BST 261: Data Science II

Lecture 13

Heather Mattie

Department of Biostatistics
Harvard T.H. Chan School of Public Health
Harvard University

April 30, 2018

LSTM in Keras

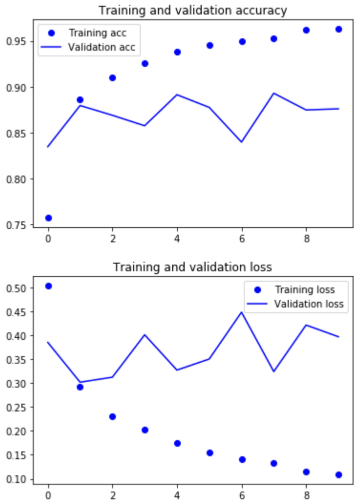
LSTM in Keras

- Now that you have an idea of how LSTM works, let's implement it in Keras
- We set up a model using an LSTM layer and train it on the IMDB data
- The network is similar to the one with `SimpleRNN` that we discussed last week
- We only specify the output dimensionality of the LSTM layer, and leave every other argument (there are lots) to the Keras defaults

```
from keras.layers import LSTM

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(LSTM(32))
model.add(Dense(1, activation='sigmoid'))

model.compile(optimizer='rmsprop',
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

LSTM in Keras

- Now we get 89% accuracy, compared to 85% using the `SimpleRNN` layer
- This also performs better than when we used a feedforward network a few lectures ago, and here we are using less data



Improving RNNs Performance and Generalization

Improving RNNs

- We will cover 3 techniques for improving RNNs:
  - **Recurrent dropout**: fights overfitting, different from the kind of dropout you are already familiar with
  - **Stacking recurrent layers**: increases generalizability, but comes with a higher computational cost
  - **Bidirectional recurrent layers**: increase accuracy and fight forgetting issues

Temperature Forecasting

- RNNs can be applied to any type of sequence data, not just text
- We will be using a weather timeseries data set recorded at the Weather Station at Max Planck Institute for Biochemistry in Jena, Germany
- 14 different variables were recorded **every 10 minutes** over several years, starting in 2003
    - Air temperature, atmospheric pressure, humidity, wind direction, etc.
- We will be using data from 2009-2016 to build a model that predicts air temperature 24 hours in the future using data from the last few days

- Load the data and take a look at the variables

```
cd ~/Downloads
mkdir jena_climate
cd jena_climate
wget https://s3.amazonaws.com/keras-datasets/jena_climate_2009_2016.csv.zip
unzip jena_climate_2009_2016.csv.zip

import os

data_dir = '/home/ubuntu/data/'
fname = os.path.join(data_dir, 'jena_climate_2009_2016.csv')

f = open(fname)
data = f.read()
f.close()

lines = data.split('\n')
header = lines[0].split(',')
lines = lines[1:]

print(header)
print(len(lines))

['"Date Time"', '"p (mbar)"', '"T (degC)"', '"Tpot (K)"', '"Tdew (degC)"', '"rh
    (%)"', '"VPmax (mbar)"', '"VPact (mbar)"', '"VPdef (mbar)"', '"sh (g/kg)"',
     '"H2OC (mmol/mol)"', '"rho (g/m**3)"', '"wv (m/s)"', '"max. wv (m/s)"', '"
    wd (deg)"']
420551
```

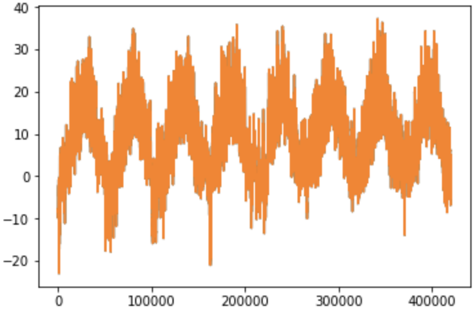
- Convert all lines of data into a numpy array

```
import numpy as np

float_data = np.zeros((len(lines), len(header) - 1))
for i, line in enumerate(lines):
    values = [float(x) for x in line.split(',')[1:]]
    float_data[i, :] = values
```

Temperature Forecasting

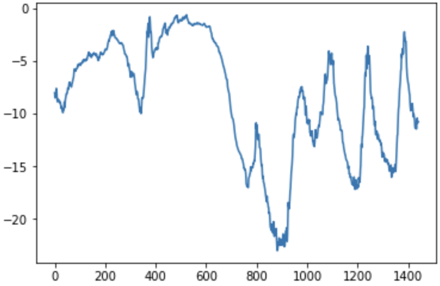
- Plot the temperature over time (years)

