

print(os.getcwd())
print("Modules imported \n")
print("Files in current directory:")
from subprocess import check_output
#print(check_output(["ls", "../data"]).decode("utf8")) #check the files available in the directory

print("Packages Loaded")
print('The Tensorflow version is {}.'.format(tf.__version__))
print('The Keras version is {}.'.format(keras.__version__))
print('The Pandas version is {}.'.format(pd.__version__))
print('The Numpy version is {}.'.format(np.__version__))
print(np.__file__)
```

```
C:\Users\pmspr\Documents\HS\MS\Sem 2\EECS 738\Lab\6
Modules imported

Files in current directory:
Packages Loaded
The Tensorflow version is 2.1.0.
The Keras version is 2.2.4-tf.
The Pandas version is 1.0.3.
The Numpy version is 1.18.1.
C:\ProgramData\Anaconda2\envs\TFK35\lib\site-packages\numpy\__init__.py
```

We will be working with the IMDB data from keras for this lab. The data is already enocded so I wanted to show an example of what text data looks like before it gets encoded. Below is the stanford sentiment treebank data broken up into its data and the sentiment values.

I wanted to use a more complex dataset for this but the time constraints due to COV-19 have made that difficult.

So now lets get into the data we are working with today. In the last couple of labs we used CNNs and ResNets a lot. This time we are going to compare CNNs with LSTMs for the purpose of classifying text. The data is setup so that a '0 label' is a negative review and a '1 label' is a postive review.

We want to create machine learning models to automatically detect whether or not a review is positive. This has wide applications for both industry and research and has been extensively researched since 2014.

Step 1: Load Imdb dataset

In [9]:
```
# Lets load our data. We will limit the number of words to 5,000 as that is how the data is setup.
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=2500)

print("train_data ", x_train.shape)
print("train_labels ", y_train.shape)
print("_"*100)
print("test_data ", x_test.shape)
print("test_labels ", y_test.shape)
print("_"*100)
print("Maximum value of a word index ")
print(max([max(sequence) for sequence in x_train]))
print("Maximum length num words of review in train ")
print(max([len(sequence) for sequence in x_train]))

# See an actual review in words
# Reverse from integers to words using the DICTIONARY (given by keras...need to do nothing to create it)

word_index = imdb.get_word_index()

reverse_word_index = dict(
[(value, key) for (key, value) in word_index.items()])

decoded_review = ' '.join(
[reverse_word_index.get(i - 0, '?') for i in x_train[1]])

#print(x_train[1])
print(decoded_review)
```

```
train_data  (25000,)
train_labels  (25000,)
```

```
test_data  (25000,)
test_labels  (25000,)
```

```
Maximum value of a word index
2499
Maximum length num words of review in train
2494
the thought solid thought and do making to is spot and and while he of jack in where picked as getting on was
did hands fact characters to always life and not as me can't in at are br of sure your way of little it stron
gly random to view of love it so and of guy it used producer of where it of here and film of outside to don't
all unique some like of direction it if out her imagination below keep of queen he and to makes this and and
of solid it thought begins br and and budget and though ok and and for ever better were and and for budget lo
ok and any to of making it out and follows for effects show to show cast this family us scenes more it and ma
king and to and finds tv tend to of and these thing wants but and an and cult as it is video do you david see
scenery it in few those are of ship for with of wild to one is very work dark they don't do dvd with those th
em
```

Step 2: Pad train and test data.

In [10]:

```
# We pad the data because not all sentences in our data are the same length. We want to use a number that is
 larger than our largest data. Here I will choose 400.
x_train = sequence.pad_sequences(x_train, maxlen=400)
x_test = sequence.pad_sequences(x_test, maxlen=400)
```

Step 3: Create a 1D CNN for baseline.

In [13]:
```
# Lets start with a very simple 1D CNN model. We will use this as our baseline for everything else in this la
b.
model = Sequential()

# This embedding is a trainable parameter. We aren't using GLoVE for this model.
model.add(keras.layers.Embedding(2500,50,input_length=400))
model.add(keras.layers.Dropout(0.2))

# There isn't much of a difference with how 1D and 2D CNNs work. They still use filters and scan the data.
# we will use a similar model as our 2D CNN with the adition of an embedding layer at the beginning.
model.add(keras.layers.Conv1D(64,3,padding='valid',activation='relu',strides=1))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Conv1D(64,3,padding='valid',activation='relu',strides=1))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.GlobalMaxPooling1D())

model.add(keras.layers.Dense(512))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Activation('relu'))

# We will use a sigmoid and a 1 neuron dense output since our data is binary.
model.add(keras.layers.Dense(1))
model.add(keras.layers.Activation('sigmoid'))

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

history = model.fit(x_train, y_train,
              batch_size=128,
              epochs=10,
              validation_split=0.1)

history = pd.DataFrame(history.history)
history.plot(figsize=(10,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.xlabel('Epochs')
plt.title('Base CNN imdb')
plt.show()

#After training the model, evaluate the test set
```

```
model.evaluate(x_test,y_test)

