BST 261: Data Science II

Lectures 13 and 14

Heather Mattie

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

April 30 & May 2, 2018

LSTM in Keras

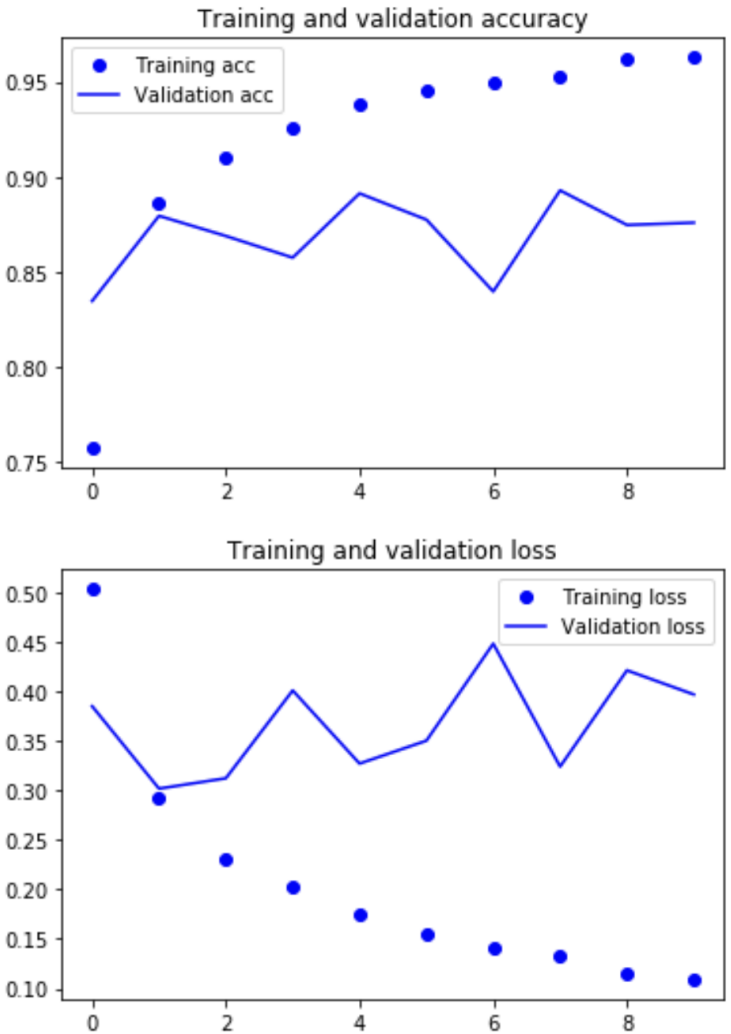
LSTM in Keras

- 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

6 recordings per hour
144 recordings per day
52,560 recordings per year

Temperature Forecasting

- 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
```

- each line is 1 recording
- variable values are separated by commas
- drop first row (header)
- ≈ 8 years

Temperature Forecasting

- Convert all lines of data into a numpy array

