BST 261: Data Science II
Lectures 9 & 10
Recurrent Neural Networks (RNNs) Continued

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2019

Recap from last week

- So far we have seen:
  - Deep feedforward networks (MLPs)
    - Map a fixed length **vector** to a fixed length **scalar/vector**
    - Use case: classical machine learning
  - CNNS
    - Map a fixed length **matrix/tensor** to a fixed length **scalar/vector**
    - Use case: image recognition

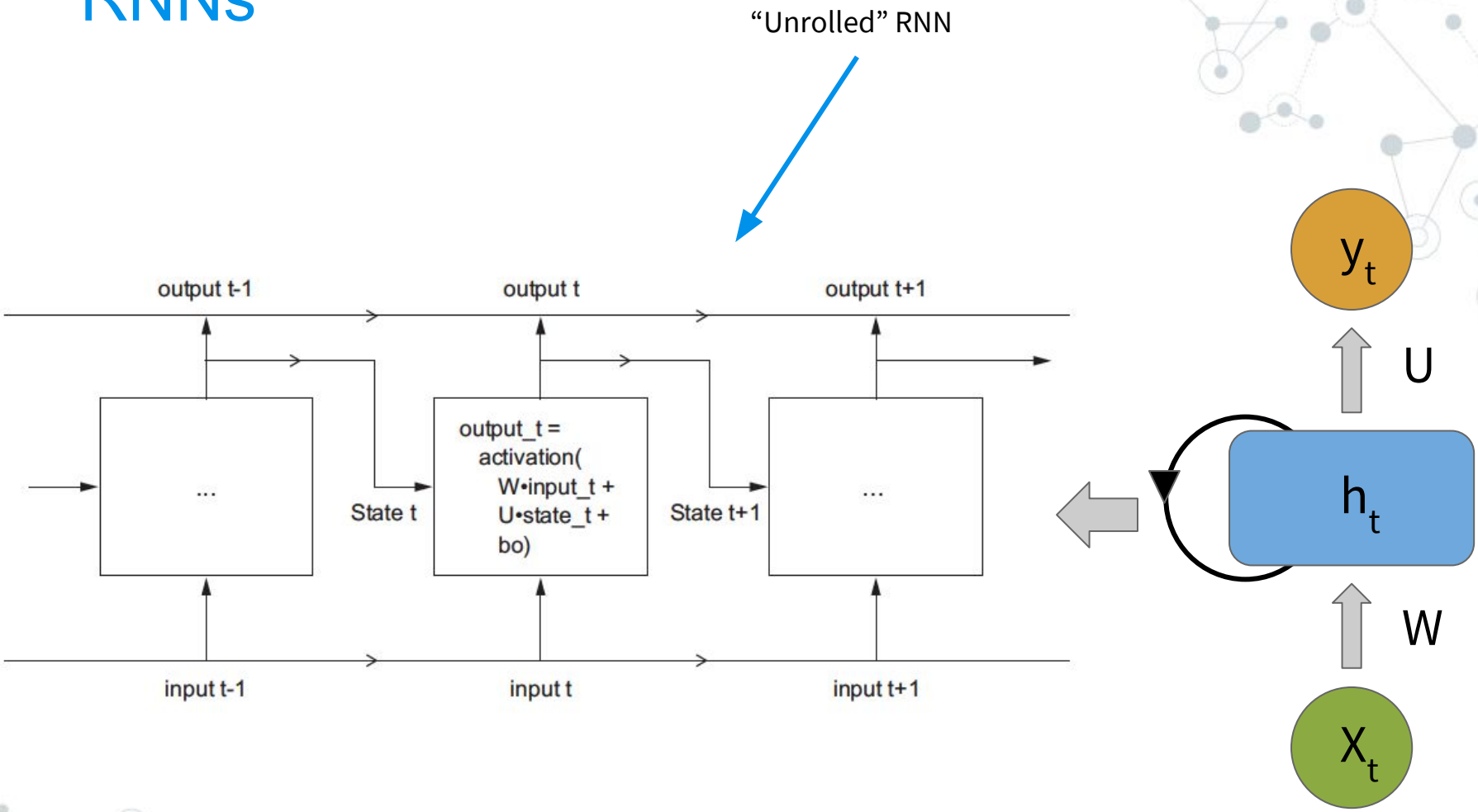
- RNNs
  - Map a **sequence** of **matrices/tensors** to a **scalar/vector**
  - Map a **sequence** to a **sequence**
  - Use case: natural language processing (NLP)

MLPs ⟹ RNNs

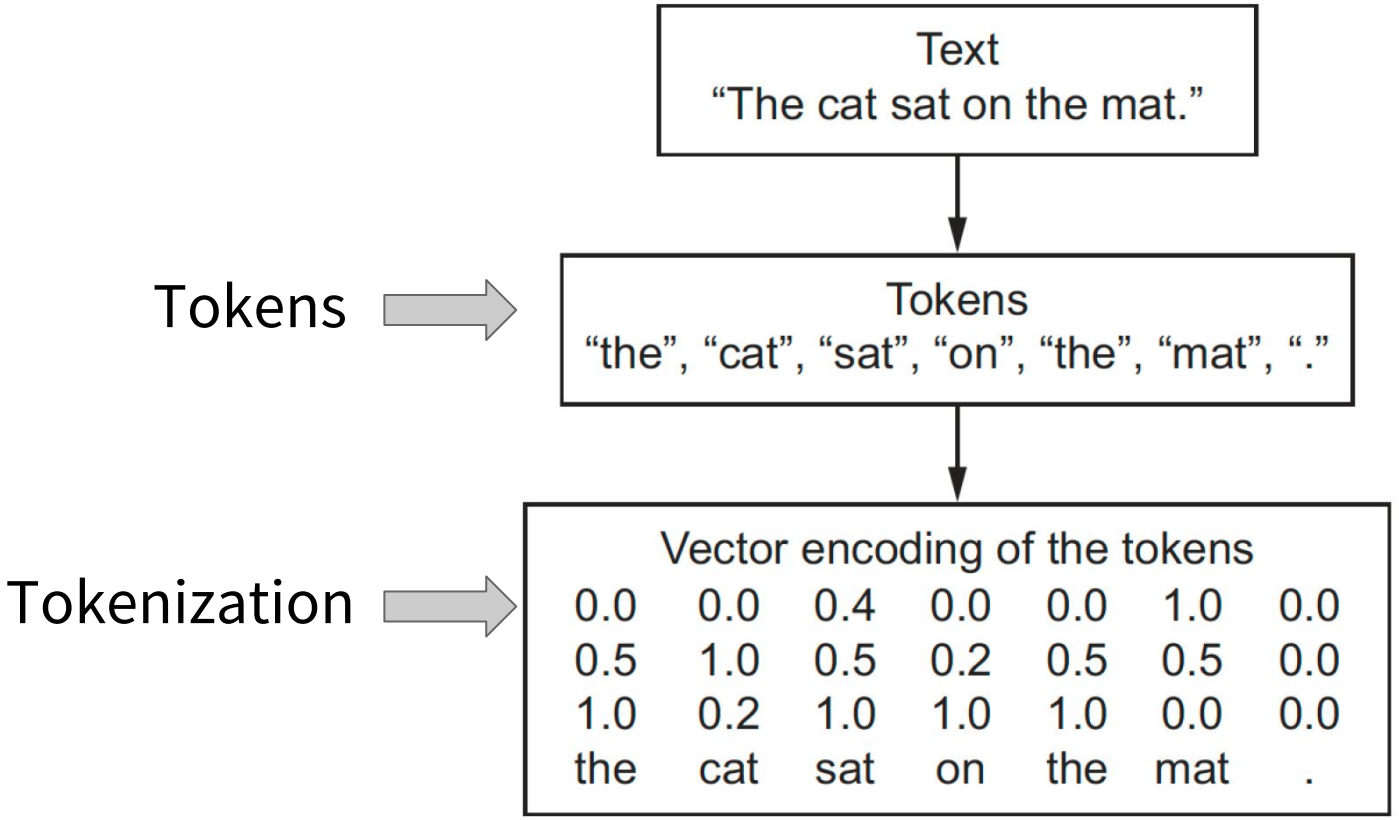
- RNNs are a natural extension of MLPs
- MLPs are “memoryless”, but often we need knowledge of the past sequence of events to predict the future

	Inputs		Outputs	Probability
MLP	X	⟹	y	$P(y\|X)$
RNN	$[x_1, x_2, x_3, \ldots, x_t]$	⟹	y	$P(y\|x_1, x_2, x_3, \ldots, x_t)$

RNNs


"Unrolled" RNN
output t-1
output t
output t+1
...
State t
output_t =
activation(
W•input_t +
U•state_t +
bo)
State t+1
...
input t-1
input t
input t+1
y_t
U
h_t
W
x_t

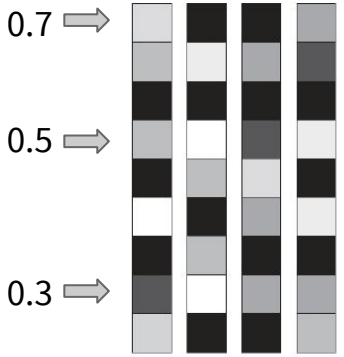
Tokenization



Word Embeddings

- Another common and powerful way to associate a vector with a word is the use of **dense word vectors** or **word embeddings**
- Word embeddings are dense, low-dimensional floating-point vectors
- Are learned from the data rather than hard coded
- 256, 512 and 1024-dimensional word embeddings are common

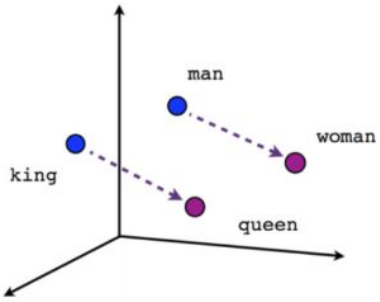
One-hot word vectors:
- Sparse
- High-dimensional
- Hardcoded



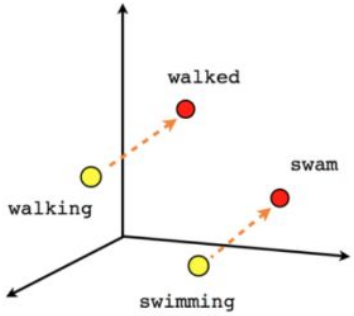
Word embeddings:
- Dense
- Lower-dimensional
- Learned from data

Learning Word Embeddings

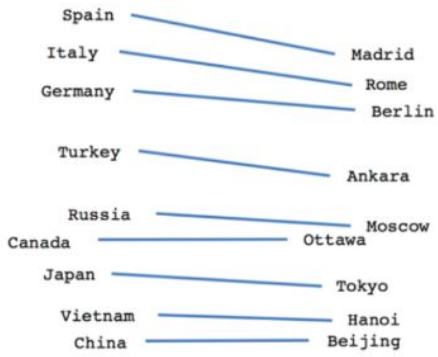
- It's easy to simply associate a vector with a word randomly - but this results in an embedding space without structure, and things like synonyms that could be interchangeable will have completely different embeddings
- This makes it difficult for a deep neural network to make sense of these representations
- It is better for similar words to have similar embeddings, and dissimilar words to have dissimilar embeddings
- We can, for example, relate the L2 distance to the similarity of the words with a smaller distance meaning the words are similar and bigger distances indicating very different words