#Print the summary of the model
model.summary()
```

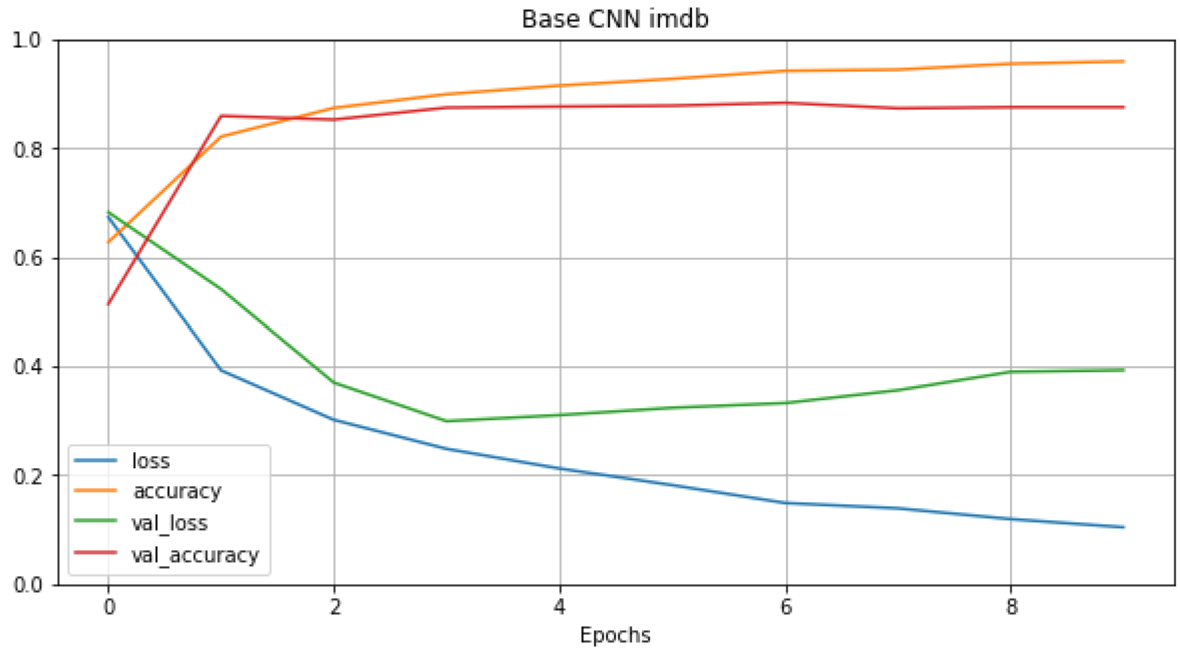
```
Train on 22500 samples, validate on 2500 samples
Epoch 1/10
22500/22500 [==============================] - 27s 1ms/sample - loss: 0.6734 - accuracy: 0.6275 - val_loss:
0.6825 - val_accuracy: 0.5140
Epoch 2/10
22500/22500 [==============================] - 27s 1ms/sample - loss: 0.3927 - accuracy: 0.8216 - val_loss:
0.5414 - val_accuracy: 0.8596
Epoch 3/10
22500/22500 [==============================] - 28s 1ms/sample - loss: 0.3018 - accuracy: 0.8743 - val_loss:
0.3702 - val_accuracy: 0.8528
Epoch 4/10
22500/22500 [==============================] - 27s 1ms/sample - loss: 0.2485 - accuracy: 0.8996 - val_loss:
0.2993 - val_accuracy: 0.8748
Epoch 5/10
22500/22500 [==============================] - 29s 1ms/sample - loss: 0.2124 - accuracy: 0.9153 - val_loss:
0.3105 - val_accuracy: 0.8768
Epoch 6/10
22500/22500 [==============================] - 28s 1ms/sample - loss: 0.1817 - accuracy: 0.9278 - val_loss:
0.3240 - val_accuracy: 0.8784
Epoch 7/10
22500/22500 [==============================] - 28s 1ms/sample - loss: 0.1489 - accuracy: 0.9424 - val_loss:
0.3326 - val_accuracy: 0.8836
Epoch 8/10
22500/22500 [==============================] - 28s 1ms/sample - loss: 0.1391 - accuracy: 0.9447 - val_loss:
0.3562 - val_accuracy: 0.8736
Epoch 9/10
22500/22500 [==============================] - 29s 1ms/sample - loss: 0.1194 - accuracy: 0.9555 - val_loss:
0.3900 - val_accuracy: 0.8756
Epoch 10/10
22500/22500 [==============================] - 29s 1ms/sample - loss: 0.1044 - accuracy: 0.9599 - val_loss:
0.3931 - val_accuracy: 0.8756
```

```
Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x1f4210499b0>
```

```
Out[13]: (0, 1)
```

```
Out[13]: Text(0.5, 0, 'Epochs')
```

```
Out[13]: Text(0.5, 1.0, 'Base CNN imdb')
```




```
25000/25000 [==============================] - 5s 212us/sample - loss: 0.3916 - accuracy: 0.8740
```

Out[13]:
```
[0.3916378186559677, 0.87404]
```