```
1 import numpy as np
2
3 float_data = np.zeros((len(lines), len(header) - 1))
4 for i, line in enumerate(lines):
5     values = [float(x) for x in line.split(',')[1:]]
6     float_data[i, :] = values
```

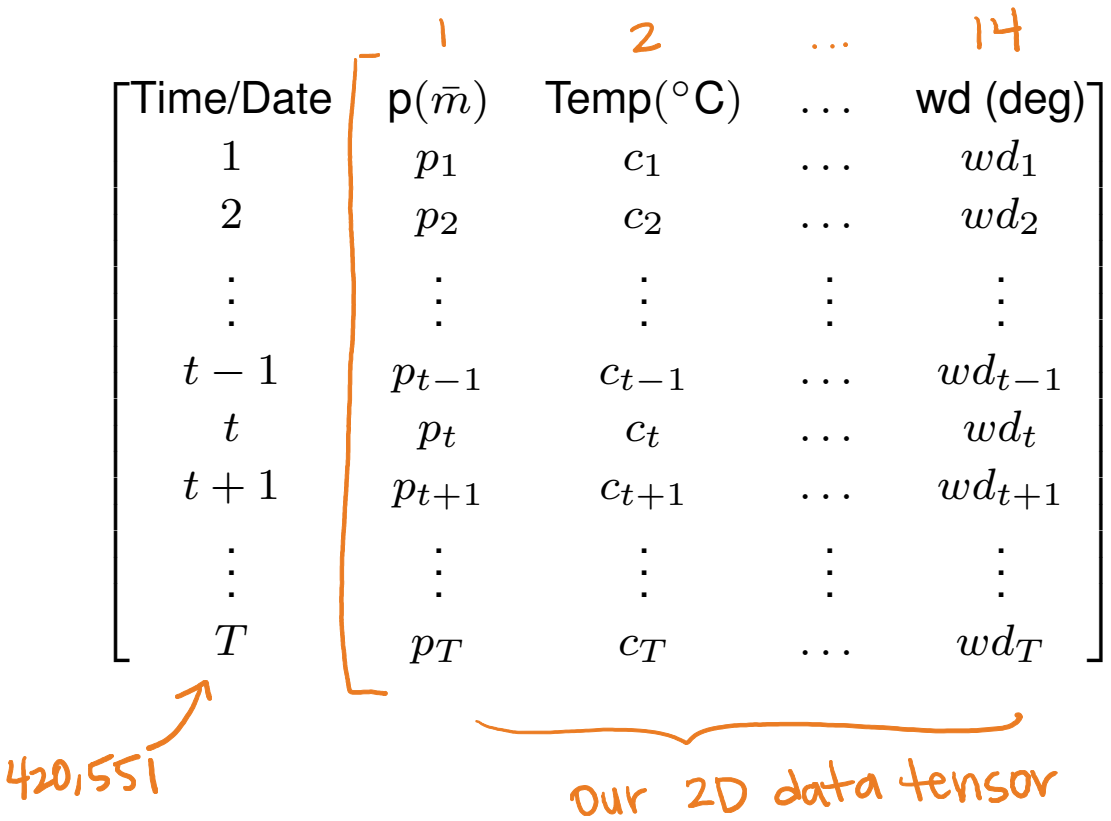
420,551

14

- One recording is saved as one row of 14 values in a 2D tensor
- Data shape: (420,551, 14)

drop first column value (date/time, don't need)

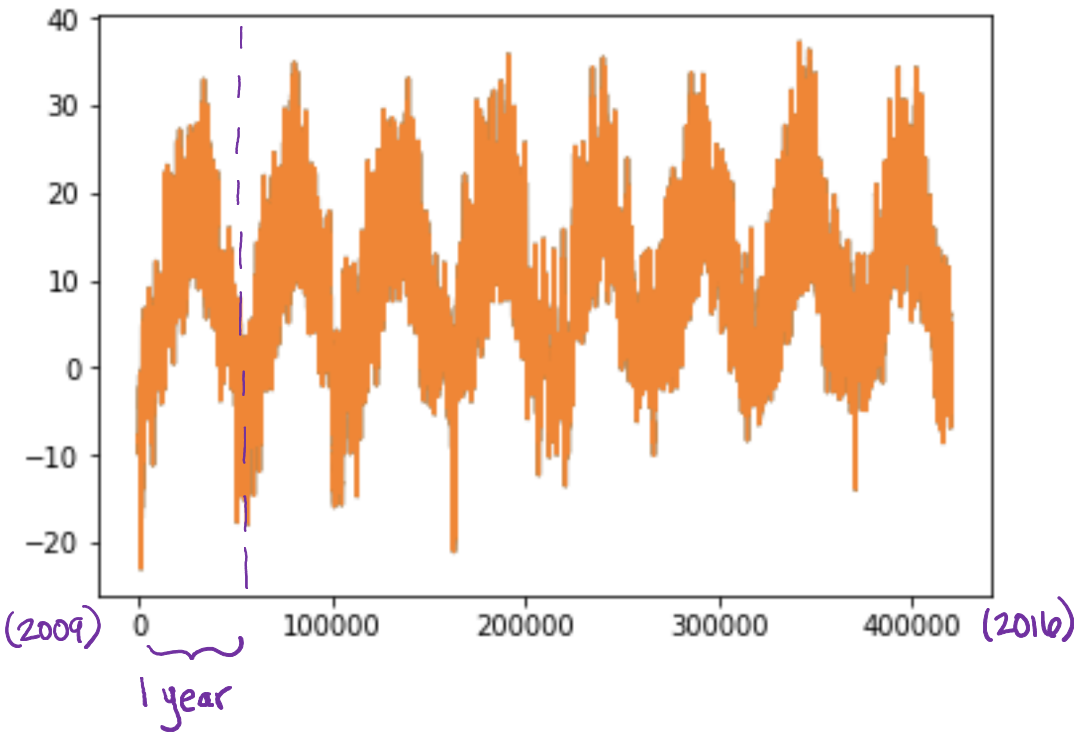
Temperature Forecasting Data



Temperature Forecasting

- Plot the temperature over time (years)

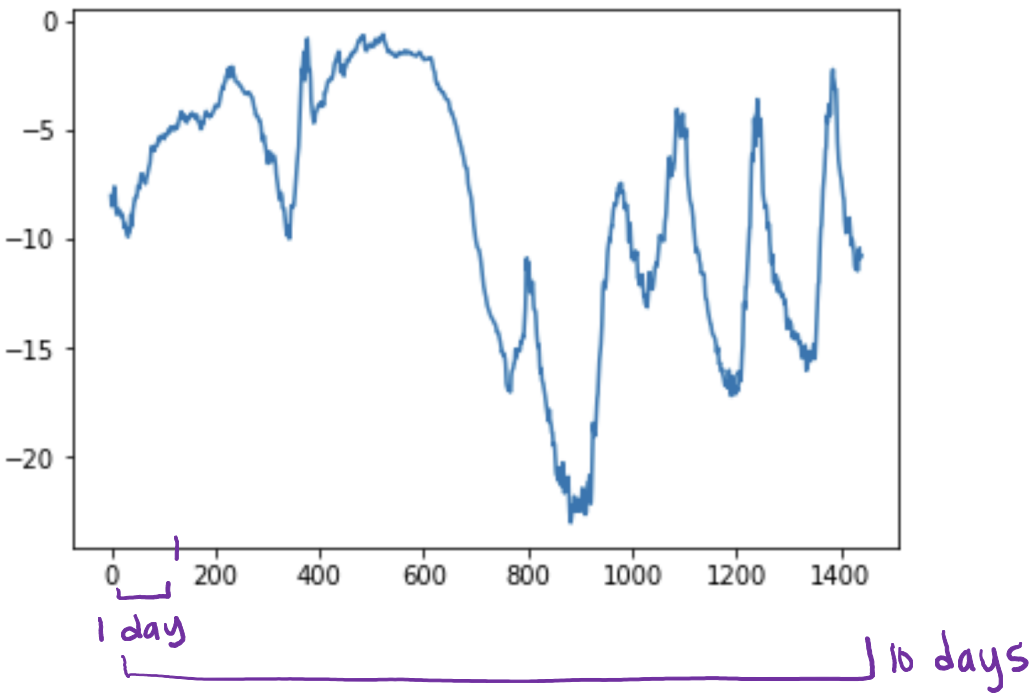
```
1 from matplotlib import pyplot as plt
2
3 temp = float_data[:, 1]  # temperature (in degrees Celsius)
4 plt.plot(range(len(temp)), temp)
5 plt.show()
```



Temperature Forecasting

- 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` = 1440 - we will go back 10 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 = 0, variance / sd = 1 ($\bar{x}$) (σ^2) (σ)

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

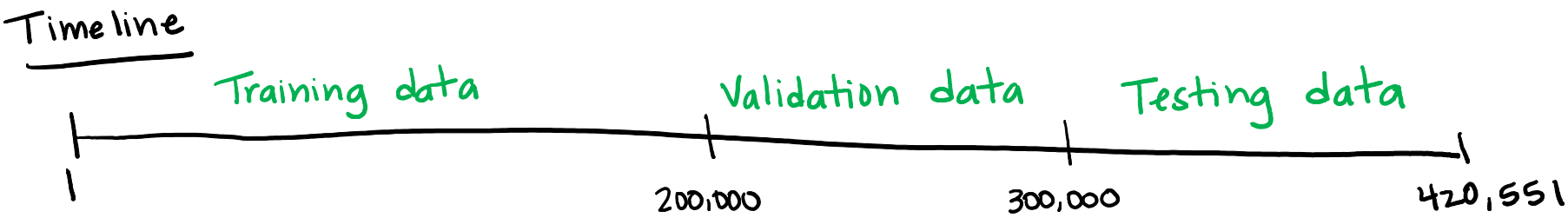
→ collapse by rows to find $\bar{x}$ and sd for each column

→ $x_i = \frac{x_i - \bar{x}}{\sigma}$

Temperature Forecasting

- 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 (1440 ←)
  - `delay`: How many timesteps in the future should our target be (144 ←)
  - `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. (6 ←)

Temperature Forecasting



1 timestep = 10 minutes

lookback = 1440 timesteps = 10 days

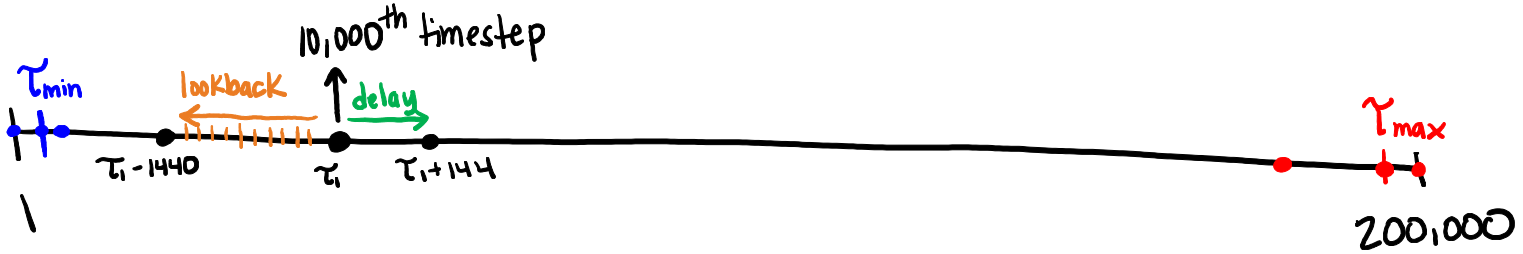
steps = 6 (sample 1 recording every 6 timesteps, or once every hour)

→ 1440/6 = 240 recordings to represent data from the last 10 days

delay = 144 (24 hours into the future)

→ This determine our true temperature, y
Our goal is to estimate y with $\hat{y}$

Training Set Timeline



1. Randomly sample a point in time, τ_i

Example: $\tau_i = 10{,}000^{th}$ timestep

$X^{<1>} = \tau_i - 1440 = 8{,}560^{th}$

$X^{<2>} = \tau_i - 1440 + 6 = 8{,}566^{th}$

$X^{<3>} = \tau_i - 1440 + 12 = 8{,}572^{nd}$

$X^{<240>} = \tau_i - 1440 + 1440 = 10{,}000^{th}$

$X = \begin{bmatrix} X^{<1>} \\ X^{<2>} \\ \vdots \end{bmatrix} = X_i$

240×14 training example matrix

$y_i = \tau_i + 144$

$= 10{,}144^{th}$ timestep

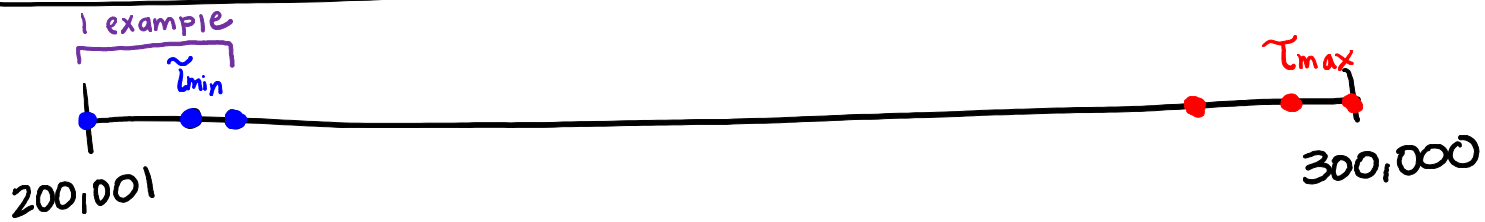
= vector with 14 elements

Training example (X_i, y_i)

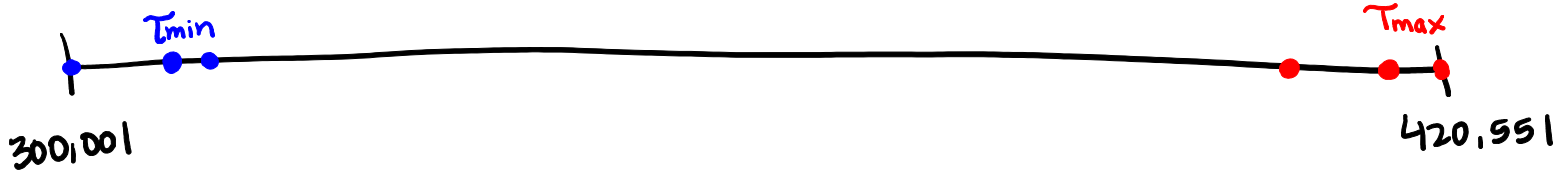
* τ_{min}: We need all 10 days worth of past data for each example. Thus, the minimum value τ_i can take is $1 + 1440 = 1441$. If we choose $\tau_i < 1441$, won't have enough timesteps in past (lookout is too large).

2. Repeat this process multiple times
3. Split training examples into batches
4. Feed into network

Validation Set Timeline



Test Set Timeline



Can progress chronologically - don't need to randomly choose points along timeline.

Temperature Forecasting