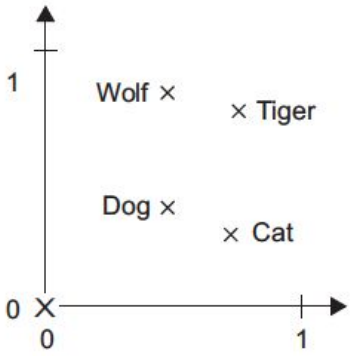
Male-Female



Verb tense



Country-Capital



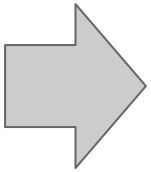
https://cbail.github.io/textasdata/word2vec/rmarkdown/word2vec.html

Pre-trained Word Embeddings

- Similar to using pre-trained convolutional bases, we can use pre-trained word embeddings
- Particularly useful when your sample size is small
- Load embedding vectors from a precomputed embedding space that is highly structured with useful properties
  - Captures generic aspects of language structure
- These embeddings are typically computed using **word-occurrence statistics**:
  - observations about what words co-occur in sentences or documents
- Various word-embedding methods exist:
  - **Word2vec** algorithm (developed by Tomas Mikolov at Google in 2013)
  - **GloVe**: Global Vectors for Word Representation (developed by researchers at Stanford in 2014)
  - Both embeddings can be used in Keras

IMDb Example

Tokenization

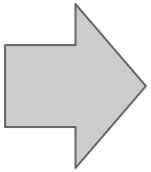


Review 1: “This movie was great!” [5, 6, 11, 32]
Review 2: “This movie was so bad I quit after ten minutes.” [5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]
Review 3: “The setting is enchanting and captivating.” [31, 12, 2, 77, 33, 78]

IMDb Example

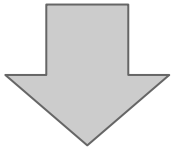
Review 1: “This movie was great!”
Review 2: “This movie was so bad I quit after ten minutes.”
Review 3: “The setting is enchanting and captivating.”

Tokenization



[5, 6, 11, 32]
[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]
[31, 12, 2, 77, 33, 78]

Padding



[5, 6, 11, 32, 0, 0, 0, 0, 0, 0]
[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]
[31, 12, 2, 77, 33, 78, 0, 0, 0, 0]

IMDb Example

Review 1: “This movie was great!”
Review 2: “This movie was so bad I quit after ten minutes.”
Review 3: “The setting is enchanting and captivating.”

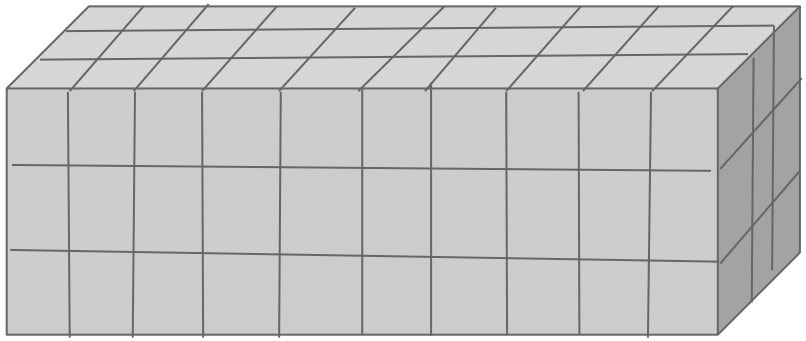
Tokenization

[5, 6, 11, 32]
[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]
[31, 12, 2, 77, 33, 78]

Padding

[5, 6, 11, 32, 0, 0, 0, 0, 0, 0]
[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]
[31, 12, 2, 77, 33, 78, 0, 0, 0, 0]

Embedding



Each word is represented by a vector with 3 elements

IMDb Example

Review 1: “This movie was great!”
Review 2: “This movie was so bad I quit after ten minutes.”
Review 3: “The setting is enchanting and captivating.”

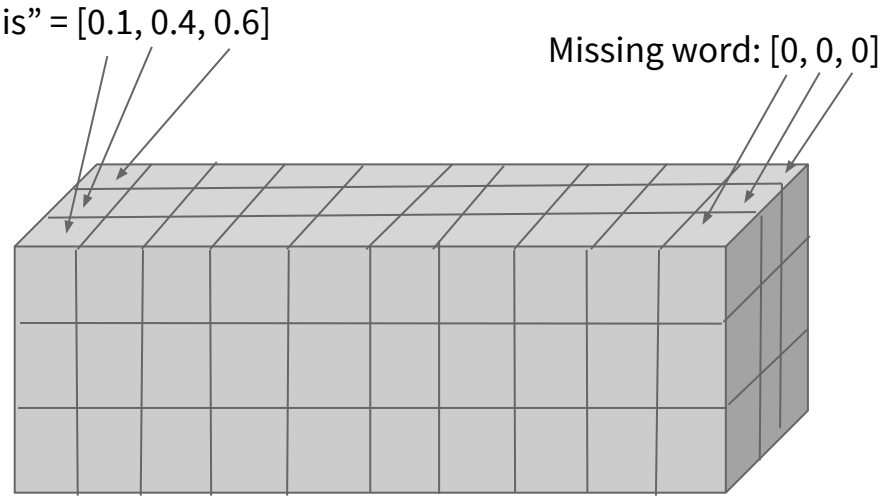
Tokenization

[5, 6, 11, 32]
[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]
[31, 12, 2, 77, 33, 78]

Padding

[5, 6, 11, 32, 0, 0, 0, 0, 0, 0]
[5, 6, 11, 14, 66, 3, 49 , 55, 98, 121]
[31, 12, 2, 77, 33, 78, 0, 0, 0, 0]

Embedding



Each word is represented by a vector with 3 elements. The input is now a 3D tensor of shape (3, 10, 3)

Number of reviews

Length of each review

Depth of word embedding: how many numbers represent a word

IMDb Example

```
from keras.datasets import imdb
from keras.preprocessing import sequence

max_features = 10000  # number of words to consider as features
maxlen = 500  # cut texts after this number of words (among top max_features most common words)
batch_size = 32

print('Loading data...')
(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)
print(len(input_train), 'train sequences')
print(len(input_test), 'test sequences')

print('Pad sequences (samples x time)')
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
print('input_train shape:', input_train.shape)
print('input_test shape:', input_test.shape)
```

```
Loading data...
25000 train sequences
25000 test sequences
Pad sequences (samples x time)
input_train shape: (25000, 500)
input_test shape: (25000, 500)
```

Let's train a simple recurrent network using an `Embedding` layer and a `SimpleRNN` layer:

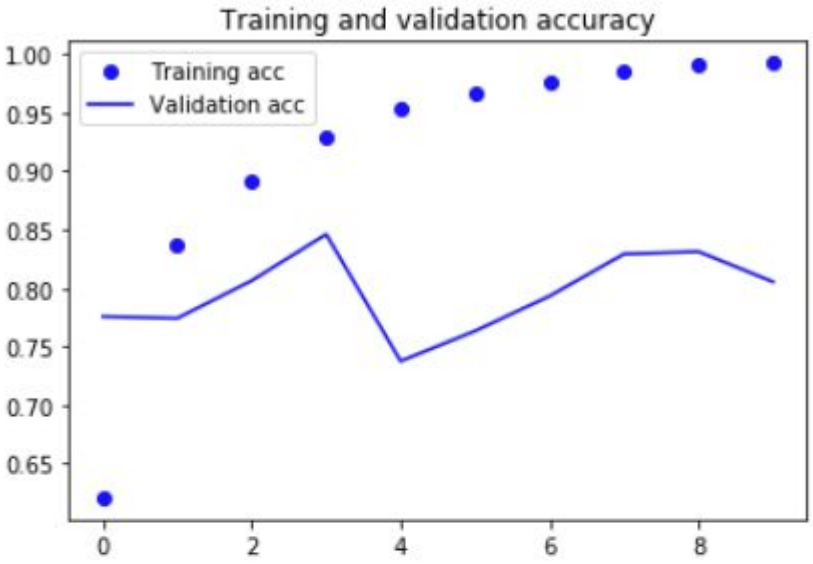
```
from keras.layers import Dense

model = Sequential()
model.add(Embedding(max_features, 32))
model.add(SimpleRNN(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)
```

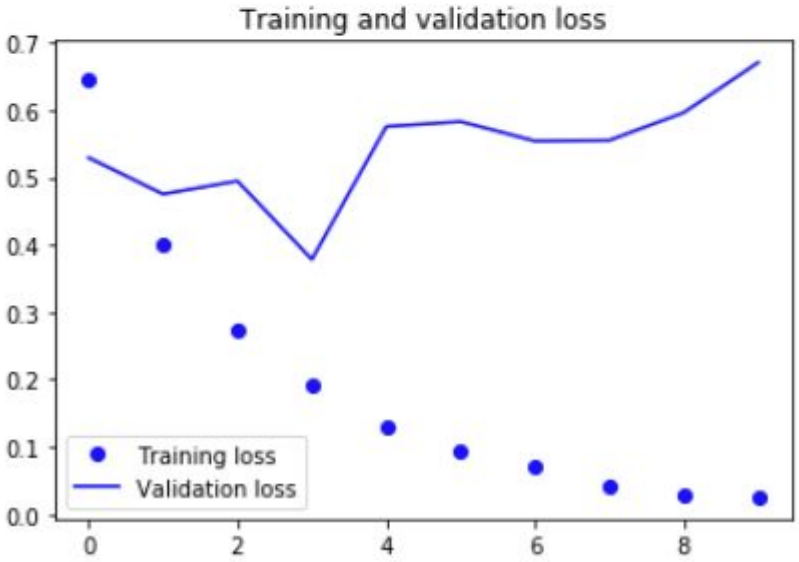
The number of total possible words

How many numbers to use to represent a word

As a reminder, in lecture 3, our very first naive approach to this very dataset got us to 88% test accuracy. Our small recurrent network doesn't perform very well at all compared to this baseline (only up to 85% validation accuracy). Part of the problem is that our inputs only consider the first 500 words rather the full sequences -- hence our RNN has access to less information than our earlier baseline model.

The remainder of the problem is simply that SimpleRNN isn't very good at processing long sequences, like text. Other types of recurrent layers perform much better. We'll talk about these next.