```
Model: "sequential_5"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_5 (Embedding)      (None, 400, 50)           125000    
_________________________________________________________________
dropout_10 (Dropout)         (None, 400, 50)           0         
_________________________________________________________________
conv1d_10 (Conv1D)           (None, 398, 64)           9664      
_________________________________________________________________
batch_normalization_10 (Batc (None, 398, 64)           256       
_________________________________________________________________
conv1d_11 (Conv1D)           (None, 396, 64)           12352     
_________________________________________________________________
batch_normalization_11 (Batc (None, 396, 64)           256       
_________________________________________________________________
global_max_pooling1d_5 (Glob (None, 64)                0         
_________________________________________________________________
dense_10 (Dense)             (None, 512)               33280     
_________________________________________________________________
dropout_11 (Dropout)         (None, 512)               0         
_________________________________________________________________
activation_10 (Activation)   (None, 512)               0         
_________________________________________________________________
dense_11 (Dense)             (None, 1)                 513       
_________________________________________________________________
activation_11 (Activation)   (None, 1)                 0         
=================================================================
Total params: 181,321
Trainable params: 181,065
Non-trainable params: 256
_________________________________________________________________
```

How well is this model doing? Is it overfitting? If so how could you fix this since we are already applying BatchNorm and dropout?

Step 4: Analysis

- I have reduced the input dataset to 2500, as my my CUP is making Jupyter notebook kernel err out.
- With 2500 dataset, for 10 epochs, we do not see much overfitting. There is no much differece between training and validation accuracies. Training accuracy - 0.95 and Validation accuracy - 0.87

Step 5: Filters change from 64 to 250

In [14]:
```
# Now we will make the network more complex by adding more filters to the data. How did this affect training?
# Lets start with a very simple 1D CNN model. We will use this as our baseline for everything else in this la
b.
model = Sequential()

# This embedding is a trainable parameter. We aren't using GLoVE for this model.
model.add(keras.layers.Embedding(2500,50,input_length=400))
model.add(keras.layers.Dropout(0.2))

# There isn't much of a difference with how 1D and 2D CNNs work. They still use filters and scan the data.
# we will use a similar model as our 2D CNN with the adition of an embedding layer at the beginning.
model.add(keras.layers.Conv1D(250,3,padding='valid',activation='relu',strides=1))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Conv1D(250,3,padding='valid',activation='relu',strides=1))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.GlobalMaxPooling1D())

model.add(keras.layers.Dense(512))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Activation('relu'))

# We will use a sigmoid and a 1 neuron dense output since our data is binary.
model.add(keras.layers.Dense(1))
model.add(keras.layers.Activation('sigmoid'))

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

history = model.fit(x_train, y_train,
              batch_size=128,
              epochs=10,
              validation_split=0.1)

history = pd.DataFrame(history.history)
history.plot(figsize=(10,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.xlabel('Epochs')
plt.title('Base CNN imdb')
plt.show()
```

```
#After training the model, evaluate the test set
model.evaluate(x_test,y_test)