```
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
```

training set

randomly choose points in time

choose patches of timesteps

Validation and test sets

14

(X) one batch of input data

corresponding target temperatures (y)

Temperature Forecasting

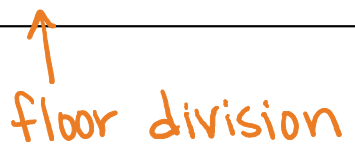
- Use the generator function to instantiate three generators, one for training, one for validation and one for testing. Each will look at different temporal segments of the original data: the training generator looks at the first 200,000 timesteps, the validation generator looks at the following 100,000,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

— only shuffle the training data
* does not shuffle timesteps. it randomly chooses points in the training data as starting points to create training examples matrix

Temperature Forecasting

```
1  test_gen = generator(float_data,
2                       lookback=lookback,
3                       delay=delay,
4                       min_index=300001,
5                       max_index=None,
6                       step=step,
7                       batch_size=batch_size)
8
9  # This is how many steps to draw from `val_gen` in order to see the whole
       validation set:
10 val_steps = (300000 - 200001 - lookback) // batch_size
11
12 # This is how many steps to draw from `test_gen` in order to see the whole test
       set:
13 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

$$\frac{1}{m}\sum_{i=1}^{m}|\hat{y}_i - y_i|$$

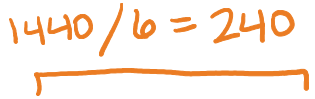
```
# Define MAE
np.mean(np.abs(preds - targets))

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

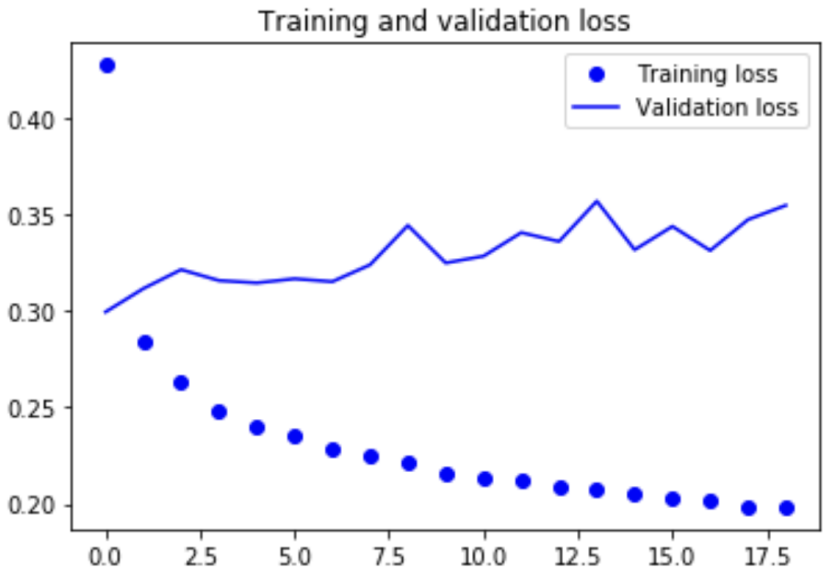
- Let's first try a simple model before developing a more complex one
- In general it's best to start with a basic model and then work your way up in complexity

```
1
2  from keras.models import Sequential
3  from keras import layers
4  from keras.optimizers import RMSprop
5
6  model = Sequential()
7  model.add(layers.Flatten(input_shape=(lookback // step, float_data.shape[-1])))
8  model.add(layers.Dense(32, activation='relu'))
9  model.add(layers.Dense(1))
10
11 model.compile(optimizer=RMSprop(), loss='mae')
12 history = model.fit_generator(train_gen,
13                                steps_per_epoch=500,
14                                epochs=20,
15                                validation_data=val_gen,
16                                validation_steps=val_steps)
```

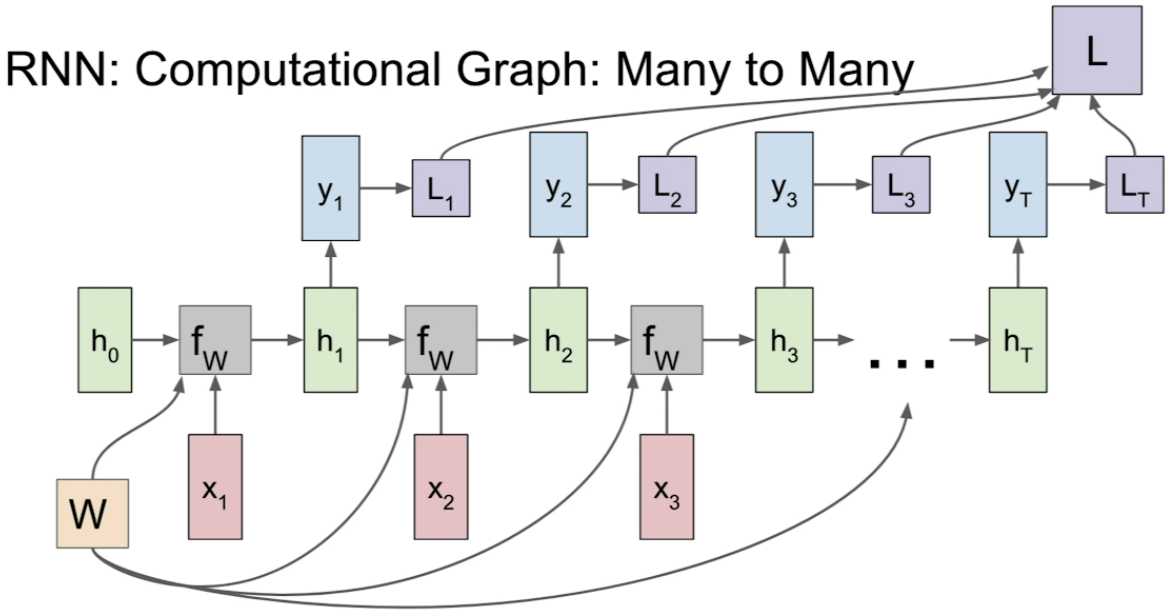


Temperature Forecasting Simple Model

- 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



Temperature Forecasting Simple RNN Model



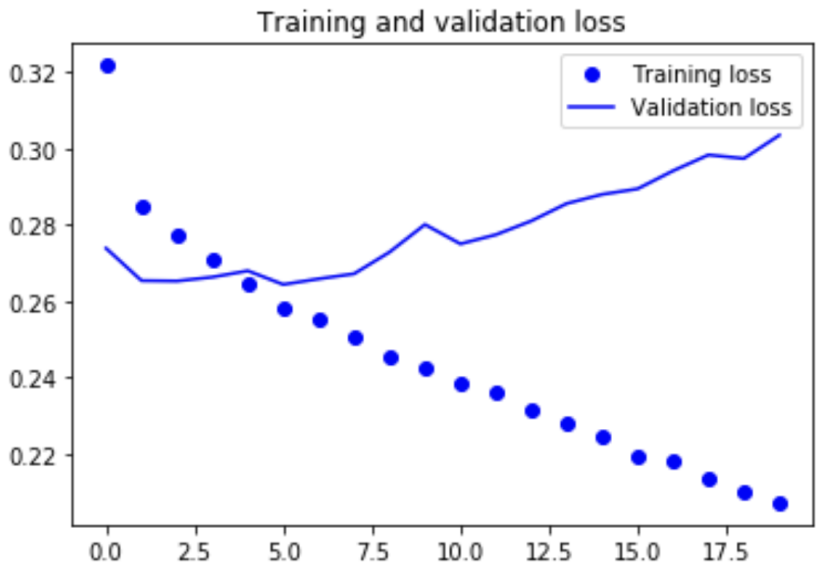
Temperature Forecasting Simple RNN Model

- 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)
```