```python
from matplotlib import pyplot as plt

temp = float_data[:, 1]  # temperature (in degrees Celsius)
plt.plot(range(len(temp)), temp)
plt.show()
```



- Plot the temperature over time (a few days)
- Notice that there is periodicity present, but that it isn't as consistent as the last plot - this will make predicting the weather in the next 24 hours using data from a few days beforehand more challenging

```
# Plot the temperature for the last 10 days
plt.plot(range(1440), temp[:1440])
plt.show()
```



Temperature Forecasting

- Task: given data going as far back as `lookback` timesteps (here a timestep is 10 minutes) and sampled every `steps` timesteps, can you predict the temperature in `delay` timesteps?

- `lookback` = 720 - we will go back 5 days
- `steps` = 6 - observations will be sampled at one data point per hour
- `delay` = 144 - targets will be 24 hours in the future

- Process the data:
  - Normalize all variables

```
mean = float_data[:200000].mean(axis=0)
float_data -= mean
std = float_data[:200000].std(axis=0)
float_data /= std
```

- Generate samples
  - `data`: The original array of floating point data, which we just normalized in the code on the last slide
  - `lookback`: How many timesteps back should our input data go
  - `delay`: How many timesteps in the future should our target be
  - `min_index` and `max_index`: Indices in the data array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another one for testing.
  - `shuffle`: Whether to shuffle our samples or draw them in chronological order
  - `batch_size`: The number of samples per batch
  - `step`: The period, in timesteps, at which we sample data. We will set it 6 in order to draw one data point every hour.

```python
def generator(data, lookback, delay, min_index, max_index, shuffle=False,
       batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(
                min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
            i += len(rows)

        samples = np.zeros((len(rows),
                           lookback // step,
                           data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
            indices = range(rows[j] - lookback, rows[j], step)
            samples[j] = data[indices]
            targets[j] = data[rows[j] + delay][1]
        yield samples, targets
```

- Use the generator function to instantiate three generators, one for training, one for validation and one for testing. Each will look at different temporal segments of the original data: the training generator looks at the first 200,000 timesteps, the validation generator looks at the following 100,000, and the test generator looks at the remainder.

```
lookback = 1440
step = 6
delay = 144
batch_size = 128

train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step,
                      batch_size=batch_size)
val_gen = generator(float_data,
                    lookback=lookback,
                    delay=delay,
                    min_index=200001,
                    max_index=300000,
                    step=step,
                    batch_size=batch_size)
```

Temperature Forecasting

```python
test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=300001,
                     max_index=None,
                     step=step,
                     batch_size=batch_size)

# This is how many steps to draw from `val_gen` in order to see the whole
     validation set:
val_steps = (300000 - 200001 - lookback) // batch_size

# This is how many steps to draw from `test_gen` in order to see the whole test
     set:
test_steps = (len(float_data) - 300001 - lookback) // batch_size
```

Temperature Forecasting Baseline

- We need to come up with a baseline benchmark to beat
- Common-sense approach: always predict that the temperature 24 hours from now will be equal to the temperature now
- We'll use mean absolute error (MAE) to measure loss

```
# Define MAE
np.mean(np.abs(preds - targets))

def evaluate_naive_method():
    batch_maes = []
    for step in range(val_steps):
        samples, targets = next(val_gen)
        preds = samples[:, -1, 1]
        mae = np.mean(np.abs(preds - targets))
        batch_maes.append(mae)
    print(np.mean(batch_maes))

evaluate_naive_method()
0.289735972991
```

- We get MAE = 0.29. Since our temperature data has been normalized to be centered at 0 and have a standard deviation of 1, this number is not immediately interpretable. It translates to an average absolute error of 0.29 * `temperature_std` degrees Celsius, i.e. 2.57°C. That's a fairly large average absolute error – now the task is to leverage our knowledge of deep learning to do better.

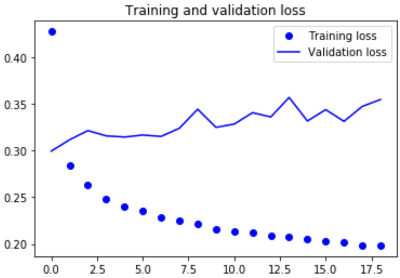
Temperature Forecasting Simple Model

- Let's first try a simple model before developing a more complex one
- In general it's best to start with a basic model and then work your way up in complexity

```

from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Flatten(input_shape=(lookback // step, float_data.shape[-1])))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