#Print the summary of the model
model.summary()
```

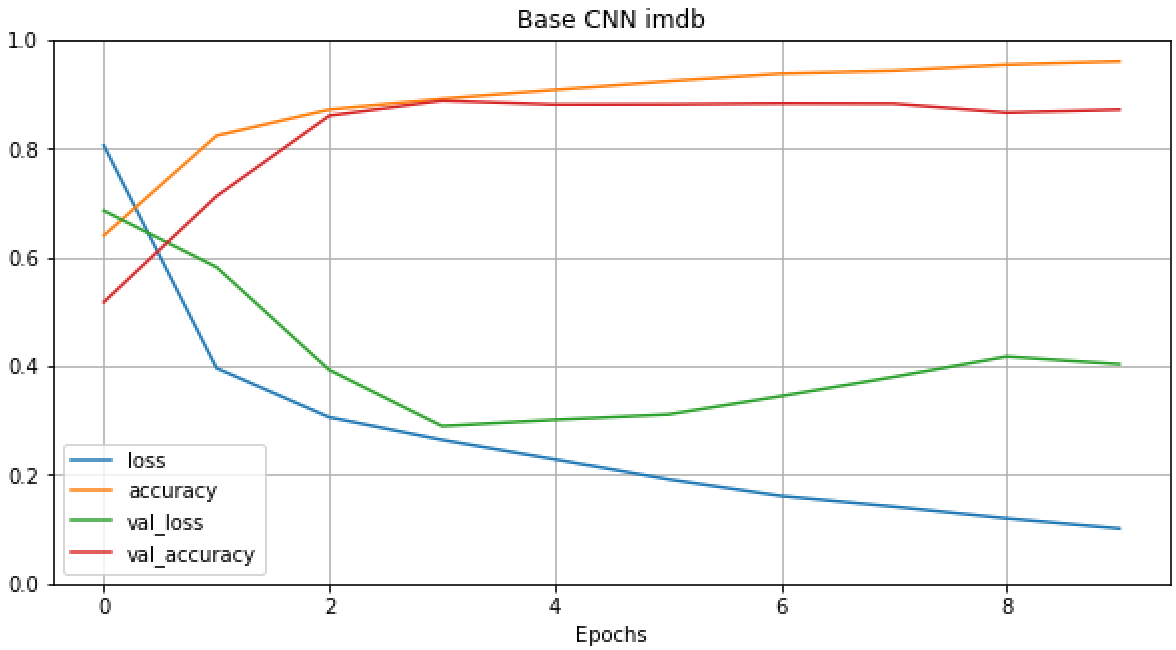
```
Train on 22500 samples, validate on 2500 samples
Epoch 1/10
22500/22500 [==============================] - 133s 6ms/sample - loss: 0.8059 - accuracy: 0.6405 - val_loss:
0.6859 - val_accuracy: 0.5180
Epoch 2/10
22500/22500 [==============================] - 125s 6ms/sample - loss: 0.3962 - accuracy: 0.8241 - val_loss:
0.5820 - val_accuracy: 0.7128
Epoch 3/10
22500/22500 [==============================] - 136s 6ms/sample - loss: 0.3061 - accuracy: 0.8721 - val_loss:
0.3929 - val_accuracy: 0.8608
Epoch 4/10
22500/22500 [==============================] - 141s 6ms/sample - loss: 0.2644 - accuracy: 0.8919 - val_loss:
0.2898 - val_accuracy: 0.8884
Epoch 5/10
22500/22500 [==============================] - 143s 6ms/sample - loss: 0.2286 - accuracy: 0.9083 - val_loss:
0.3013 - val_accuracy: 0.8812
Epoch 6/10
22500/22500 [==============================] - 119s 5ms/sample - loss: 0.1917 - accuracy: 0.9243 - val_loss:
0.3113 - val_accuracy: 0.8816
Epoch 7/10
22500/22500 [==============================] - 167s 7ms/sample - loss: 0.1612 - accuracy: 0.9382 - val_loss:
0.3450 - val_accuracy: 0.8832
Epoch 8/10
22500/22500 [==============================] - 145s 6ms/sample - loss: 0.1415 - accuracy: 0.9436 - val_loss:
0.3801 - val_accuracy: 0.8828
Epoch 9/10
22500/22500 [==============================] - 120s 5ms/sample - loss: 0.1200 - accuracy: 0.9548 - val_loss:
0.4178 - val_accuracy: 0.8664
Epoch 10/10
22500/22500 [==============================] - 122s 5ms/sample - loss: 0.1014 - accuracy: 0.9608 - val_loss:
0.4038 - val_accuracy: 0.8720
```

```
Out[14]: <matplotlib.axes._subplots.AxesSubplot at 0x1f423ff3668>

Out[14]: (0, 1)

Out[14]: Text(0.5, 0, 'Epochs')

Out[14]: Text(0.5, 1.0, 'Base CNN imdb')
```




```
25000/25000 [==============================] - 23s 909us/sample - loss: 0.3618 - accuracy: 0.8803
```

Out[14]:
```
[0.3617532423210144, 0.88028]
```

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_6 (Embedding)      (None, 400, 50)           125000    
_________________________________________________________________
dropout_12 (Dropout)         (None, 400, 50)           0         
_________________________________________________________________
conv1d_12 (Conv1D)           (None, 398, 250)          37750     
_________________________________________________________________
batch_normalization_12 (Batc (None, 398, 250)          1000      
_________________________________________________________________
conv1d_13 (Conv1D)           (None, 396, 250)          187750    
_________________________________________________________________
batch_normalization_13 (Batc (None, 396, 250)          1000      
_________________________________________________________________
global_max_pooling1d_6 (Glob (None, 250)               0         
_________________________________________________________________
dense_12 (Dense)             (None, 512)               128512    
_________________________________________________________________
dropout_13 (Dropout)         (None, 512)               0         
_________________________________________________________________
activation_12 (Activation)   (None, 512)               0         
_________________________________________________________________
dense_13 (Dense)             (None, 1)                 513       
_________________________________________________________________
activation_13 (Activation)   (None, 1)                 0         
=================================================================
Total params: 481,525
Trainable params: 480,525
Non-trainable params: 1,000
_________________________________________________________________
```

Analysis:

- Changing filters from 64 to 250, did not contribute much in improving the accuracies. Training and validation accuracies remained same for the dataset of 2500. On the contrary, training speed is increased.

Step 6: Add pair of CNN and normalization layers

In [17]:
```
# Now lets add more CNN and BatchNorm layers to the network. Did this have the same affect as 2D CNNs from la
b 5?
# Lets start with a very simple 1D CNN model. We will use this as our baseline for everything else in this la
b.
model = Sequential()

# This embedding is a trainable parameter. We aren't using GLoVE for this model.
model.add(keras.layers.Embedding(2500,50,input_length=400))
model.add(keras.layers.Dropout(0.2))

# There isn't much of a difference with how 1D and 2D CNNs work. They still use filters and scan the data.
# we will use a similar model as our 2D CNN with the adition of an embedding layer at the beginning.
model.add(keras.layers.Conv1D(64,3,padding='valid',activation='relu',strides=1))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Conv1D(64,3,padding='valid',activation='relu',strides=1))
model.add(keras.layers.BatchNormalization())

model.add(keras.layers.Dense(512))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Activation('relu'))

model.add(keras.layers.Conv1D(64,3,padding='valid',activation='relu',strides=1))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.Conv1D(64,3,padding='valid',activation='relu',strides=1))
model.add(keras.layers.BatchNormalization())
model.add(keras.layers.GlobalMaxPooling1D())

model.add(keras.layers.Dense(512))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Activation('relu'))

# We will use a sigmoid and a 1 neuron dense output since our data is binary.
model.add(keras.layers.Dense(1))
model.add(keras.layers.Activation('sigmoid'))

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

history = model.fit(x_train, y_train,
              batch_size=128,
              epochs=10,
```

```
                  validation_split=0.1)

history = pd.DataFrame(history.history)
history.plot(figsize=(10,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.xlabel('Epochs')
plt.title('Base CNN imdb')
plt.show()

#After training the model, evaluate the test set
model.evaluate(x_test,y_test)

#Print the summary of the model
model.summary()
```