Temperature Forecasting Simple RNN Model

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



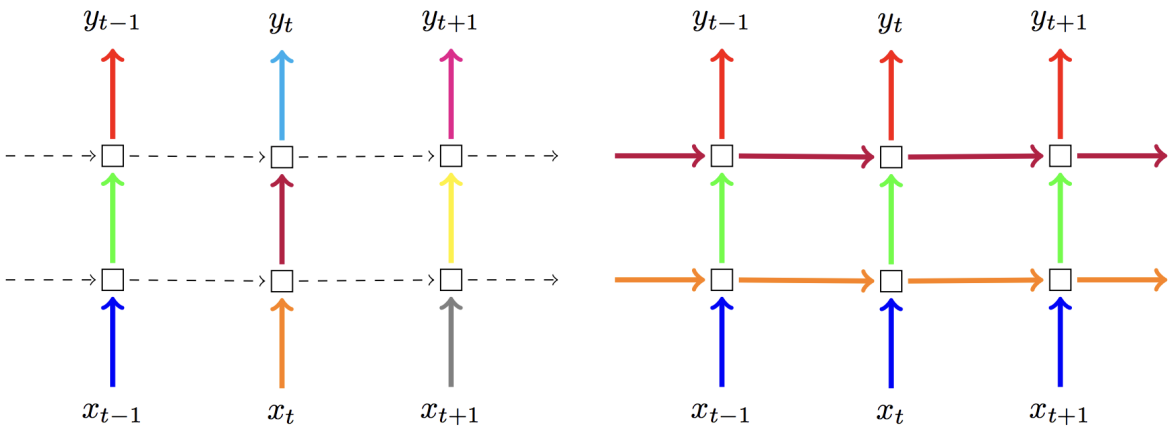
Recurrent Dropout

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

Recurrent Dropout

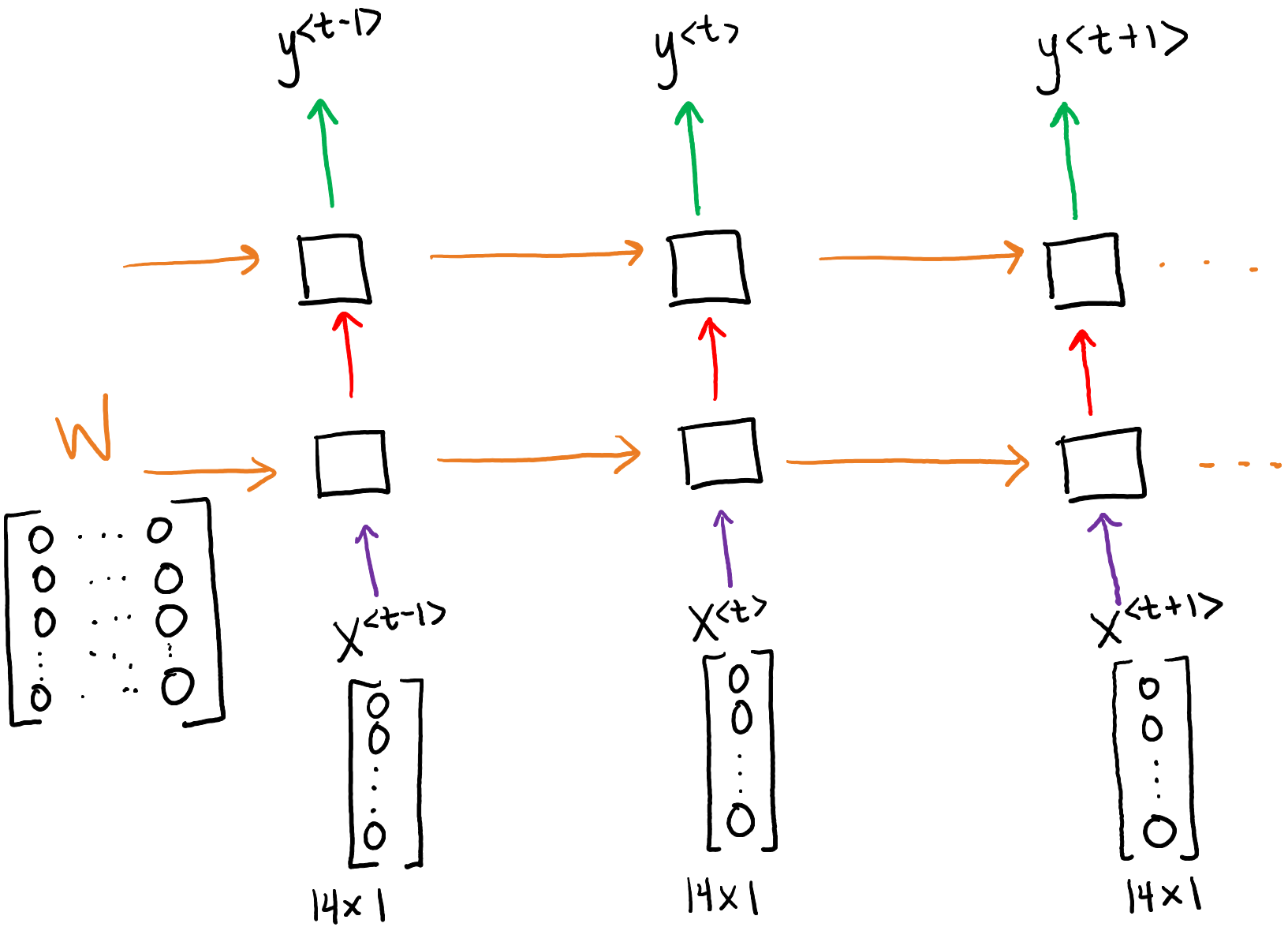
- Gal and Ghahramani, (2015) “A Theoretically Grounded Application of Dropout in Recurrent Neural Networks"
- `https://pdfs.semanticscholar.org/58ab/4ad843ce2a3494578ca62a70d189c10d7e3b.pdf`



(a) Naive dropout RNN (b) Variational RNN

Figure 1: **Depiction of the dropout technique following our Bayesian interpretation (right) compared to the standard technique in the field (left).** Each square represents an RNN unit, with horizontal arrows representing time dependence (recurrent connections). Vertical arrows represent the input and output to each RNN unit. Coloured connections represent dropped-out inputs, with different colours corresponding to different dropout masks. Dashed lines correspond to standard connections with no dropout. Current techniques (naive dropout, left) use different masks at different time steps, with no dropout on the recurrent layers. The proposed technique (Variational RNN, right) uses the same dropout mask at each time step, including the recurrent layers.

Recurrent Dropout

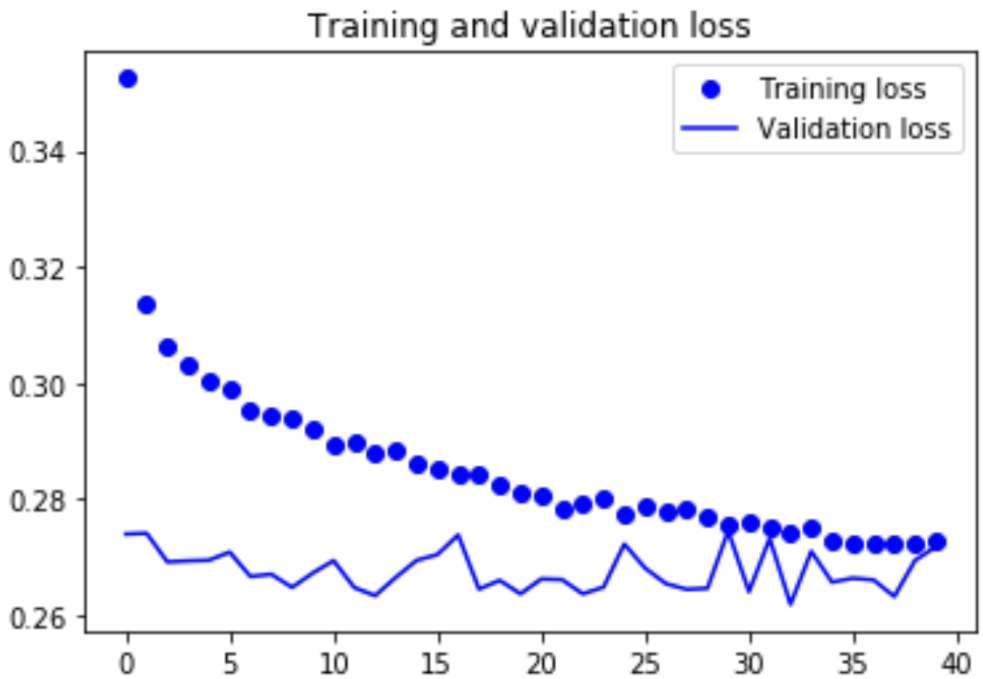


Temperature Forecasting Simple RNN Model with Dropout

```
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)
```