Long Short Term Memory (LSTM)

Problems with RNNs

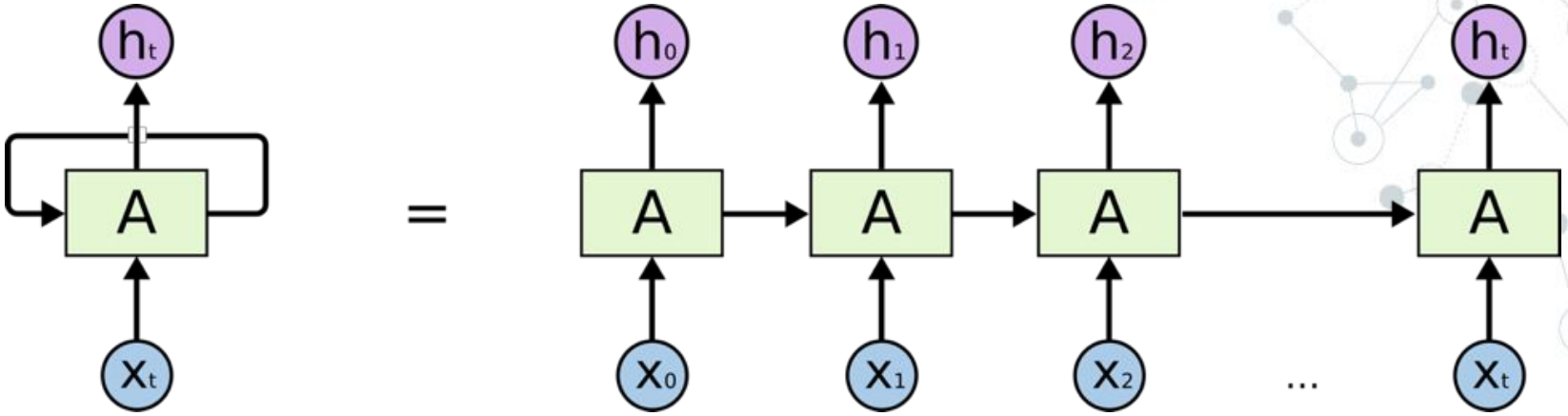
- Recall the formula for a generic RNN:

$$h_t = f(X_t W + h_{t-1} U + b)$$

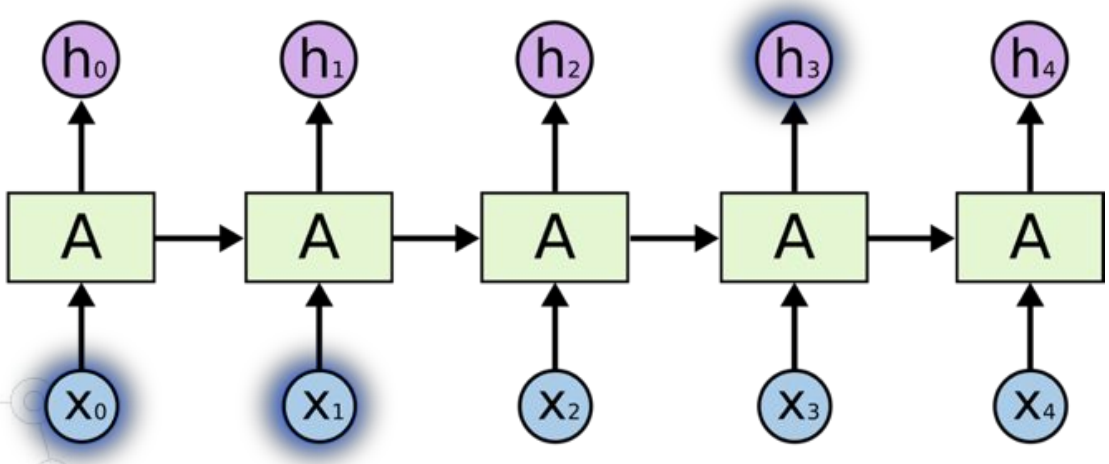
- What happens for really long sequences during backprop?
  - You multiply by the matrix U repeatedly
  - Largest eigenvalue > 1, gradient → infinity (explodes)
  - Largest eigenvalue < 1. gradient → 0 (vanishes)

- This is known as the **vanishing or exploding gradient problem**

An “unrolled” RNN

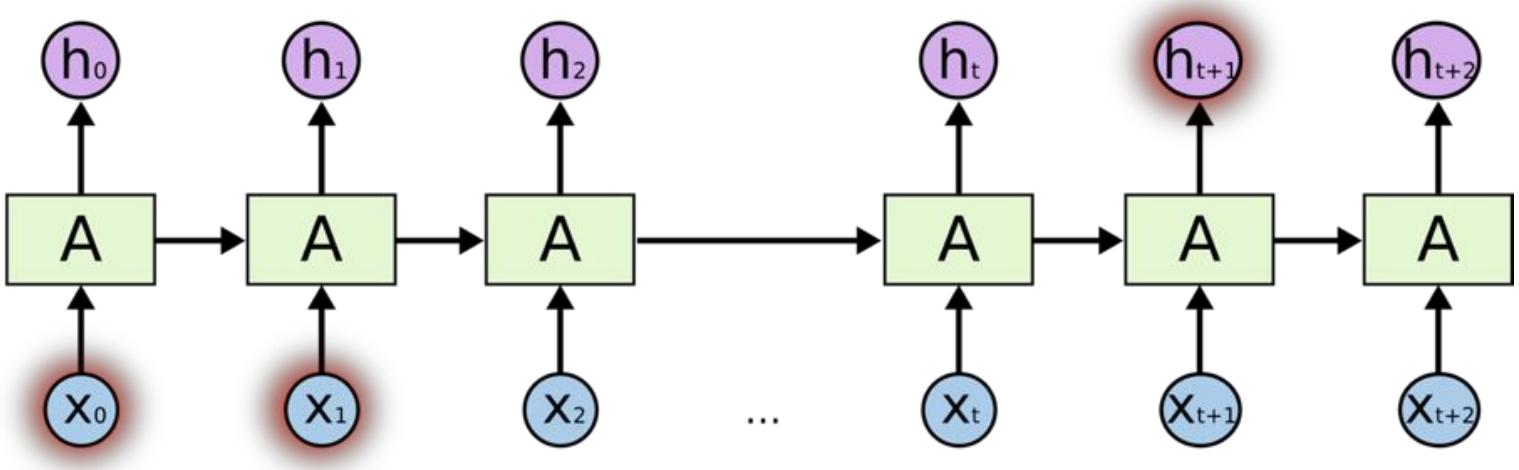


RNN where the output h_3 only depends on the input from X_0 and X_1

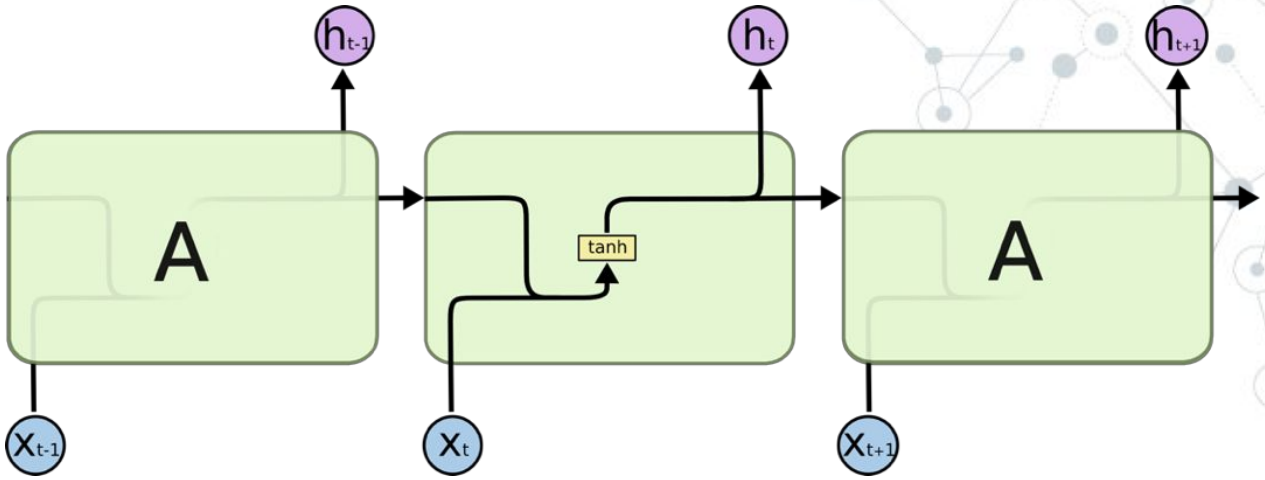


http://colah.github.io/posts/2015-08-Understanding-LSTMs/

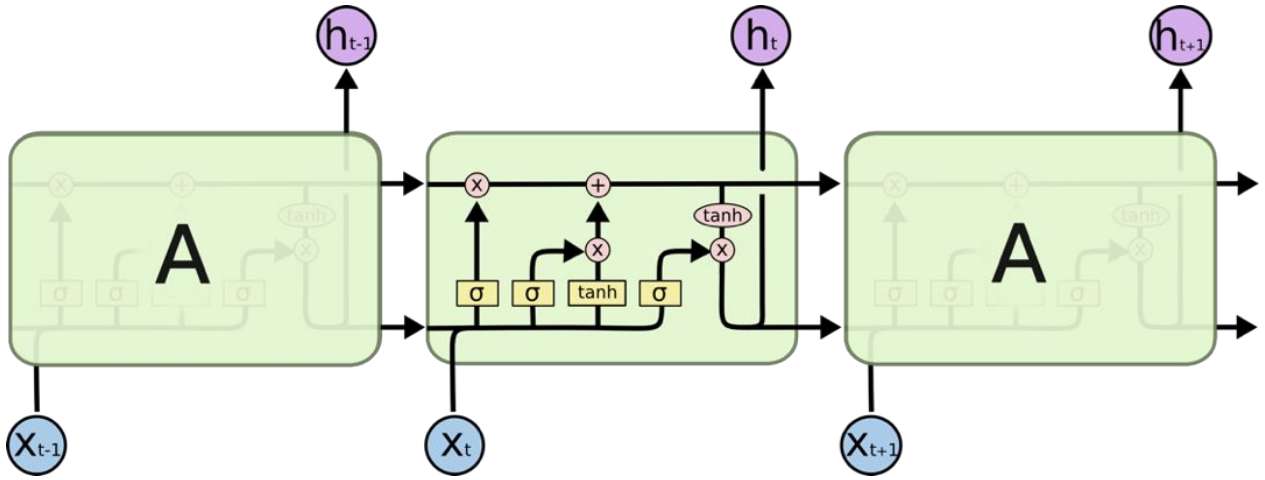
RNN where the output h_{t+1} is dependent on data inputs X_0 and X_1 that are too far for the gradient to carry



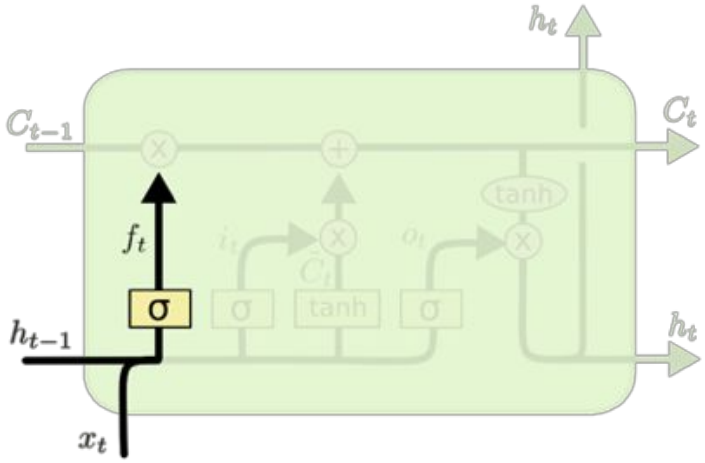
Simple, “vanilla” RNN:



RNN with LSTM units:

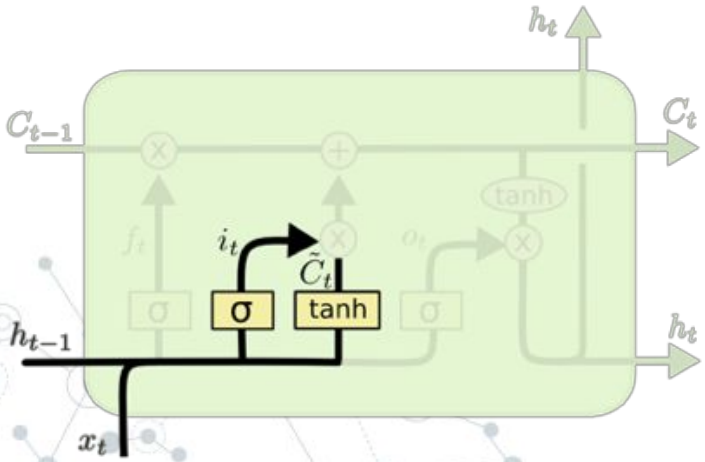


Step 1: Forget Gate - Determine how much of the previous state should affect the current state based on the current observed input x_t



$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

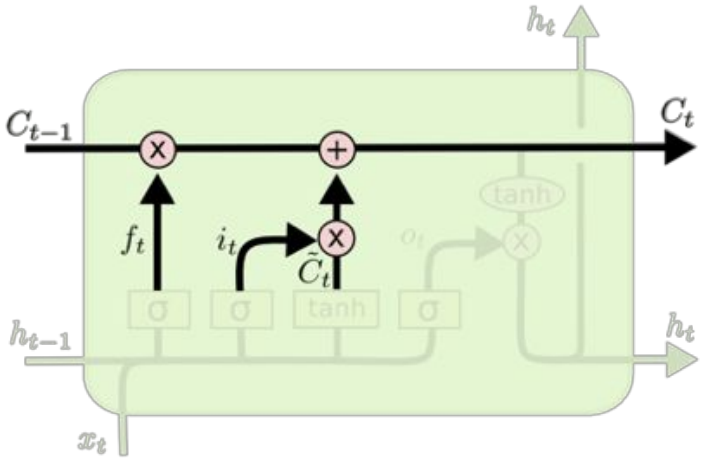
Step 2: Update Cell State - First determine which values we will update (gate i_t), then create a list of candidate values that we will add to the current state (C_t) based on the current input (x_t) and the previous output (h_{t-1}).



$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$

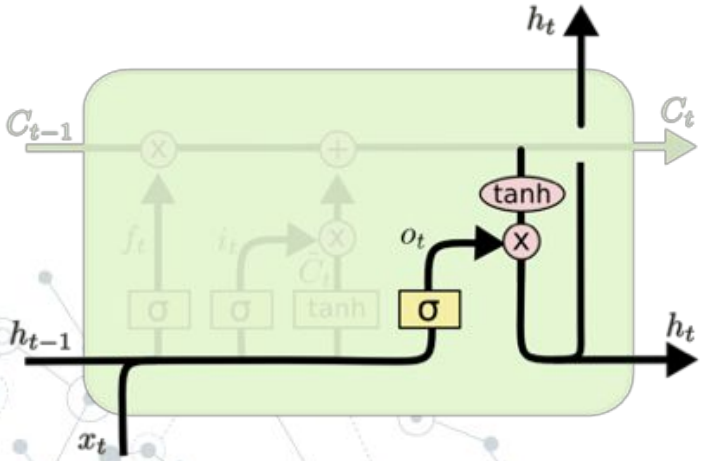
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Step 3: Execute the Update - update the amount of carried state, then add the selected new states to the retained state to create the new internal state.



$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$

Step 4: Compute Unit Output - determine which parts of the cell state will be used as unit output.



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

$$h_t = o_t * \tanh\left(C_t\right)$$

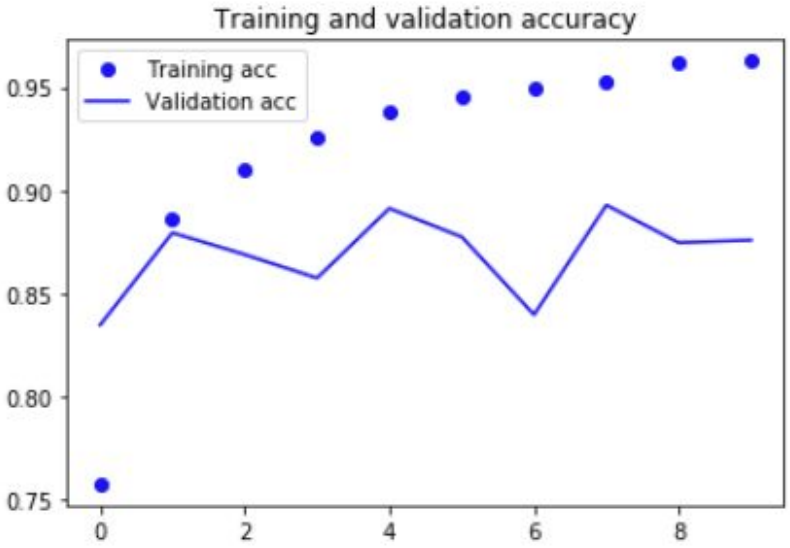
LSTM in Keras

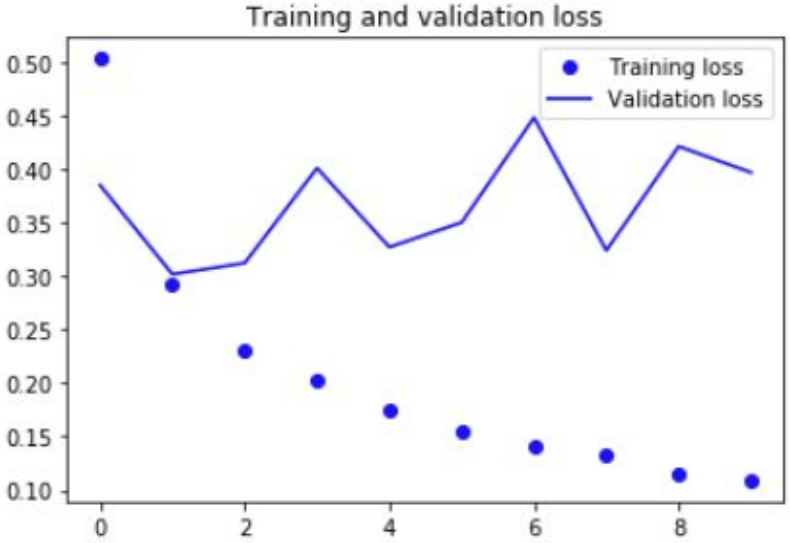
- Now that you have an idea of how LSTM works, let's implement it in Keras
- We set up a model using an LSTM layer and train it on the IMDb data
- The network is similar to the one with SimpleRNN that we discussed last week
- We only specify the output dimensionality of the LSTM layer, and leave every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

Just replace "SimpleRNN" with "LSTM"

LSTM in Keras



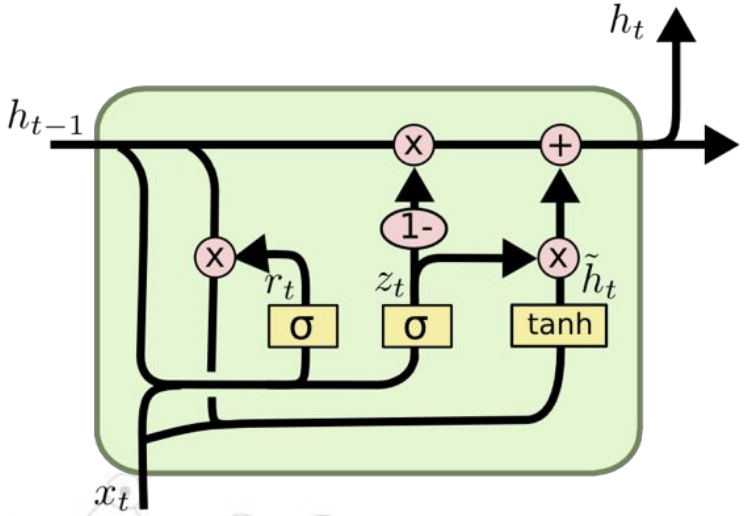


Best performance so far - high 80s in terms of accuracy %

Gated Recurrent Unit (GRU)

GRU

- Relatively new (2014)
- Combines the “forget” and “input” gates into an “update gate”
- Merges cell state and hidden state
- Performance on par with LSTM, but is computationally more efficient



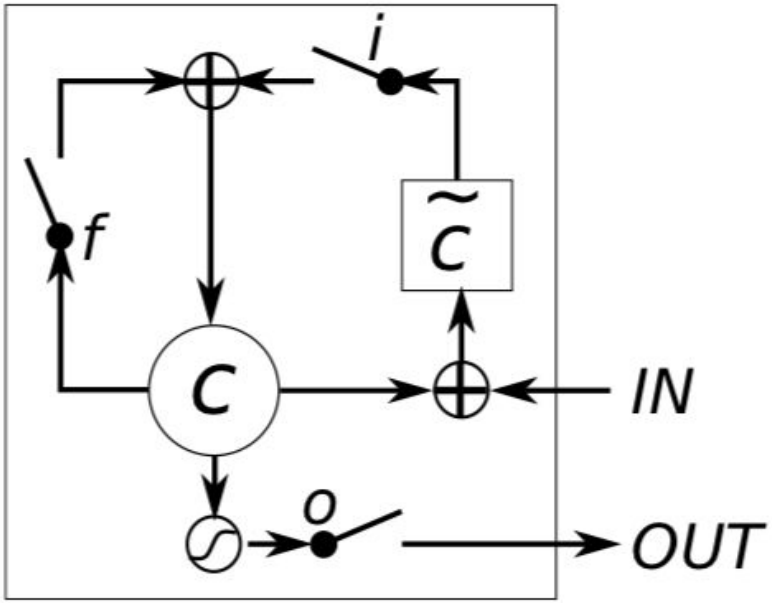
$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

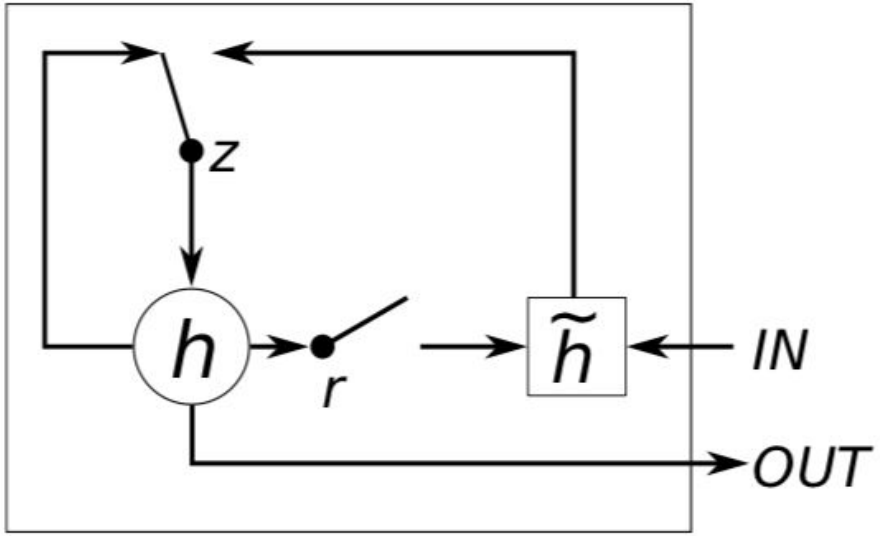
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