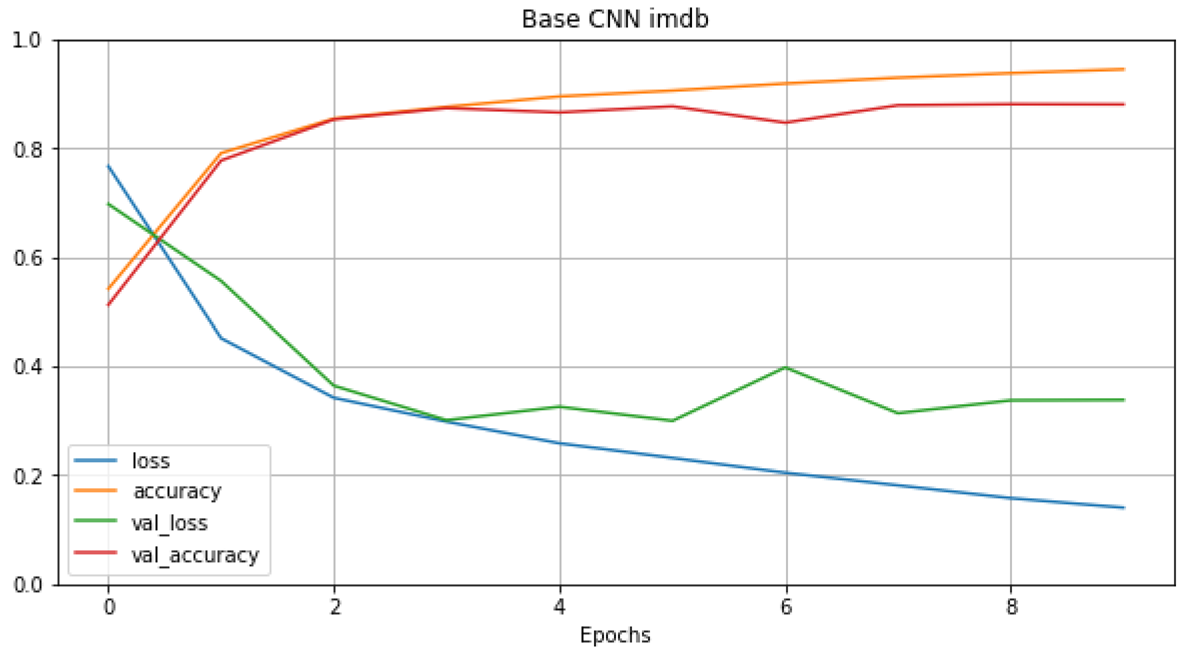
```
Train on 22500 samples, validate on 2500 samples
Epoch 1/10
22500/22500 [==============================] - 134s 6ms/sample - loss: 0.7671 - accuracy: 0.5418 - val_loss:
0.6976 - val_accuracy: 0.5124
Epoch 2/10
22500/22500 [==============================] - 135s 6ms/sample - loss: 0.4518 - accuracy: 0.7912 - val_loss:
0.5563 - val_accuracy: 0.7776
Epoch 3/10
22500/22500 [==============================] - 121s 5ms/sample - loss: 0.3424 - accuracy: 0.8546 - val_loss:
0.3646 - val_accuracy: 0.8532
Epoch 4/10
22500/22500 [==============================] - 110s 5ms/sample - loss: 0.2986 - accuracy: 0.8762 - val_loss:
0.3012 - val_accuracy: 0.8740
Epoch 5/10
22500/22500 [==============================] - 114s 5ms/sample - loss: 0.2588 - accuracy: 0.8954 - val_loss:
0.3260 - val_accuracy: 0.8660
Epoch 6/10
22500/22500 [==============================] - 115s 5ms/sample - loss: 0.2318 - accuracy: 0.9061 - val_loss:
0.3000 - val_accuracy: 0.8772
Epoch 7/10
22500/22500 [==============================] - 118s 5ms/sample - loss: 0.2046 - accuracy: 0.9191 - val_loss:
0.3982 - val_accuracy: 0.8472
Epoch 8/10
22500/22500 [==============================] - 178s 8ms/sample - loss: 0.1814 - accuracy: 0.9299 - val_loss:
0.3140 - val_accuracy: 0.8792
Epoch 9/10
22500/22500 [==============================] - 147s 7ms/sample - loss: 0.1579 - accuracy: 0.9382 - val_loss:
0.3377 - val_accuracy: 0.8812
Epoch 10/10
22500/22500 [==============================] - 160s 7ms/sample - loss: 0.1404 - accuracy: 0.9453 - val_loss:
0.3385 - val_accuracy: 0.8808
```

```
Out[17]: <matplotlib.axes._subplots.AxesSubplot at 0x1f422bd2438>

Out[17]: (0, 1)

Out[17]: Text(0.5, 0, 'Epochs')

Out[17]: Text(0.5, 1.0, 'Base CNN imdb')
```




```
25000/25000 [==============================] - 28s 1ms/sample - loss: 0.3556 - accuracy: 0.8720
```