Temperature Forecasting Simple RNN Model with Dropout

- 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

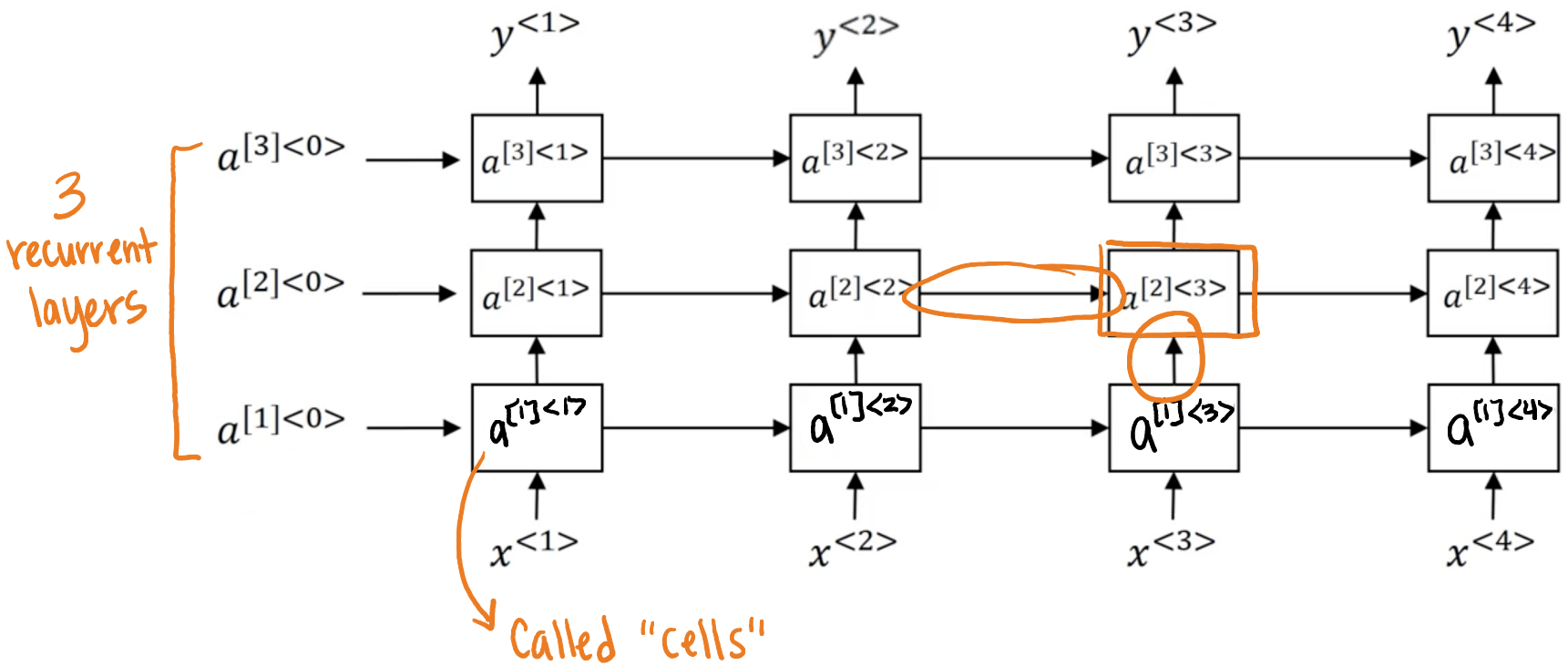


Stacking Recurrent Layers

Stacking Recurrent Layers

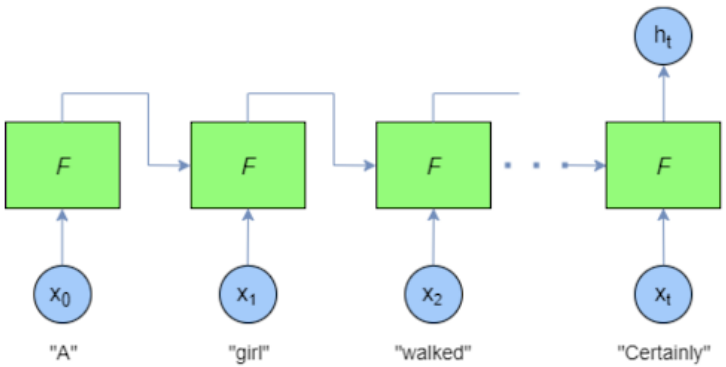
$a^{[l]<t>}$ — l: layer, t: timestep

$$a^{[2]<3>} = g(W_a^{[2]}[a^{[2]<2>}, a^{[1]<3>}] + b_a^{[2]})$$

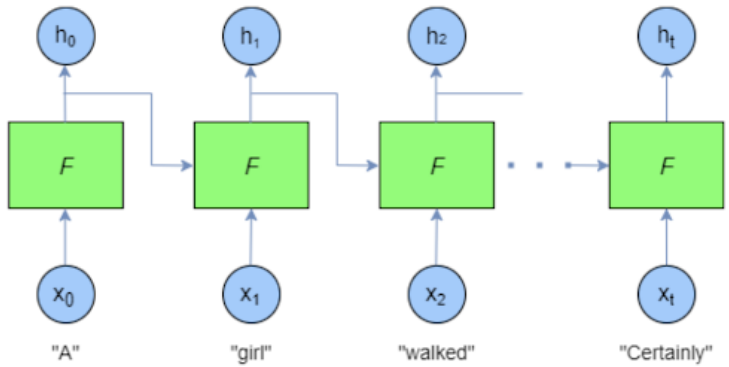


- The output dimension of a layer is the number of nodes in a layer

Stacking Recurrent Layers



return_sequences = False



return_sequences = True

Keras LSTM return sequences argument comparison

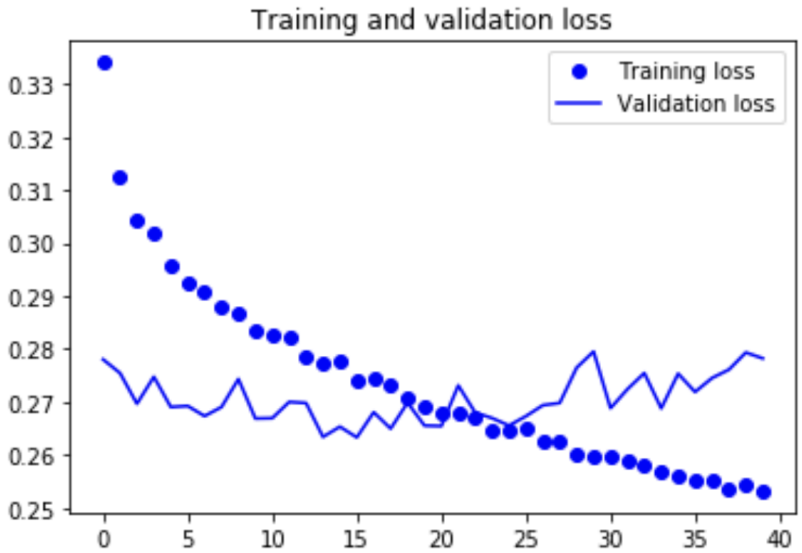
Temperature Forecasting Stacking Recurrent Layers

- 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

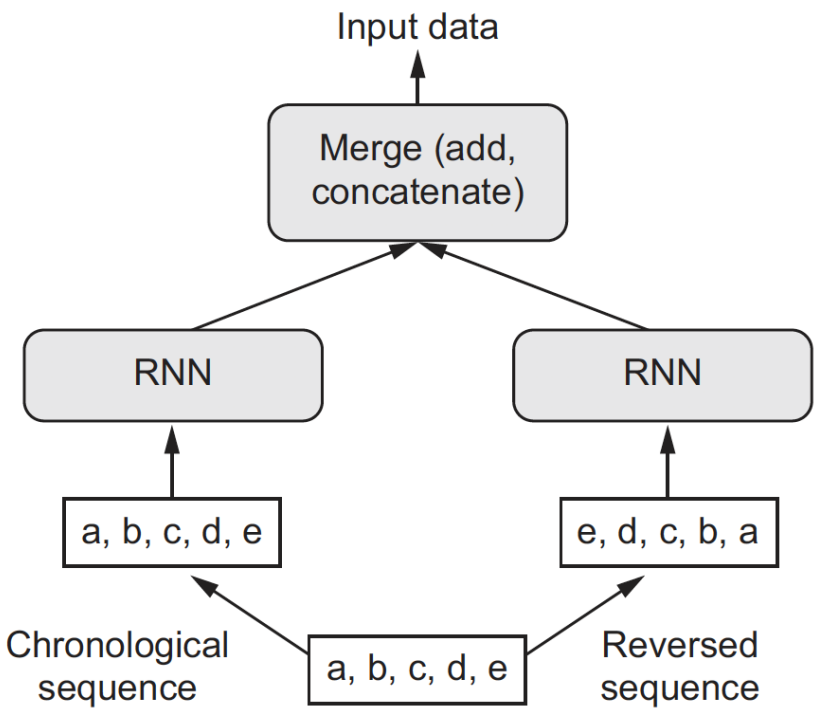
- The added layer does improve the results, but not significantly
- We could add more layers, but this will be more computationally expensive
- Let's now try another approach - a bidirenctional RNN