Temperature Forecasting Simple Model

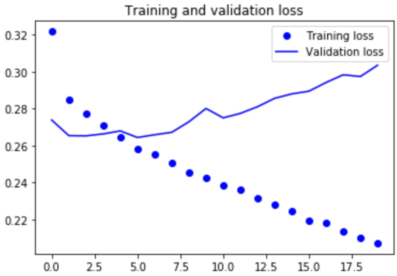
- The loss hovers around the loss value of the common-sense baseline, proving that having a baseline to beat is useful
- Why isn't the loss better than the baseline?
  - It doesn't take time into account
  - The hypothesis space of possible models is large



- Now let's fit a model with a `GRU` layer

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

- This model is better than the previous simple model and the common-sense baseline
- Evidence of overfitting, so let's try dropout next



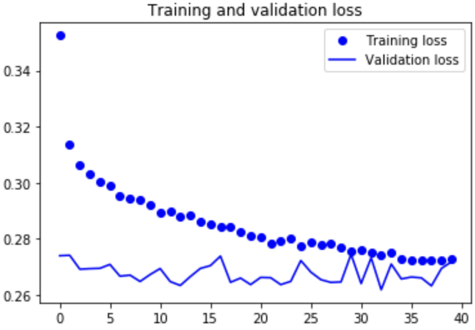
Recurrent Dropout

Recurrent Dropout

- It turns out that the classic technique of dropout we saw in earlier lectures can't be applied in the same way for recurrent layers
- Applying dropout before a recurrent layer impedes learning rather than helping to implement regularization
- The proper way to apply dropout with a recurrent network was discovered in 2015
  - Yarin Gal, "Uncertainty in Deep Learning (PhD Thesis)," `http://mlg.eng.cam.ac.uk/yarin/blog_2248.html`
  - The **same pattern of dropped units** should be applied at every timestep
  - This allows the network to properly propagate its learning error rate through time - a temporally random dropout pattern would disrupt the error signal and hinder the learning process
- Yarin's mechanism has been built into Keras
  - Every recurrent layer has 2 dropout-related arguments:
    - `dropout`: a float number specifying the dropout rate for input units of the layer
    - `recurrent_dropout`: a float number specifying the dropout rate of the recurrent units

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.2,
                     recurrent_dropout=0.2,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

- Success: we are no longer overfitting during the first 30 epochs
- We have more stable evaluation scores, but our best scores are not much lower than they were previously



Stacking Recurrent Layers

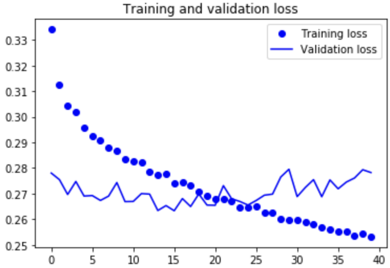
Temperature Forecasting Stacking Recurrent Layers

- We have dealt with overfitting, but our model still doesn't perform much better than the common-sense baseline
- Let's add a recurrent layer to increase the complexity of the model

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.GRU(32,
                     dropout=0.1,
                     recurrent_dropout=0.5,
                     return_sequences=True,
                     input_shape=(None, float_data.shape[-1])))
model.add(layers.GRU(64, activation='relu',
                     dropout=0.1,
                     recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

Temperature Forecasting Stacking Recurrent Layers

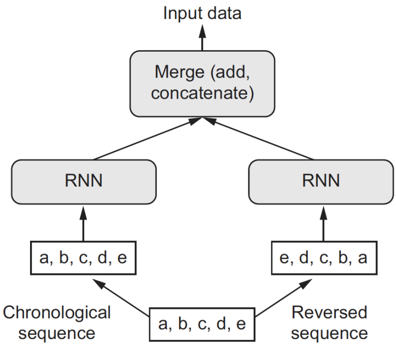
- The added layer does improve the results, but not significantly
- We could add more layers, but this will be more computationally expensive
- Let's now try another approach - a bidirenctional RNN



Bidirectional RNNs

Bidirectional RNNs

- A **bidirectional RNN** (BRNN) can offer greater performance on certain tasks
- Frequently used in natural-language processing (NLP)
- BRNNs exploit the order sensitivity of RNNs
  - Uses 2 regular RNNs, each of which processes the input sequence in one direction (chronologically and antichronologically), and then merges their representations
  - Catches patterns that may be overlooked by a regular RNN