Out[17]:
```
[0.3555895343375206, 0.87204]
```

```
Model: "sequential_9"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_9 (Embedding)      (None, 400, 50)           125000    
_________________________________________________________________
dropout_17 (Dropout)         (None, 400, 50)           0         
_________________________________________________________________
conv1d_20 (Conv1D)           (None, 398, 64)           9664      
_________________________________________________________________
batch_normalization_18 (Batc (None, 398, 64)           256       
_________________________________________________________________
conv1d_21 (Conv1D)           (None, 396, 64)           12352     
_________________________________________________________________
batch_normalization_19 (Batc (None, 396, 64)           256       
_________________________________________________________________
dense_15 (Dense)             (None, 396, 512)          33280     
_________________________________________________________________
dropout_18 (Dropout)         (None, 396, 512)          0         
_________________________________________________________________
activation_15 (Activation)   (None, 396, 512)          0         
_________________________________________________________________
conv1d_22 (Conv1D)           (None, 394, 64)           98368     
_________________________________________________________________
batch_normalization_20 (Batc (None, 394, 64)           256       
_________________________________________________________________
conv1d_23 (Conv1D)           (None, 392, 64)           12352     
_________________________________________________________________
batch_normalization_21 (Batc (None, 392, 64)           256       
_________________________________________________________________
global_max_pooling1d_9 (Glob (None, 64)                0         
_________________________________________________________________
dense_16 (Dense)             (None, 512)               33280     
_________________________________________________________________
dropout_19 (Dropout)         (None, 512)               0         
_________________________________________________________________
activation_16 (Activation)   (None, 512)               0         
_________________________________________________________________
dense_17 (Dense)             (None, 1)                 513       
_________________________________________________________________
activation_17 (Activation)   (None, 1)                 0         
=================================================================
Total params: 325,833
```

```
Trainable params: 325,321
Non-trainable params: 512
________________________________________________________________
```

Do 1D CNNs and 2D CNNs behave the same from the changes we are making?

Now lets look at some LSTMs. LSTMs and RNNs in general were the racehorse of deep learning from 2014-2016. Now they have drastically fallen off of favor in the DL community. The questions we want to answer in this lab are: Why do you think this is? Do you think it was a mistake to stray away from RNNs? What changes do you think we could make to make them better or should we just drop them all together?

The resources to learn more about this debate can be found here: https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0 (https://towardsdatascience.com/the-fall-of-rnn-lstm-2d1594c74ce0)

and here: https://towardsdatascience.com/memory-attention-sequences-37456d271992 (https://towardsdatascience.com/memory-attention-sequences-37456d271992)

and here: https://towardsdatascience.com/visual-attention-model-in-deep-learning-708813c2912c (https://towardsdatascience.com/visual-attention-model-in-deep-learning-708813c2912c)

These are optional readings but they serve to give you a firm foundation on the knowledge of current deep learning thought. Feel free to answer the above questions after we train our LSTMs.

If you don't know anything about RNNs read this: http://karpathy.github.io/2015/05/21/rnn-effectiveness/ (http://karpathy.github.io/2015/05/21/rnn-effectiveness/)

Step 7: LSTM model

In [20]:
```
# Now we will make our LSTMs. We will use a smaller batch size as they take longer to train.
# We use the same embedding layers as we did for our CNNs.
model = keras.models.Sequential()
model.add(keras.layers.Embedding(5000,50,input_length=400))

# Here we will add in our LSTM layers. They should be directly after the embedding layer.
model.add(keras.layers.LSTM(128, return_sequences=True))
model.add(keras.layers.LSTM(128))

# Now we will cast the LSTM output to a dense layer to sort it. If you haven't noticed, thick dense layers at the end of networks are how every model 'collects its thoughts'.
model.add(keras.layers.Dense(512, activation='relu'))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(1, activation='sigmoid'))

model.compile('Nadam', 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, y_train,
              batch_size=128,
              epochs=5,
              validation_data=[x_test, y_test])

history = pd.DataFrame(history.history)
history.plot(figsize=(10,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.xlabel('Epochs')
plt.title('Base CNN imdb')
plt.show()

#After training the model, evaluate the test set
model.evaluate(x_test,y_test)

#Print the summary of the model
model.summary()
```