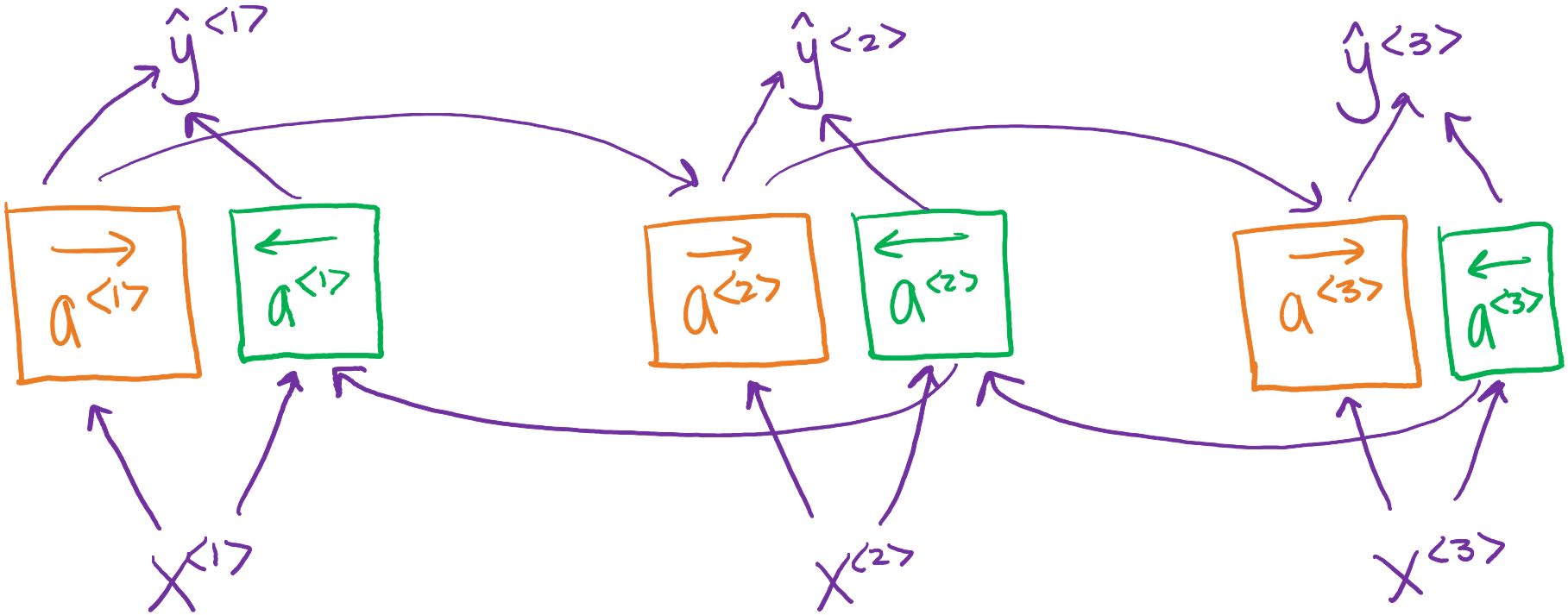
Bidirectional RNNs

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



Bidirectional RNNs



$$\hat{y}^{<t>} = g(W_y[\vec{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y)$$

Temperature Forecasting with Antichronological RNN

```
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
```

only this line changes - reverses the order

Temperature Forecasting with Antichronological RNN

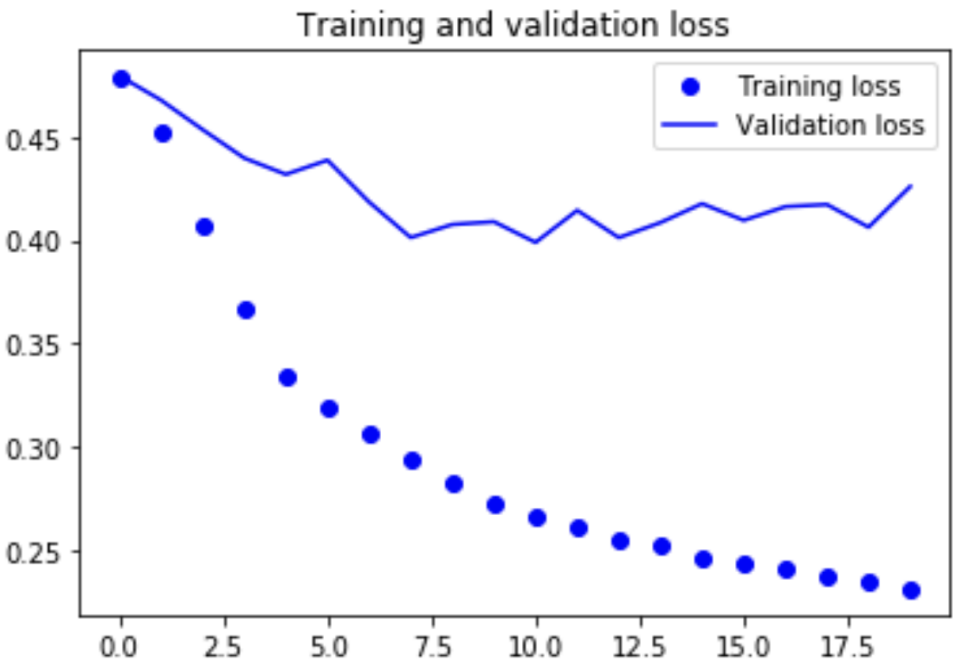
```
train_gen_reverse = reverse_order_generator(
    float_data,
    lookback=lookback,
    delay=delay,
    min_index=0,
    max_index=200000,
    shuffle=True,
    step=step,
    batch_size=batch_size)
val_gen_reverse = reverse_order_generator(
    float_data,
    lookback=lookback,
    delay=delay,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

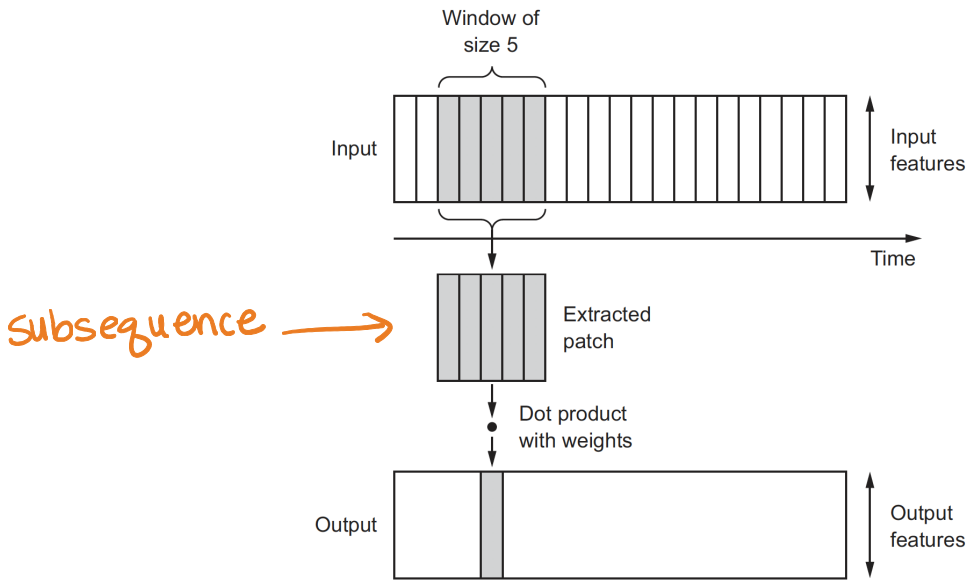
- This model performs about as well as the regular `GRU` layer (Loss ≈ 0.3)
- All of the predictive capacity comes from the chronological half of the network

- 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

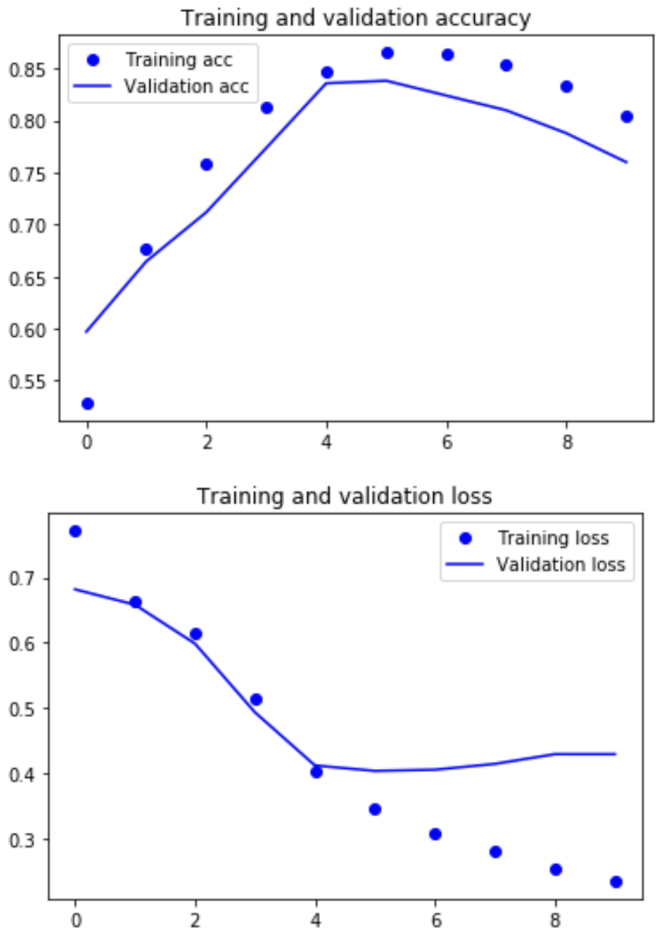
- In Keras, use the `Conv1D` layer
  - Takes as input (`samples`, `time`, `features`)
  - Returns 3D tensors
- Combine `Conv1D` layer with a `MaxPooling1D` layer
- End with a `GlobalMaxPooling` or `Flatten` layer
- Can use larger windows for 1D CNNs - typically windows of size 7 or 9
  - In a 2D convolution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

1D CNN IMDB Example