LSTM vs GRU



(a) Long Short-Term Memory



(b) Gated Recurrent Unit

LSTM vs GRU

- LSTM performs better on long sequences
- GRUs are faster to train
- GRUs are simpler to understand and modify

Improving RNN 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

Example: 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.
  - **1 recording every 10 minutes = 6 recordings per hour = 144 recordings per day = 52,560 recordings per year**

- 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

```
import os

fname = '/Users/heathermattie/Desktop/jena_climate_2009_2016.csv'
f = open(fname)
data = f.read()
f.close()

lines = data.split('\n')      # Each line is 1 recording
header = lines[0].split(',') # Variable names are separated by commas
lines = lines[1:]             # Drop first line (it's a header)

print(header)
print(len(lines))
```

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

Let's convert all of these 420,551 lines of data into a Numpy array:

Number of rows (observations)

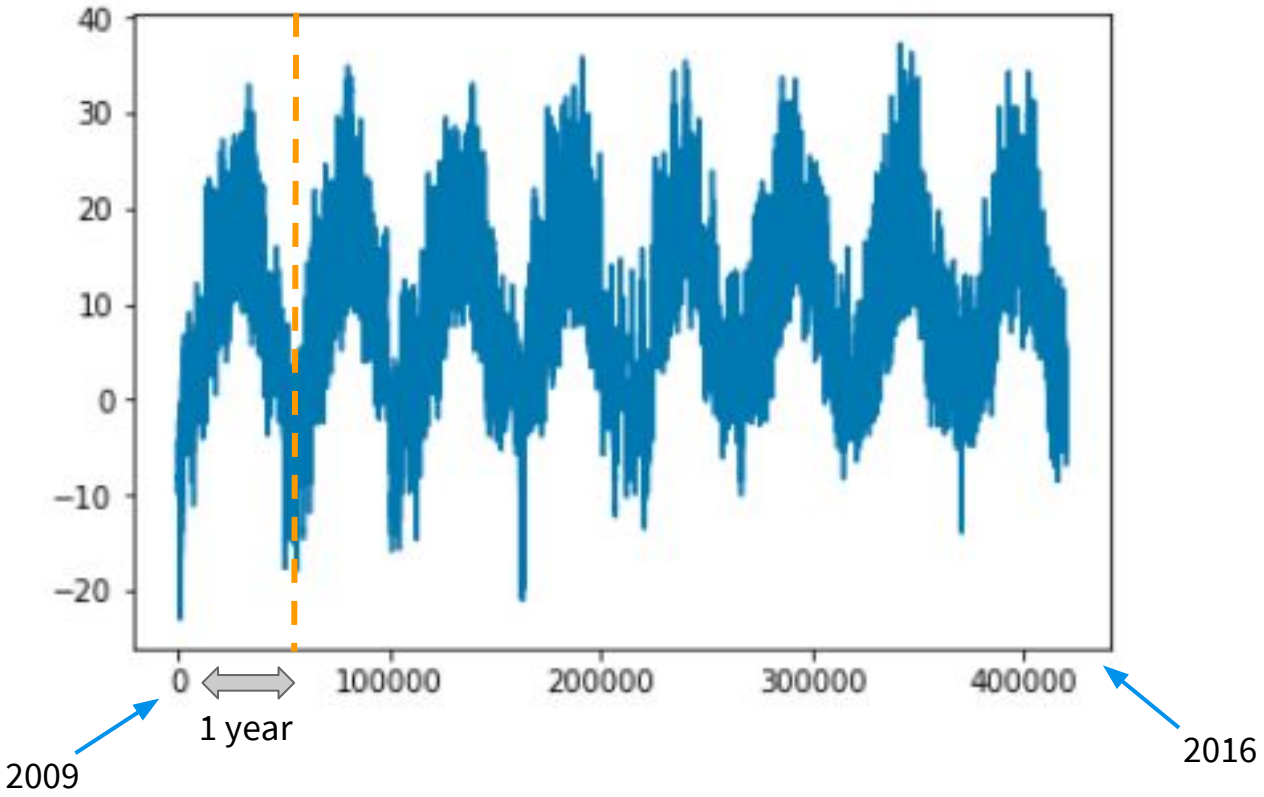
Number of columns (-1 for unnecessary 1st column: date/time)

```
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
print(float_data.shape)
```

Drop first column (the unnecessary date/time)

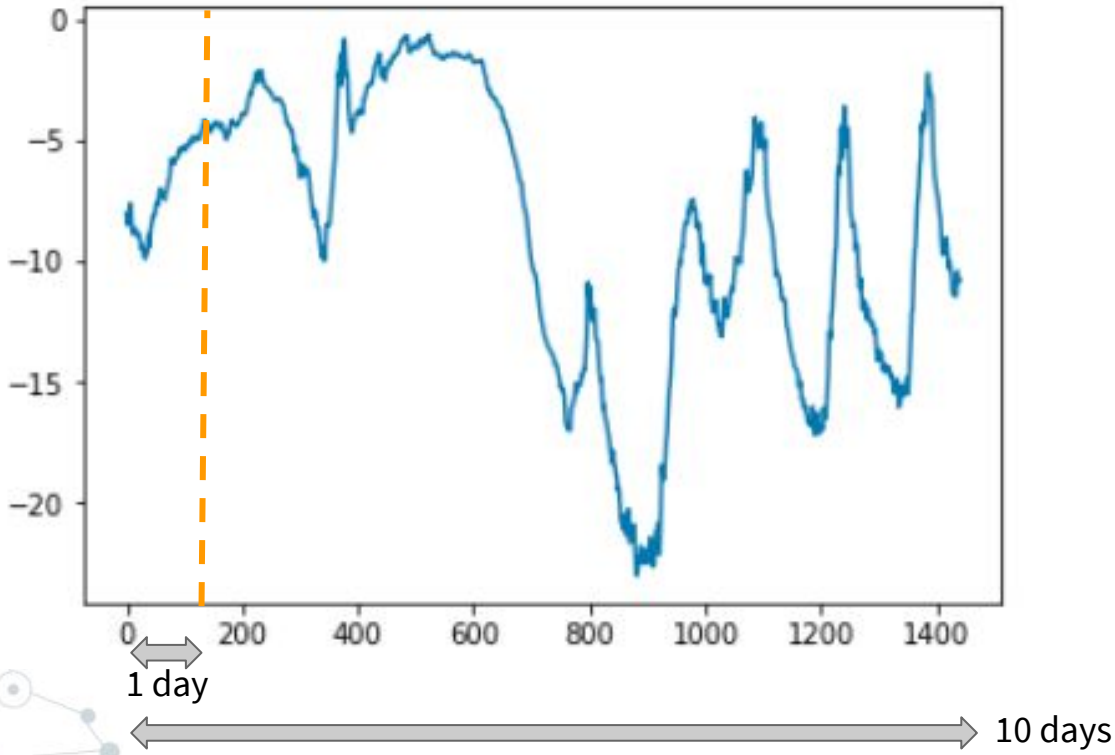
```
(420551, 14)
```

Temperature over time



Temperature over time

- Let's 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



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?

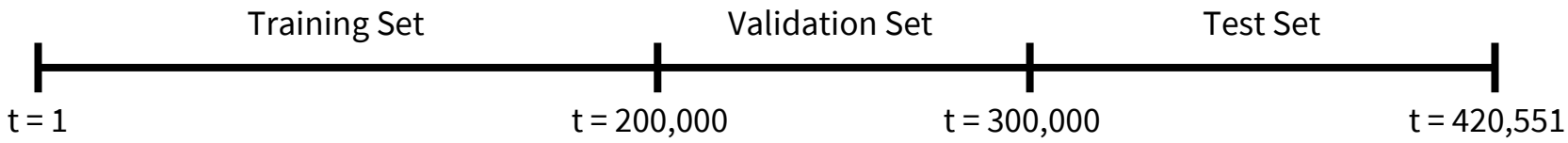
- **lookback** = 1440; we will go back 10 days
- **steps** = 6; observations will be sampled at one data point per hour - we will only take into account every 6th recording
- **delay** = 144; targets will be 24 hours in the future