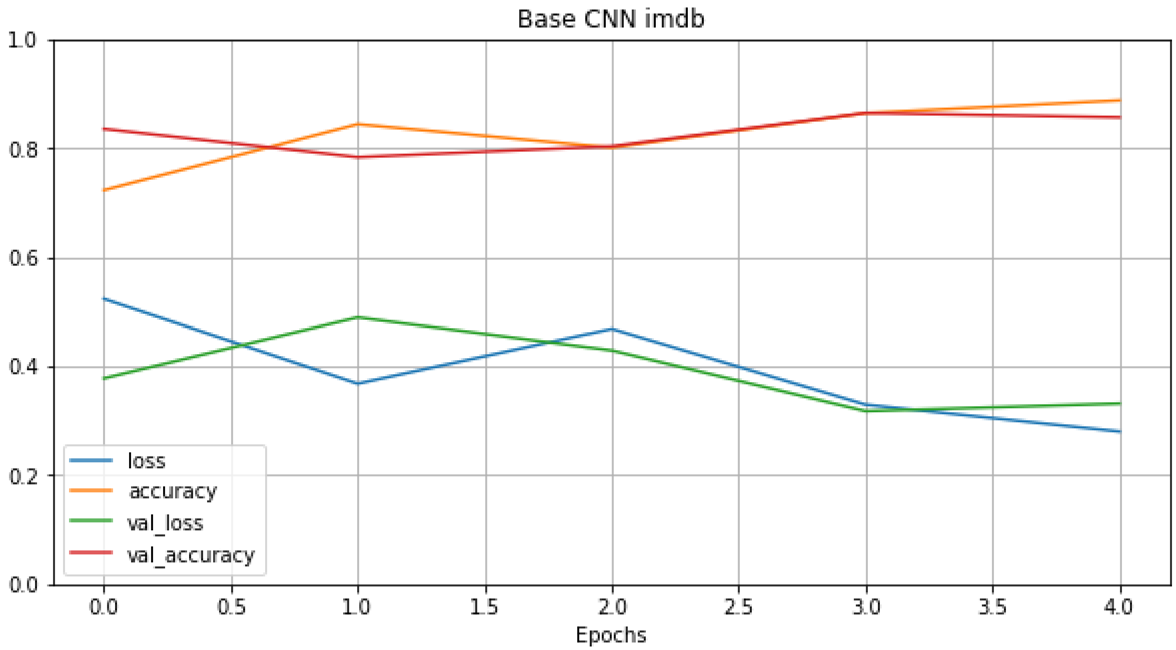
```
Train on 25000 samples, validate on 25000 samples
Epoch 1/5
25000/25000 [==============================] - 831s 33ms/sample - loss: 0.5236 - accuracy: 0.7232 - val_loss:
0.3781 - val_accuracy: 0.8356
Epoch 2/5
25000/25000 [==============================] - 793s 32ms/sample - loss: 0.3683 - accuracy: 0.8443 - val_loss:
0.4897 - val_accuracy: 0.7838
Epoch 3/5
25000/25000 [==============================] - 942s 38ms/sample - loss: 0.4685 - accuracy: 0.8014 - val_loss:
0.4291 - val_accuracy: 0.8038
Epoch 4/5
25000/25000 [==============================] - 913s 37ms/sample - loss: 0.3297 - accuracy: 0.8647 - val_loss:
0.3179 - val_accuracy: 0.8650
Epoch 5/5
25000/25000 [==============================] - 966s 39ms/sample - loss: 0.2804 - accuracy: 0.8881 - val_loss:
0.3317 - val_accuracy: 0.8570
```

```
Out[20]: <matplotlib.axes._subplots.AxesSubplot at 0x1f4208b3fd0>

Out[20]: (0, 1)

Out[20]: Text(0.5, 0, 'Epochs')

Out[20]: Text(0.5, 1.0, 'Base CNN imdb')
```




```
25000/25000 [==============================] - 187s 7ms/sample - loss: 0.3317 - accuracy: 0.8570
```

Out[20]:
```
[0.3317405798149109, 0.85704]
```

```
Model: "sequential_11"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
embedding_11 (Embedding)     (None, 400, 50)           250000    
_________________________________________________________________
lstm_2 (LSTM)                (None, 400, 128)          91648     
_________________________________________________________________
lstm_3 (LSTM)                (None, 128)               131584    
_________________________________________________________________
dense_20 (Dense)             (None, 512)               66048     
_________________________________________________________________
dropout_21 (Dropout)         (None, 512)               0         
_________________________________________________________________
dense_21 (Dense)             (None, 1)                 513       
=================================================================
Total params: 539,793
Trainable params: 539,793
Non-trainable params: 0
_________________________________________________________________
```

How does the basic LSTM compare to the 1D CNN? Is it overfitting as much? is it's testing accuracy better? Answer:

- CNN do not have the ability to know the past information of the input. RNN solves this problem by taking the output as input.
- But normal RNN has the problem of short term memory. Sometimes, domain knowledge would be lost by the time information is passed to output neuron. LSTM were designed to overcome this problem.
- There is a slight improvement in accuracy. I think overfitting is not a problem with 1D CNN either for the given dataset. But RNN performs better than 1D CNN