```
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



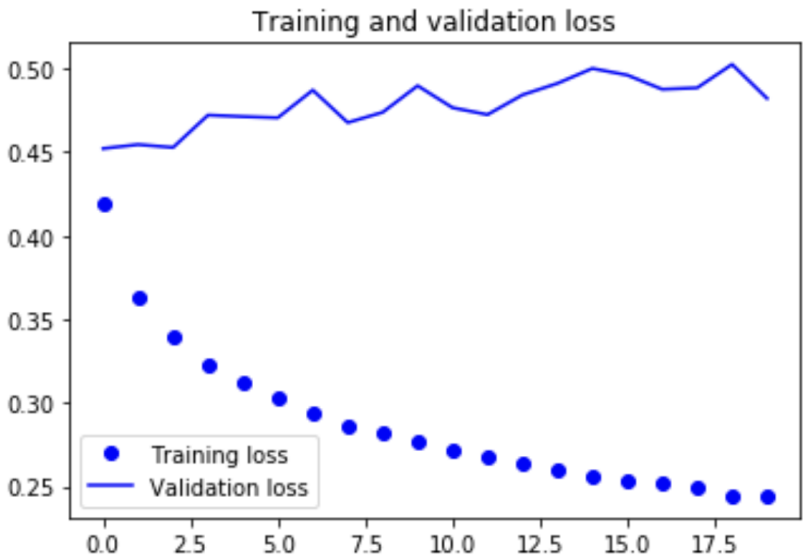
Stacking 1D CNN Layers

- 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)
```

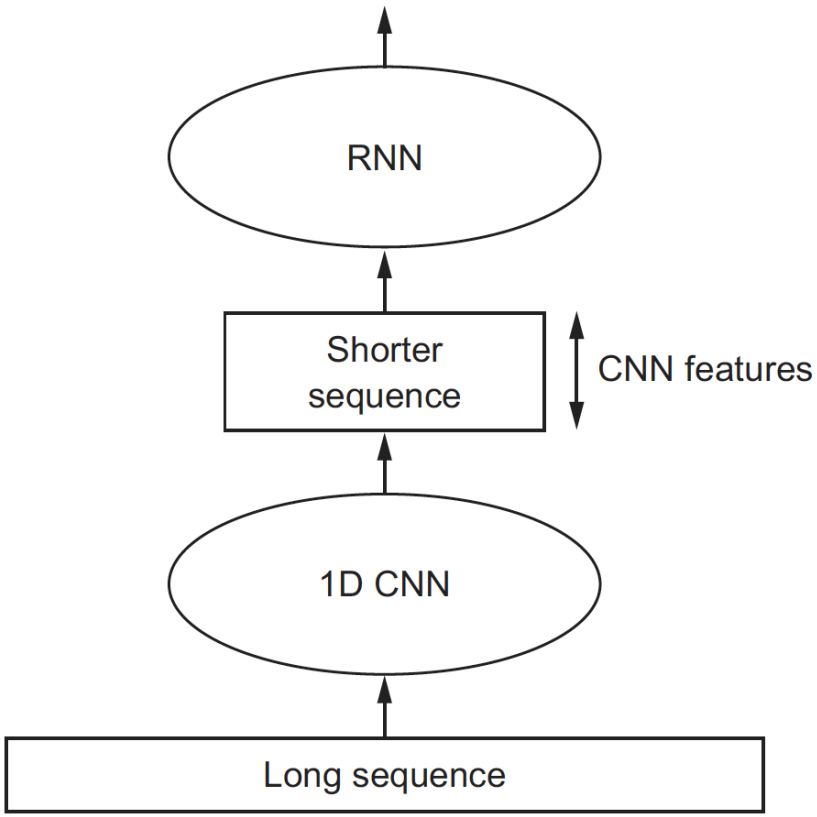
Stacking 1D CNN Layers

- 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

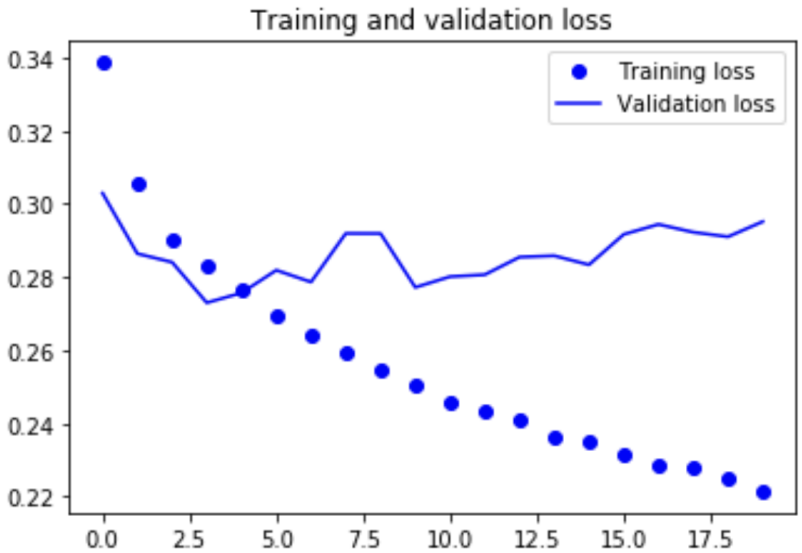
```
# This was previously set to 6 (one point per hour).
# Now 3 (one point per 30 min).
step = 3
lookback = 1440   # Unchanged
delay = 144 # Unchanged