Temperature Forecasting with Antichronological RNN

```python
def reverse_order_generator(data, lookback, delay, min_index, max_index,
                            shuffle=False, batch_size=128, step=6):
    if max_index is None:
        max_index = len(data) - delay - 1
    i = min_index + lookback
    while 1:
        if shuffle:
            rows = np.random.randint(
                min_index + lookback, max_index, size=batch_size)
        else:
            if i + batch_size >= max_index:
                i = min_index + lookback
            rows = np.arange(i, min(i + batch_size, max_index))
            i += len(rows)

        samples = np.zeros((len(rows),
                            lookback // step,
                            data.shape[-1]))
        targets = np.zeros((len(rows),))
        for j, row in enumerate(rows):
            indices = range(rows[j] - lookback, rows[j], step)
            samples[j] = data[indices]
            targets[j] = data[rows[j] + delay][1]
        yield samples[:, ::-1, :], targets
```

Temperature Forecasting with Antichronological RNN

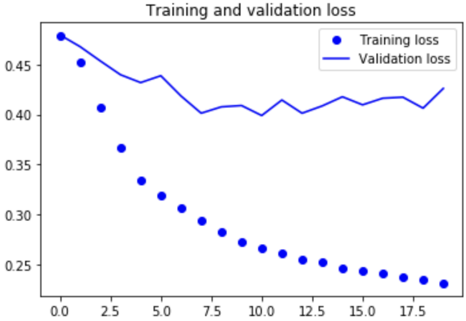
```
train_gen_reverse = reverse_order_generator(
    float_data,
    lookback=lookback,
    delay=delay,
    min_index=0,
    max_index=200000,
    shuffle=True,
    step=step,
    batch_size=batch_size)
val_gen_reverse = reverse_order_generator(
    float_data,
    lookback=lookback,
    delay=delay,
    min_index=200001,
    max_index=300000,
    step=step,
    batch_size=batch_size)
```

Temperature Forecasting with Antichronological RNN

```
model = Sequential()
model.add(layers.GRU(32, input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen_reverse,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen_reverse,
                              validation_steps=val_steps)
```

Temperature Forecasting with Antichronological RNN

- The antichronological RNN significantly underperforms the original, chronological RNN as well as the common-sense baseline
- Why?
  - The `GRU` layer will be better at remembering the recent past than the distant past
  - More recent weather data points are more predictive than older data points
- Note that this isn't true for other types of problems
  - Order typically doesn't matter as much in NLP



Temperature Forecasting with Bidirectional RNN

- To instantiate a bidirectional RNN in Keras, use the `Bidirectional layer`, which takes as first argument a recurrent layer instance
- `Bidirectional` will create a second, separate instance of this recurrent layer, and will use one instance for processing the input sequences in chronological order and the other instance for processing the input sequences in reversed order

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Bidirectional(
    layers.GRU(32), input_shape=(None, float_data.shape[-1])))
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=40,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

Temperature Forecasting with Bidirectional RNN

- This model performs about as well as the regular `GRU` layer (Loss ≈ 0.3)
- All of the predictive capacity comes from the chronological half of the network

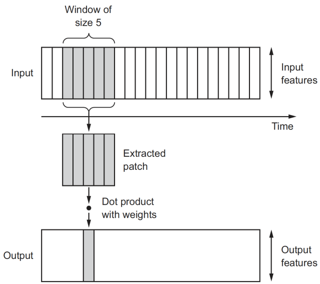
Other Performance Options

- There are several other things you can try to improve performance
  - Change the number of units in each recurrent layer
  - Try using `LSTM` layers instead of `GRU` layers
  - Change the learning rate used by the `RMSprop` optimizer
  - Try a bigger densely connected classifier on top of the recurrent layers

1D Convolution for Sequence Data

1D Convolution for Sequence Data

- Recall that CNNs can extract features from local input patches and then recognize them anywhere
- If we think of time as a spatial dimension, we can use 1D CNNs for sequence data
- Great for audio generation and machine translation
- Faster to run than RNNs
- 1D CNNs extract subsequences (patches) from sequences and perform the same transformation on each subsequence
  - A pattern learned at a specific position of a sequence can be recognized at a different position (translation invariant)
- Pooling in this case is similar to what we have seen before - output the maximum or average value of a subsequence