Step 8: LSTM - 2 layers

In [ ]:
```
# Now lets add another LSTM layer to our model. Did that improve overfitting/accuracy?
model = keras.models.Sequential()
model.add(keras.layers.Embedding(5000,50,input_length=400))

# Here we will add in our LSTM layers. They should be directly after the embedding layer.
model.add(keras.layers.LSTM(128, return_sequences=True))
model.add(keras.layers.LSTM(128, return_sequences=True)) ## Added layer
model.add(keras.layers.LSTM(128))

# Now we will cast the LSTM output to a dense layer to sort it. If you haven't noticed, thick dense layers at the end of networks are how every model 'collects its thoughts'.
model.add(keras.layers.Dense(512, activation='relu'))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(1, activation='sigmoid'))

model.compile('Nadam', 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, y_train,
              batch_size=128,
              epochs=5,
              validation_data=[x_test, y_test])

history = pd.DataFrame(history.history)
history.plot(figsize=(10,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.xlabel('Epochs')
plt.title('Base CNN imdb')
plt.show()

#After training the model, evaluate the test set
model.evaluate(x_test,y_test)

#Print the summary of the model
model.summary()
```

Step 9: LSTM with 256 neurons

In [ ]:
```
# Now lets use larger LSTM layers. What affect did that have? Why do you think that is based off of your know
ledge of RNNs.
# Now we will make our LSTMs. We will use a smaller batch size as they take longer to train.
# We use the same embedding layers as we did for our CNNs.
model = keras.models.Sequential()
model.add(keras.layers.Embedding(5000,50,input_length=400))

# Here we will add in our LSTM layers. They should be directly after the embedding layer.
model.add(keras.layers.LSTM(256, return_sequences=True))
model.add(keras.layers.LSTM(256))

# Now we will cast the LSTM output to a dense layer to sort it. If you haven't noticed, thick dense layers at
the end of networks are how every model 'collects its thoughts'.
model.add(keras.layers.Dense(512, activation='relu'))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(1, activation='sigmoid'))

model.compile('Nadam', 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, y_train,
              batch_size=128,
              epochs=5,
              validation_data=[x_test, y_test])

history = pd.DataFrame(history.history)
history.plot(figsize=(10,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.xlabel('Epochs')
plt.title('Base CNN imdb')
plt.show()

#After training the model, evaluate the test set
model.evaluate(x_test,y_test)

#Print the summary of the model
model.summary()
```

Step 10: LSTM with bi-directional wrapper

In [ ]:
```
# Now lets add Bi-directional layers to each of our RNNs. These make the model learn the data scanning both f
orwards and backwards.
# Here is a detailed description: https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5
bd25a5dd66

# The bidirectional layer is a wrapper, you can apply it like so to each LSTM layer.
model.add(Bidirectional(LSTM(128)))
model = keras.models.Sequential()
model.add(keras.layers.Embedding(5000,50,input_length=400))

# Here we will add in our LSTM layers. They should be directly after the embedding layer.
model.add(keras.layers.Bidirectional(LSTM(128, return_sequences=True)))
model.add(keras.layers.Bidirectional(LSTM(128)))

# Now we will cast the LSTM output to a dense layer to sort it. If you haven't noticed, thick dense layers at
the end of networks are how every model 'collects its thoughts'.
model.add(keras.layers.Dense(512, activation='relu'))
model.add(keras.layers.Dropout(0.5))
model.add(keras.layers.Dense(1, activation='sigmoid'))

model.compile('Nadam', 'binary_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, y_train,
              batch_size=128,
              epochs=5,
              validation_data=[x_test, y_test])

history = pd.DataFrame(history.history)
history.plot(figsize=(10,5))
plt.grid(True)
plt.gca().set_ylim(0,1)
plt.xlabel('Epochs')
plt.title('Base CNN imdb')
plt.show()

#After training the model, evaluate the test set
model.evaluate(x_test,y_test)

#Print the summary of the model
model.summary()
```

Step 11: Attention Mechanisms

- Using attention mechanism, the path from an input word to its translation is much shorter. So the short-term memory limitations of RNNs have much less impact.
- Instead of just sending encoder's final hidden state to the decoder, we now send all of its outputs to the decoder. At each time step, the decoder's memory cell computes a weighted sum of all these encoder outputs. This determines which words it will focus on at this step.
- At each time step, the memory cell receives the inputs as we discussed in above step, plus the hidden state from the previous time step and finally it receieves the target word from the previous time step.
- I think, for NLP , RNN with attention wrapper will do better and for image domain, ResNet with CNN along with attention will do better.

Can you think of anyway to prevent overfitting in an LSTM? got down some ideas and feel free to try them. If you get a signifcant result post it to the discussion board for the rest of the class!

Now that we have looked at the classical examples of 1D CNNs and LSTMs, what do you think are the potential tradeoffs between using each one? Which one makes more sense to use and is there a reason to use LSTMs or RNNs in general for sequential data?

If you are feeling brave and have the extra time I encourage you to impliment an attention layer for both the 1D CNN and bi-directional LSTM and see how much Attention helps. You can also use image attention layers to improve 2D CNNs!