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)
```

Temperature Forecasting with CNN and RNN

- 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



Advanced Network Architecture

Batch Normalization

- **Normalization** is a category of methods that aim to make different samples similar to each other, which helps the model learn and generalize well to new data
- Most common method is centering around 0 and scaling the standard deviation to be 1
  - This makes the assumption that the data follows a Normal distribution
- **Batch Normalization** is a type of layer that was introduced in 2015 by Sergey Ioffe and Christian Szegedy `https://arxiv.org/pdf/1502.03167v3.pdf`
  - Adaptively normalizes data as the mean and variance change over time during training
  - Internally maintains an exponential moving average of the batch-wise mean and variance of the data seen during training
  - Helps with gradient propagation and allows for deeper networks
  - Used in many advanced architecture, e.g. ResNet50, Inception V3, etc.
  - In Keras: `model.add(layers.BatchNormalization())`

Batch Normalization Algorithm

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

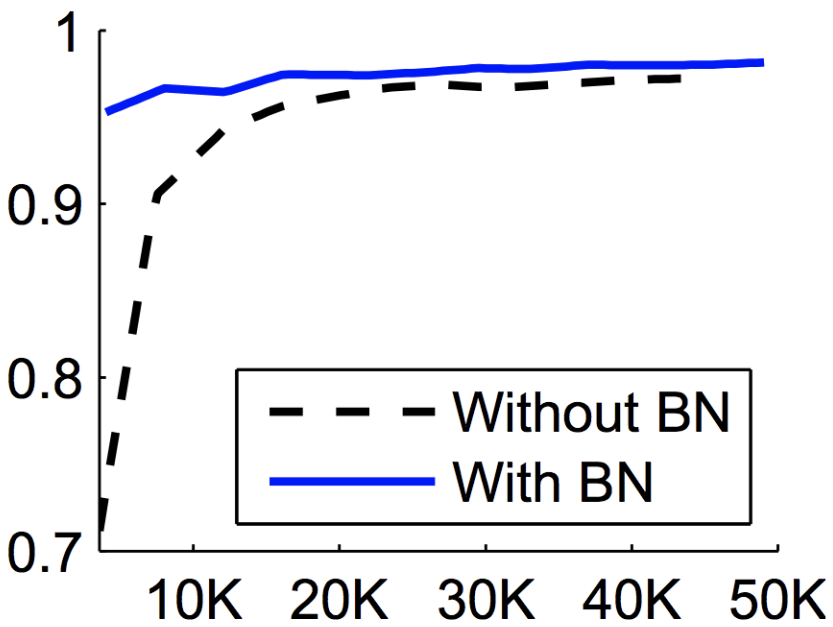
$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Batch Normalization Example

- Batch normalization used on MNIST data



Batch Normalization

- More recently (2017) **batch renormalization** was introduced by Ioffe
  - Offers benefits over batch normalization at no apparent cost
  - Hasn't supplanted batch normalization yet
- Also recently - **self-normalizing neural networks**
  - Klambauer et al (2017)
  - Keeps data normalized by using a `selu` function and a specific initializer
  - Limited to densely connected networks at the moment

Hyperparameter Optimization

- Choosing hyperparameters is an iterative process, but it can be somewhat automated
- This is an entire field of research and still in its infancy
- Need to search the architecture space and find the best-performing hyperparameters empirically
    - 1. Choose a set of hyperparameters (automatically)
    - 2. Build the corresponding model
    - 3. Fit it to your training data and measure the final performance on the validation data
    - 4. Choose the next set of hyperparameters to try (automatically)
    - 5. Repeat
    - 6. Eventually, measure performance on test data
- Key to process: use the history of validation performance given various sets of hyperparameters to choose the next set of hyperparameters to evaluate

Hyperparameter Optimization

- Many techniques exist:
  - Bayesian optimization
  - Generic algorithms
  - Simple random search
  - etc.
- Turns out that (for now), random search performs the best
- CAUTION!!
  - Can easily overfit to the validation data
  - Can be very computationally expensive

Model Ensembling

- **Ensembling** consists of pooling together the predictions of a set of different models to produce better predictions
- Ensemble models are as good or better than one model alone
- Assumes that different good models trained independently are likely to be good for different reasons - each model looks at slightly different aspects of the data to make its predictions, getting part of the 'truth', but not all of it
- SuperLearner
  - In R: `https://cran.r-project.org/web/packages/SuperLearner/SuperLearner.pdf`
  - In Python: `https://github.com/lendle/SuPyLearner`

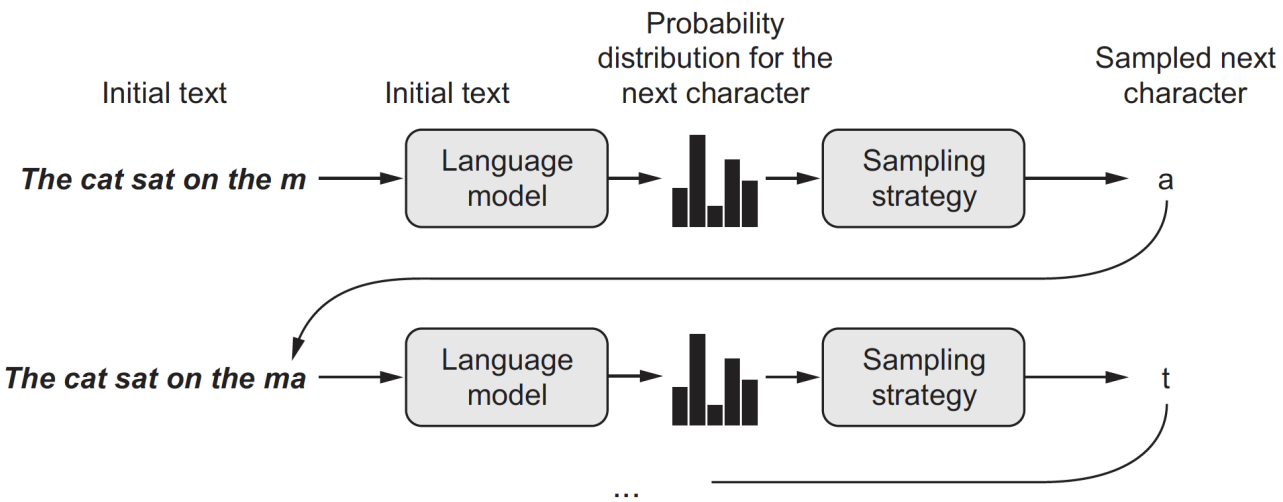
Text Generation with LSTM

Generative RNNs

- RNNs have been successfully used for:
  - Music generation
  - Dialogue generation
  - Image generation
  - Speech synthesis
  - Molecule design
- Main idea: train a model to predict the next token or next few tokens in a sequence
- **Language Model**: any network that can model the probability of the next token given the previous ones
  - Captures the latent space of language - its statistical structure
  - Once it is trained, you can sample from it to generate new sequences

Generative RNNs

- Process:
  - 1. Feed it an initial string of text (called conditioning data)
  - 2. Ask the model to generate the next character or word
  - 3. Add the generated output back to the input data
  - 4. Repeat many times



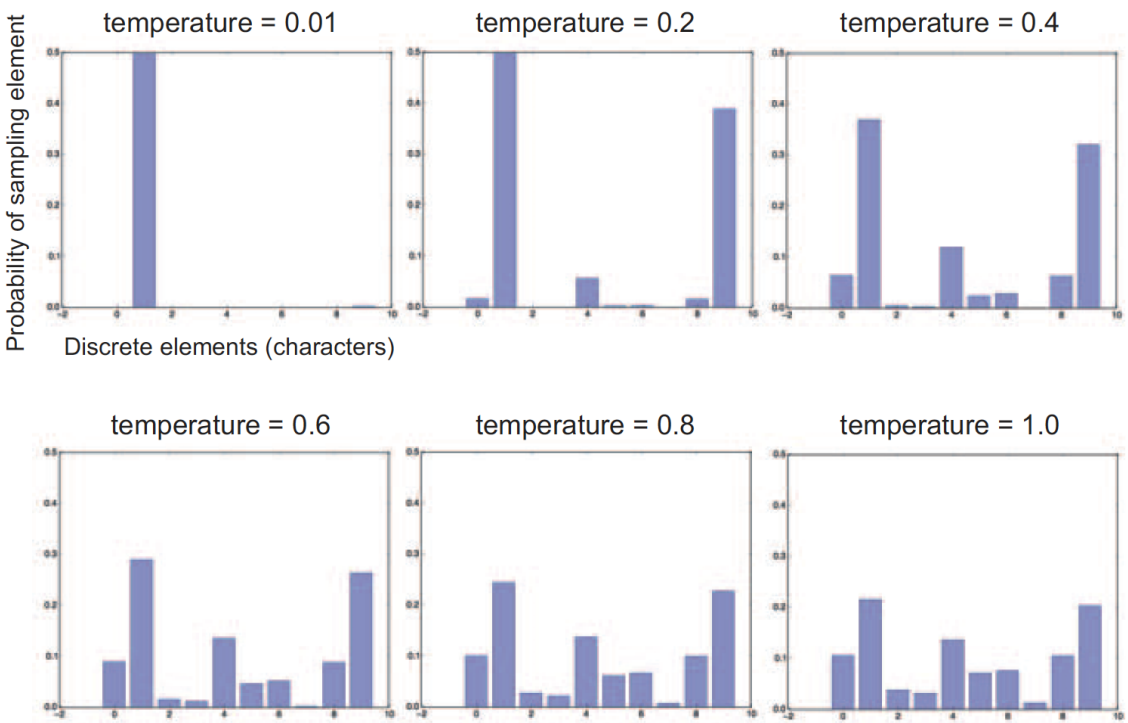
- Can generate sequences of arbitrary length
- The generated sequences will reflect the structure of the training data

Generative RNNs

- We can choose the next character or word in different ways - some are better than others
- A naive approach is ***greedy sampling*** - always choosing the most likely next character or word
  - This results in repetitive, predictable strings and not very coherent language
- Better approach is ***stochastic sampling***
  - Sample next characters or words with specific probability from a probability distribution
  - Allows even unlikely characters or words to be sampled at times, generating more interesting and creative sentences
  - Doesn't offer a way of controlling the randomness in the sampling process

Generative RNNs

- New parameter to tune: **softmax temperature**
  - Controls the amount of randomness
  - More randomness = similar probability for every character or word and results in more interesting output
  - Less randomness = higher probability for just one or a few characters or words and results in repetitive output
  - Can change the amount of randomness via the **temperature** value
    - Higher temperature = more randomness
    - Lower temperature = more deterministic



Character-level LSTM Text Generation

- Need a lot of data to train from
- Can choose from many sources, referred to as a **corpus**
  - Wikipedia
  - *The Lord of the Rings*
  - The writings of Nietzsche translated into English
- Let's see an example with the writings of Nietzsche as our corpus

Character-level LSTM Text Generation

- Random seed: “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 20 with temperature = 0.2:

“ new faculty, and the jubilation reached its climax when kant and such a man in the same time the spirit of the surely and the such the such as a man is the sunligh and subject the present to the superiority of the special pain the most man and strange the subjection of the special conscience the special and nature and such men the subjection of the special men, the most surely the subjection of the special intellect of the subjection of the same things and"

Character-level LSTM Text Generation

- Random seed: “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 20 with temperature = 0.5:

“new faculty, and the jubilation reached its climax when kant in the eterned and such man as it’s also become himself the condition of the experience of off the basis the superiory and the special morty of the strength, in the langus, as which the same time life and "even who discless the mankind, with a subject and fact all you have to be the stand and lave no comes a troveration of the man and surely the conscience the superiority, and when one must be w "

Character-level LSTM Text Generation

- Random seed: “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 20 with temperature = 1.0:

“new faculty, and the jubilation reached its climax when kant, as a periliting of manner to all definites and transpects it it so hicable and ont him artiar resull too such as if ever the proping to makes as cnecience. to been juden, all every could coldiciousnike hother aw passife, the plies like which might thiod was account, indifferent germin, that everythery certain destrution, intellect into the deteriorablen origin of moralian, and a lessority o"

Character-level LSTM Text Generation

- Random seed: “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 60 with temperature = 0.2:

“cheerfulness, friendliness and kindness of a heart are the sense of the spirit is a man with the sense of the sense of the world of the self-end and self-concerning the subjection of the strengthorixes–the subjection of the subjection of the subjection of the self-concerning the feelings in the superiority in the subjection of the subjection of the spirit isn’t to be a man of the sense of the subjection and said to the strength of the sense of the"

Character-level LSTM Text Generation

- Random seed: “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 60 with temperature = 0.5:

“cheerfulness, friendliness and kindness of a heart are the part of the soul who have been the art of the philosophers, and which the one won’t say, which is it the higher the and with religion of the frences. the life of the spirit among the most continuess of the strengther of the sense the conscience of men of precisely before enough presumption, and can mankind, and something the conceptions, the subjection of the sense and suffering and the"

Character-level LSTM Text Generation

- Random seed: “new faculty, and the jubilation reached its climax when kant"
- Output at epoch 60 with temperature = 1.0:

“cheerfulness, friendliness and kindness of a heart are spiritual by the ciuture for the entalled is, he astraged, or errors to our you idstood–and it needs, to think by spars to whole the amvives of the newoatly, prefectly raals! it was name, for example but voludd atu-especity"–or rank onee, or even all "solett increessic of the world and implussional tragedy experience, transf, or insiderar,–must hast if desires of the strubction is be stronges"