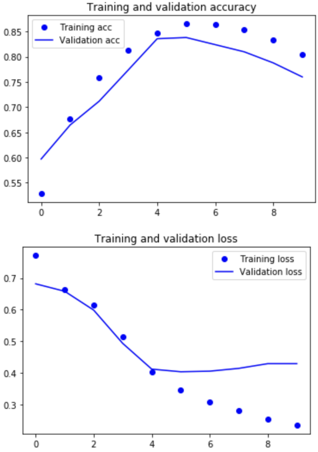
1D CNNs for Sequence Data

- In Keras, use the `Conv1D` layer
  - Takes as input (`samples`, `time`, `features`)
  - Returns 3D tensors
- Combine `Conv1D` layer with a `MaxPooling1D` layer
- End with a `GlobalMaxPooling` or `Flatten` layer
- Can use larger windows for 1D CNNs - typically windows of size 7 or 9
  - In a 2D convolution layer, a 3×3 filter contains 9 feature vectors
  - A 1D convolution layer with a window of size 3 contains only 3 vectors

```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Embedding(max_features, 128, input_length=max_len))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.MaxPooling1D(5))
model.add(layers.Conv1D(32, 7, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(lr=1e-4),
              loss='binary_crossentropy',
              metrics=['acc'])
history = model.fit(x_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

1D CNN IMDB Example

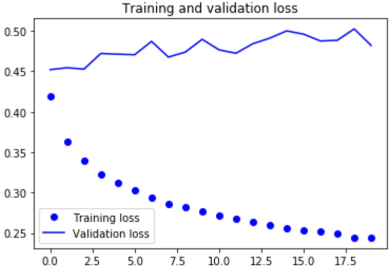
- The validation accuracy is a little lower than when using an RNN with an `LSTM` layer, but the running time for this model is much lower



- 1D CNNs process subsequences independently and aren't sensitive to the order of the timesteps - thus, they don't perform well when faced with long sequences
- Could try stacking 1D CNN layers, but still doesn't induce order sensitivity

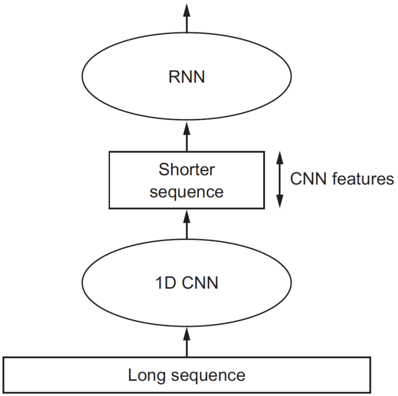
```
from keras.models import Sequential
from keras import layers
from keras.optimizers import RMSprop

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                        input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GlobalMaxPooling1D())
model.add(layers.Dense(1))

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

- Doesn't beat the common-sense baseline
  - Due to the fact that 1D CNNs aren't time sensitive
- Let's try combining CNNs and RNNs



Combining CNNs and RNNs

- To combine the speed of 1D CNNs with the time sensitivity of RNNs, could do the following:
    - Use the 1D CNN as a preprocessing step
    - Feed this output into an RNN
- The CNN turns long sequences into much shorter sequences



Temperature Forecasting with CNN and RNN

- With the combination of a CNN and RNN we can now either look at data from longer ago (increase the `lookback` parameter), or look at high-resolution timeseries (decrease the `step` parameter)
- Let's decrease the `step` parameter by half - this gives us sequences that are twice as long
- Temperature data is now sampled at a rate of 1 point per 30 minutes

```
# This was previously set to 6 (one point per hour).
# Now 3 (one point per 30 min).
step = 3
lookback = 720  # Unchanged
delay = 144 # Unchanged

train_gen = generator(float_data,
                      lookback=lookback,
                      delay=delay,
                      min_index=0,
                      max_index=200000,
                      shuffle=True,
                      step=step)
val_gen = generator(float_data,
                    lookback=lookback,
                    delay=delay,
                    min_index=200001,
                    max_index=300000,
                    step=step)
```

Temperature Forecasting with CNN and RNN

```
test_gen = generator(float_data,
                     lookback=lookback,
                     delay=delay,
                     min_index=300001,
                     max_index=None,
                     step=step)
val_steps = (300000 - 200001 - lookback) // 128
test_steps = (len(float_data) - 300001 - lookback) // 128

model = Sequential()
model.add(layers.Conv1D(32, 5, activation='relu',
                        input_shape=(None, float_data.shape[-1])))
model.add(layers.MaxPooling1D(3))
model.add(layers.Conv1D(32, 5, activation='relu'))
model.add(layers.GRU(32, dropout=0.1, recurrent_dropout=0.5))
model.add(layers.Dense(1))

model.summary()

model.compile(optimizer=RMSprop(), loss='mae')
history = model.fit_generator(train_gen,
                              steps_per_epoch=500,
                              epochs=20,
                              validation_data=val_gen,
                              validation_steps=val_steps)
```

- This model doesn't perform as well as the regularized `GRU` model, but it does run a lot faster
- This model also looks at twice as much data as the other model, but that doesn't seem to help with accuracy in this case - it could for other data sets