- Process the data:
  - Normalize all variables to have mean 0 and sd 1

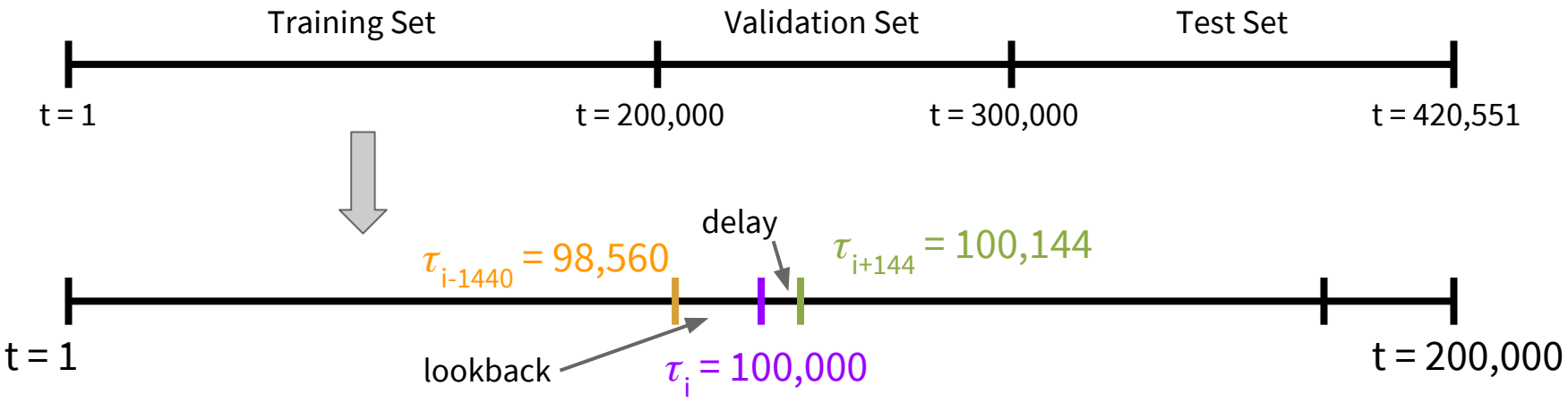
Temperature Forecasting in Keras

- 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

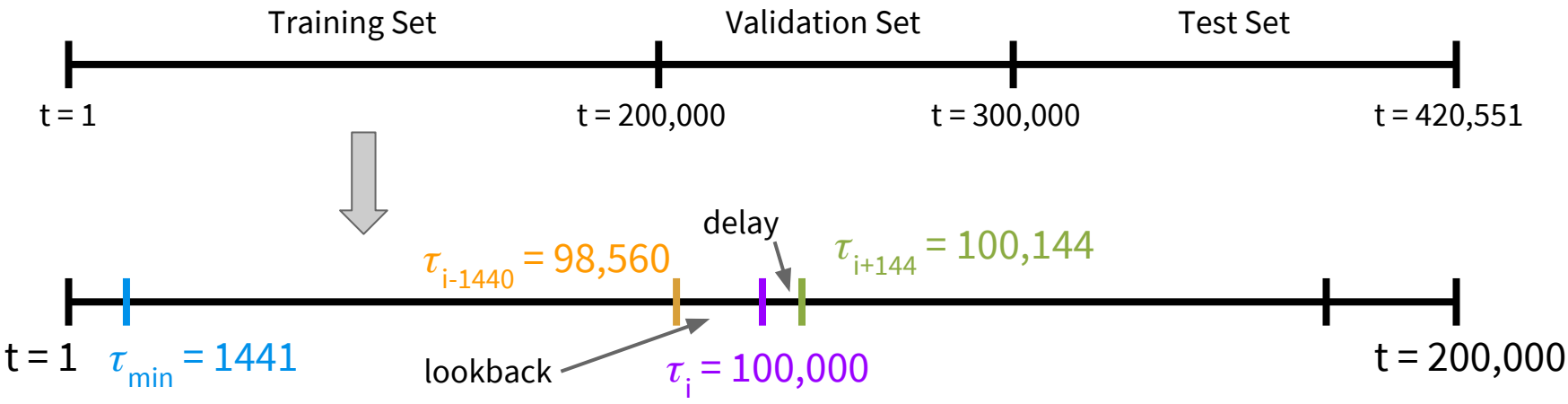
Timeline


Training Set
Validation Set
Test Set
t = 1
t = 200,000
t = 300,000
t = 420,551

Timeline

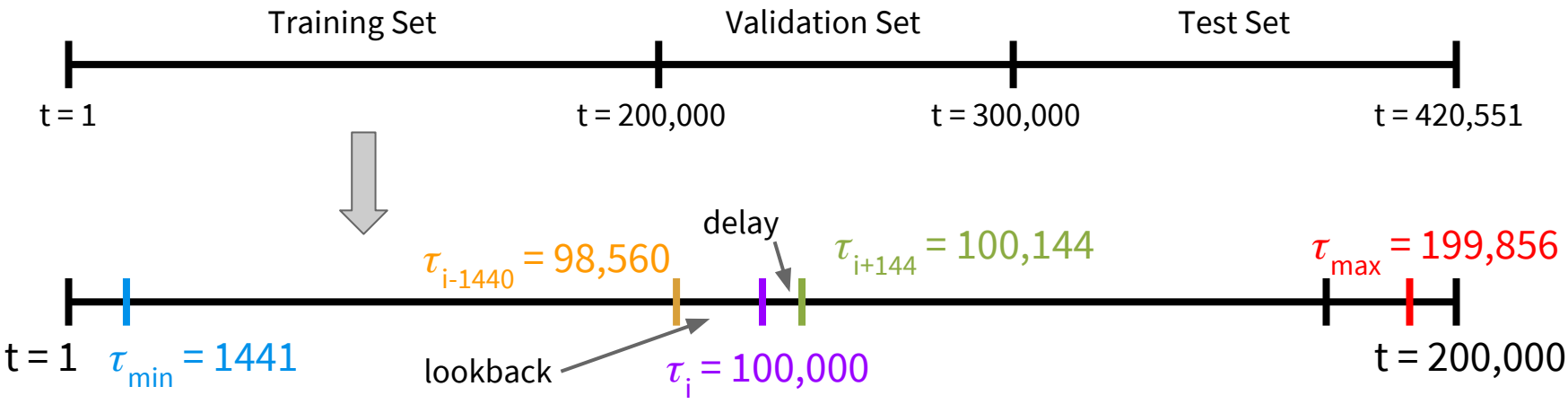

Training Set
Validation Set
Test Set
t = 1
t = 200,000
t = 300,000
t = 420,551
delay
τ_{i-1440} = 98,560
τ_{i+144} = 100,144
t = 1
lookback
τ_i = 100,000
t = 200,000

Timeline



τ_{min}: we need all 10 days worth of past data to predict the temperature for the next time point. Thus, the minimum value τ_i can take is 1400 (10 days of previous data) + 1 (time point) = 1441. If we choose τ_i < 1441, we won't have enough prior data to make a prediction.

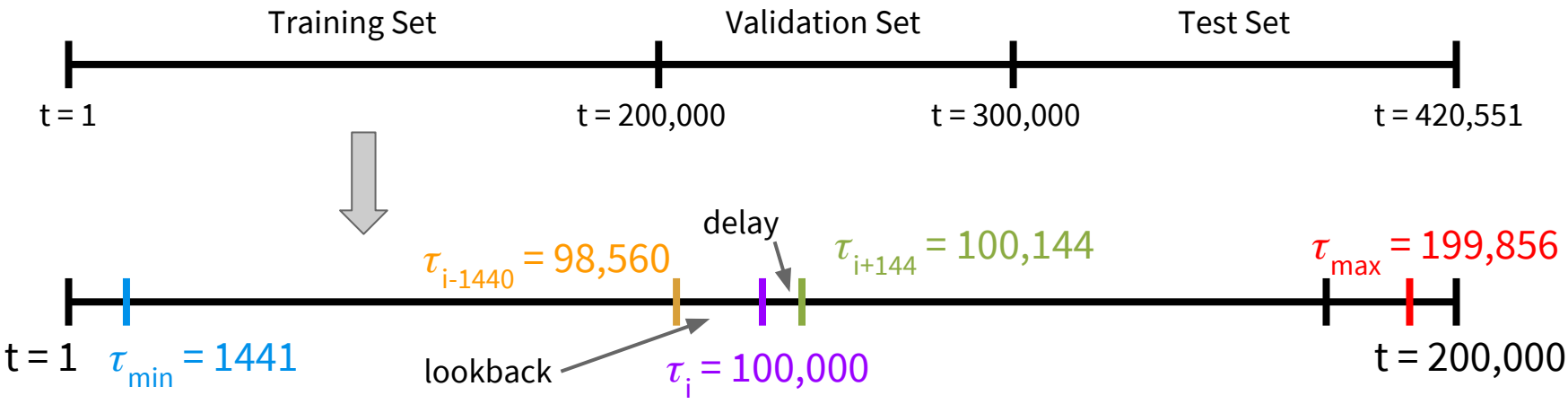
Timeline



τ_{min}: we need all 10 days worth of past data to predict the temperature for the next time point. Thus, the minimum value τ_i can take is 1400 (10 days of previous data) + 1 (time point) = 1441. If we choose τ_i < 1441, we won't have enough prior data to make a prediction.

τ_{max}: we need 24 hours worth of data after this point in order to have a point to make a prediction for. Thus, the maximum value τ_i can take is 200,000 - 144 = 199,856. If we choose τ_i >199,856, we won't have a data point to make a prediction for.

Timeline



τ_{min}: we need all 10 days worth of past data to predict the temperature for the next time point. Thus, the minimum value τ_i can take is 1400 (10 days of previous data) + 1 (time point) = 1441. If we choose τ_i < 1441, we won't have enough prior data to make a prediction.

τ_{max}: we need 24 hours worth of data after this point in order to have a point to make a prediction for. Thus, the maximum value τ_i can take is 200,000 - 144 = 199,856. If we choose τ_i >199,856, we won't have a data point to make a prediction for.

Steps:

1. Randomly sample a point in time, τ_i, between τ_{min} and τ_{max}
2. Keep 10 days of data prior to τ_i and 24 hours after τ_i.
3. Repeat this process multiple times
4. Split training examples into batches
5. Feed into the network
6. Repeat similar process for validation and test sets

Whether to shuffle points in time or not

While true, meaning while there are still examples to include in a batch

```
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
```

For the training set - randomly choose points in time

For the validation and test sets - choose batches of timesteps (in chronological order)

Floor division

One batch of input data

Corresponding target temperatures

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)
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

print(val_steps)
print(test_steps)
```

```
769
930
```

Temperature Forecasting

- 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

```
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()
```

```
0.2897359729905486
```

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℃.

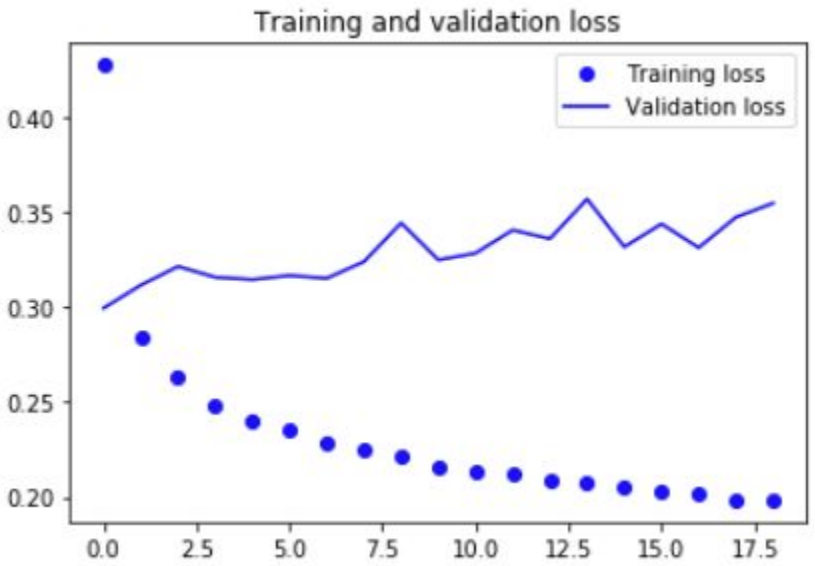
That's a fairly large average absolute error – now the task is to leverage our knowledge of deep learning to do better.

Temperature Forecasting - Simple Model

- Let's first try a simple model (MLP) 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

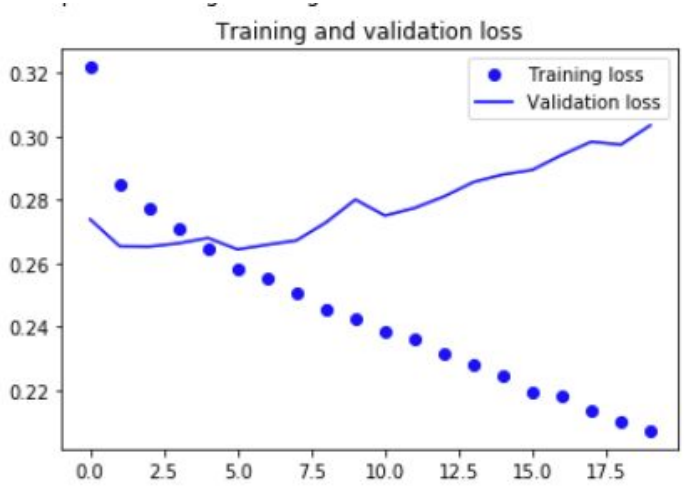


We get MAEs above 0.3 for the validation loss - worse than our benchmark.

This shows that the simple model isn't complex enough for our data and task (it's not taking time into account), and that some benchmarks can be difficult to beat.

Temperature Forecasting - RNN

```
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))
print(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)
```



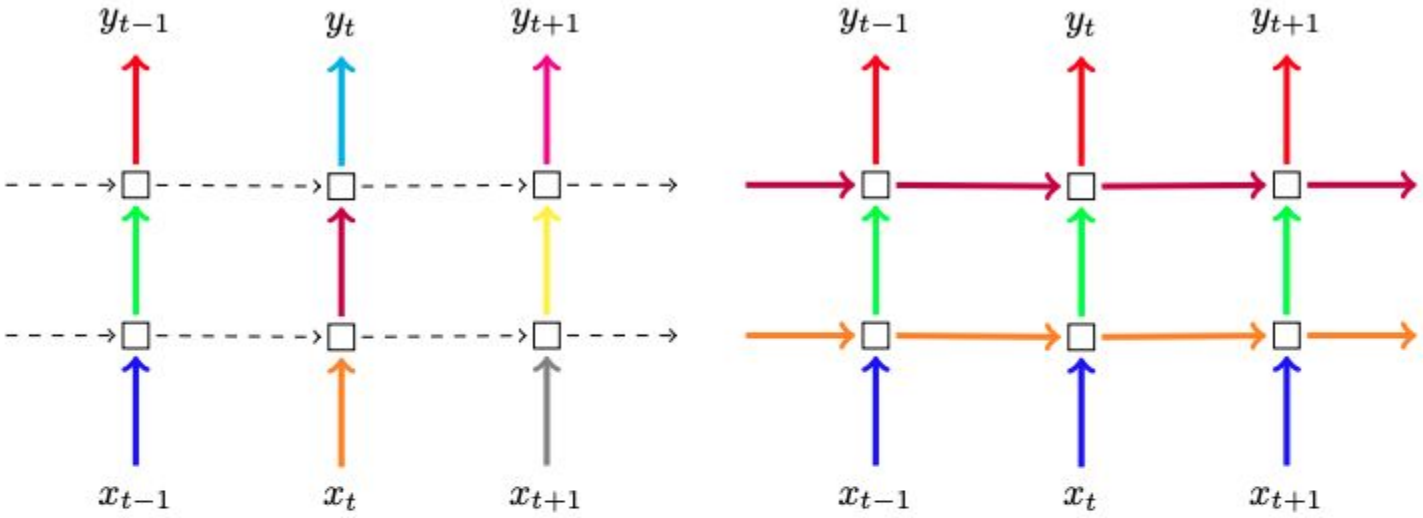
This model is better than the previous simple model and the common-sense baseline.

There is 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),"
  - **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



(a) Naive dropout RNN

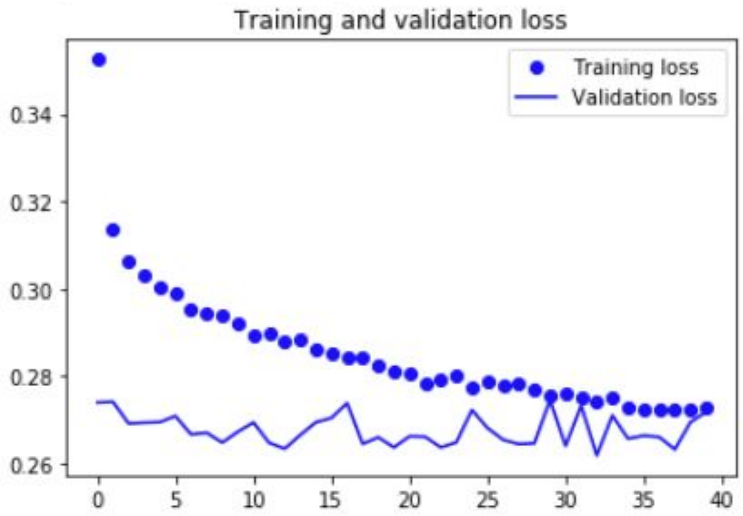
(b) Variational RNN

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.

Source: https://arxiv.org/pdf/1512.05287.pdf

Recurrent Dropout in Keras

```
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

Stacking Recurrent Layers

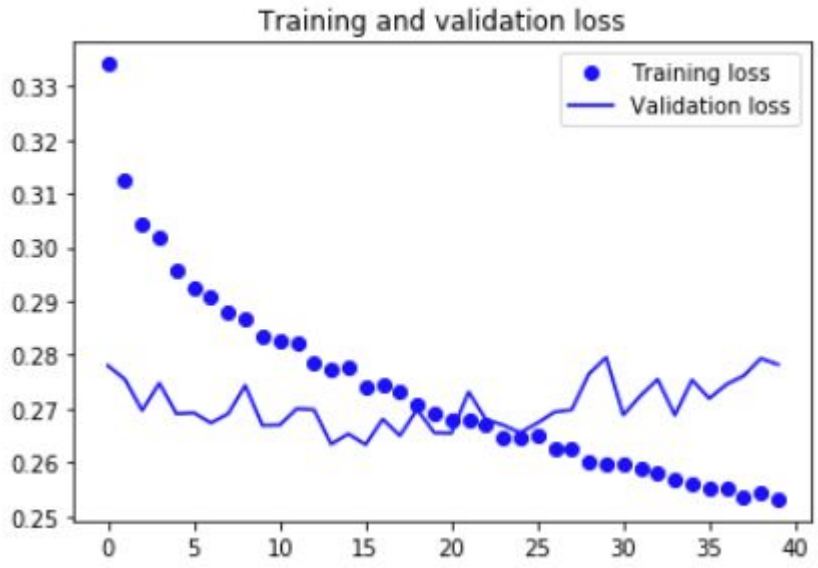
- Because we're no longer overfitting but seem to have hit a performance bottleneck, we should consider increasing the capacity of the network - make the model more complex

- Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers.

- Recurrent layer stacking is a classic way to build more-powerful recurrent networks: for instance, what currently powers the Google Translate algorithm is a stack of 7 large LSTM layers—that's huge!

Stacking Recurrent Layers in Keras

```
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)
```

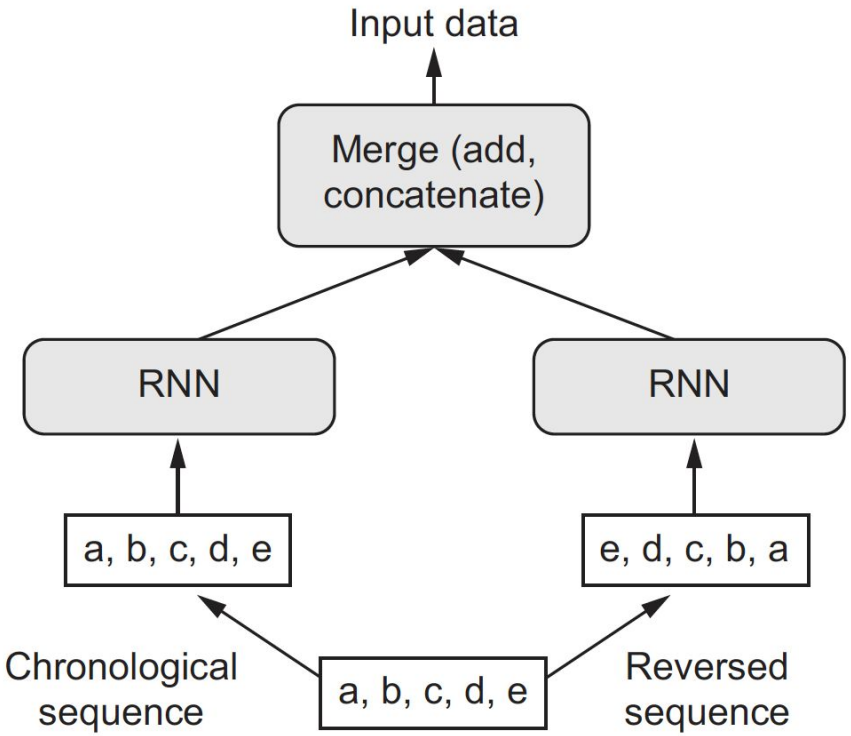


- Since we are still not overfitting too badly, we could safely increase the size of our layers, in quest for a bit of validation loss improvement. This does have a non-negligible computational cost, though.
- Since adding a layer did not help us by a significant factor, we may be seeing diminishing returns to increasing network capacity at this point.

Bidirectional Recurrent Layers

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 a BRNN

```
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)
```

It performs about as well as the regular GRU layer. It's easy to understand why: all of the predictive capacity must be coming from the chronological half of the network, since the anti-chronological half is known to be severely underperforming on this task (again, because the recent past matters much more than the distant past in this case).

Summary

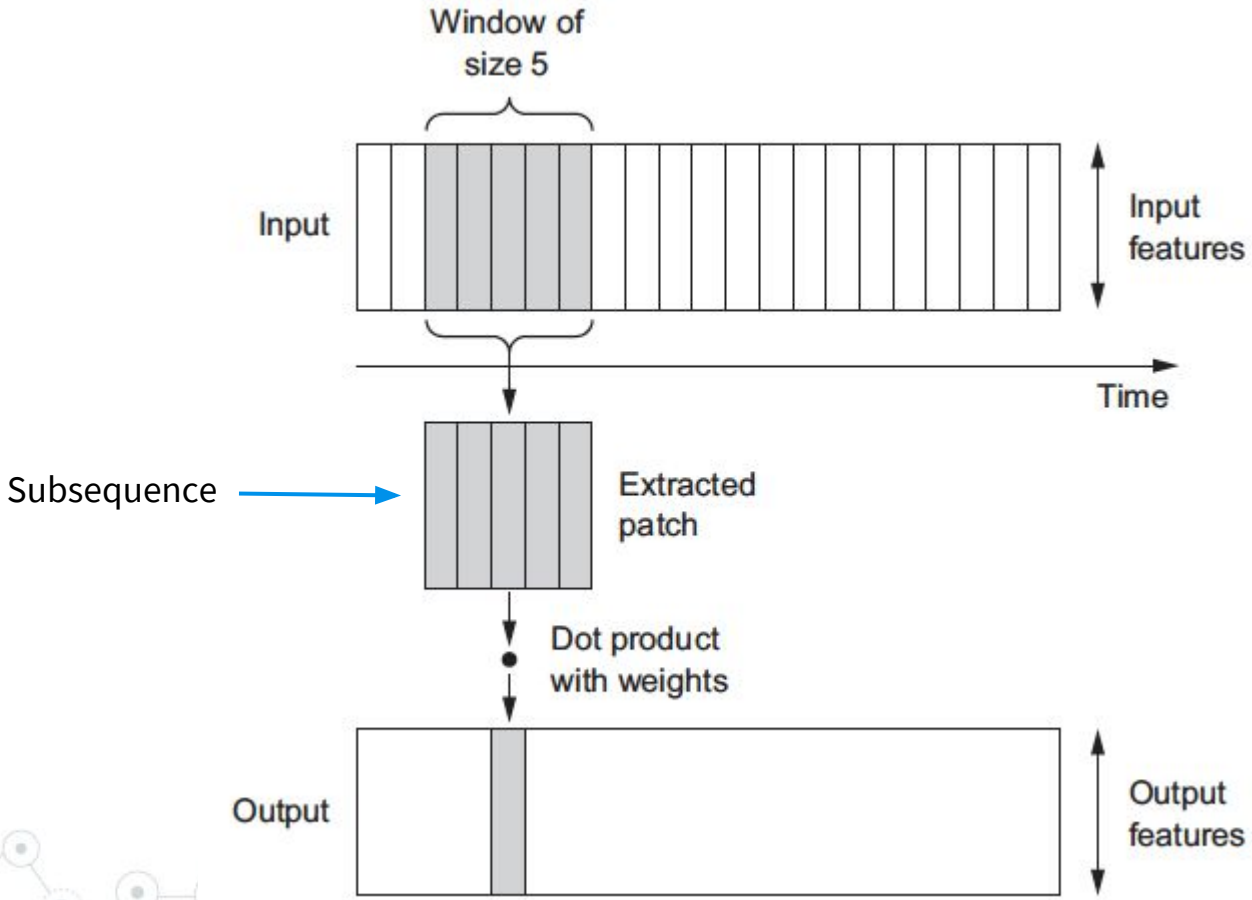
- There are several other things you can try to improve performance
  - Change the number of units in each recurrent layer
  - Try using LSTM layers instead of GRU layers
  - Change the learning rate used by the RMSprop optimizer
  - Try a bigger densely connected classifier on top of the recurrent layers

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 Convolution for Sequence Data



1D Convolution for Sequence Data

- In Keras, use the **Conv1D** layer
  - Takes as input (samples, time, features)
  - Returns 3D tensors

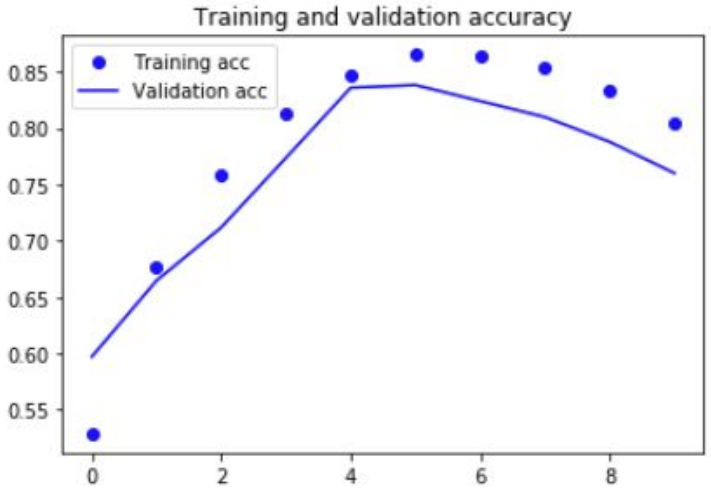
- Combine Conv1D layer with a **MaxPooling1D** layer

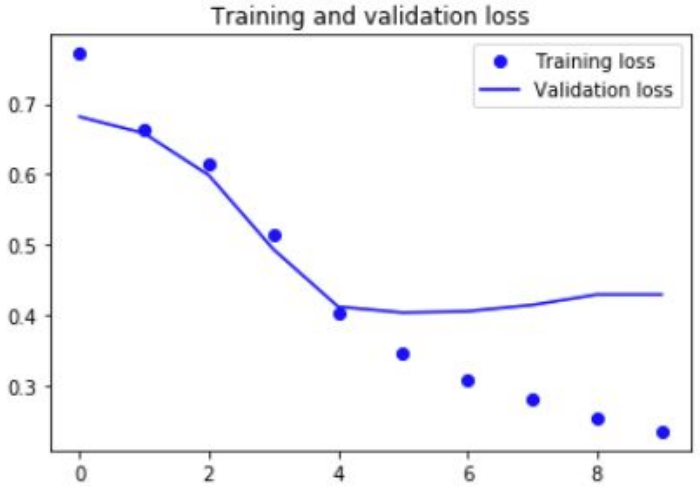
- End with a **GlobalMaxPooling** or **Flatten** layer

- Can use larger windows for 1D CNNs - typically windows of size 7 or 9
  - In a 2D convolution layer, a 3 x 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)
```

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
embedding_1 (Embedding)      (None, 500, 128)          1280000
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 494, 32)           28704
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 98, 32)            0
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 92, 32)            7200
_________________________________________________________________
global_max_pooling1d_1 (Glob (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 33
=================================================================
Total params: 1,315,937
Trainable params: 1,315,937
Non-trainable params: 0
```



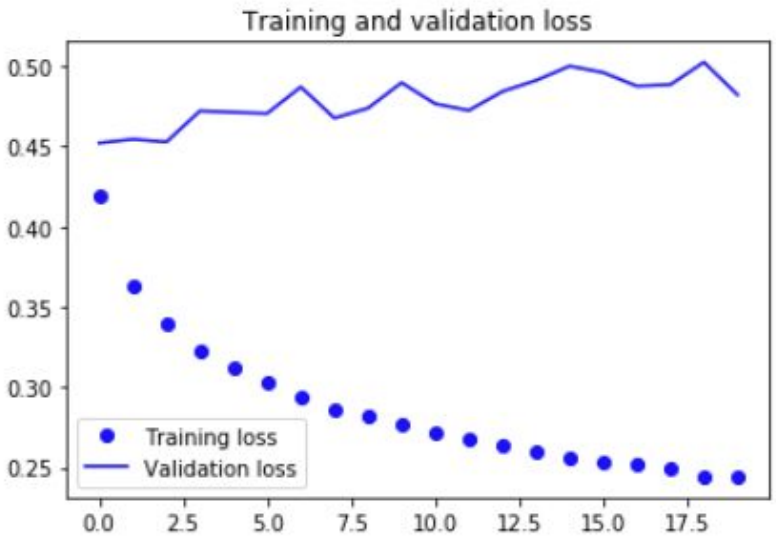


The validation accuracy is a little lower than when using an RNN with an LSTM layer, but the running time for this model is much lower.

Stacking 1D CNN layers

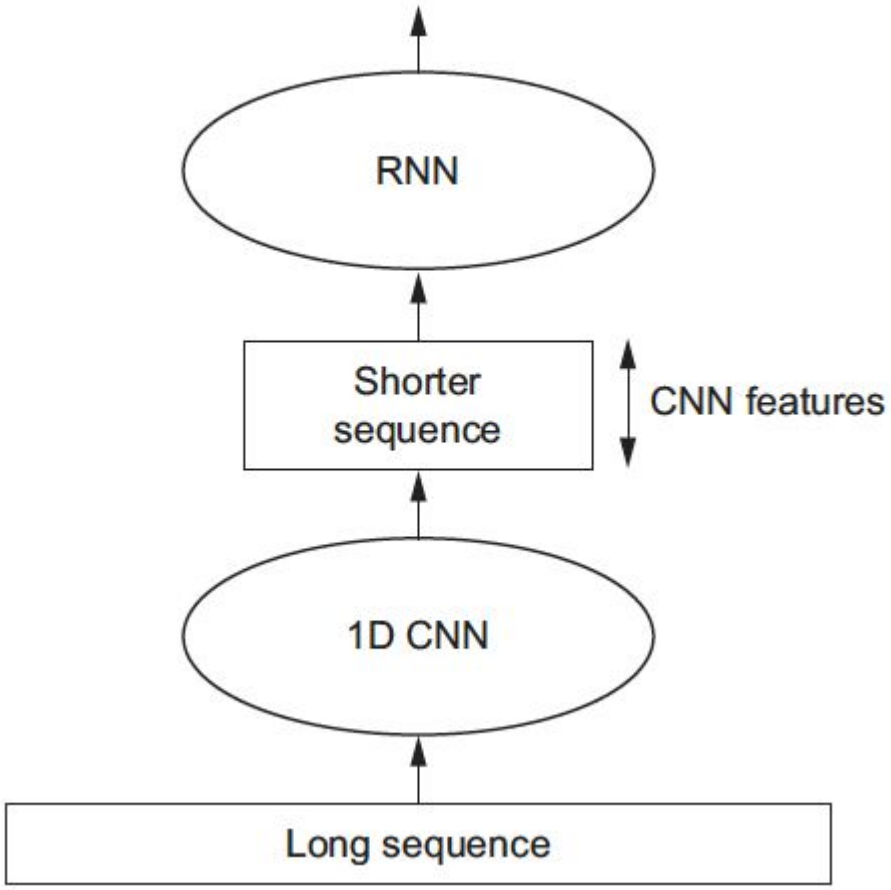
- 1D CNNs process subsequences independently and aren't sensitive to the order of the timesteps - thus, they don't perform well when faced with long sequences
- Could try stacking 1D CNN layers, but still doesn't induce order sensitivity
- Doesn't beat the common-sense baseline
- Due to the fact that 1D CNNs aren't time sensitive
- Let's try combining CNNs and RNNs

```
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)
```



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

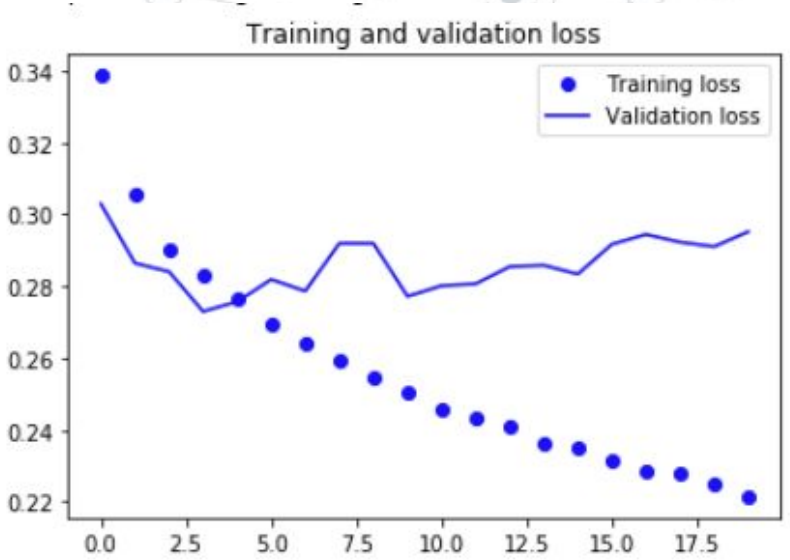


Temperature Forecasting

- With 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)
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.

Summary

- In the same way that 2D convnets perform well for processing visual patterns in 2D space, 1D convnets perform well for processing temporal patterns. They offer a faster alternative to RNNs on some problems, in particular natural language processing tasks.

- Typically, 1D convnets are structured much like their 2D equivalents from the world of computer vision: they consist of stacks of Conv1D layers and Max-Pooling1D layers, ending in a global pooling operation or flattening operation.

- Because RNNs are extremely expensive for processing very long sequences, but 1D convnets are cheap, it can be a good idea to use a 1D convnet as a preprocessing step before an RNN, shortening the sequence and extracting useful representations for the RNN to process.