

A decorative network diagram in the top-left corner of the slide. It features a complex web of interconnected nodes and edges. The nodes are represented by small circles, some of which are solid blue, some are solid grey, and some are hollow blue. The edges are thin grey lines connecting the nodes. The overall shape of the network is irregular and spreads across the top-left portion of the slide.

BST 261: Data Science II

Lecture 10

Recurrent Neural Networks (RNNs) Continued

Heather Mattie
Harvard T.H. Chan School of Public Health
Spring 2 2021

A decorative network diagram in the bottom-right corner of the slide. It is similar in style to the one in the top-left, showing a web of interconnected nodes and edges. The nodes are small circles, some solid blue, some solid grey, and some hollow blue. The edges are thin grey lines. The network is spread across the bottom-right portion of the slide.

Recipe of the Day!

Pink Champagne Cake





Problems with RNNs

Problems with RNNs

- ◎ Recall the formula for a generic RNN:

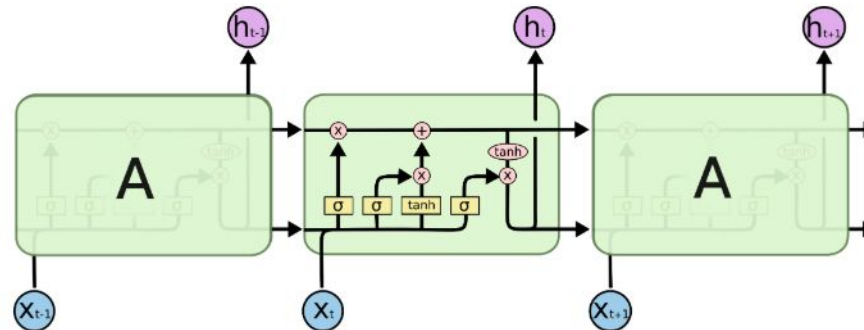
$$h_t = f(X_t W + h_{t-1} U + b)$$

- ◎ What happens for really long sequences during backprop?
 - You multiply by the matrix U repeatedly
 - Largest eigenvalue > 1 , gradient $\longrightarrow \infty$ (explodes)
 - Largest eigenvalue < 1 , gradient $\longrightarrow 0$ (vanishes)

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

Fixing RNNs

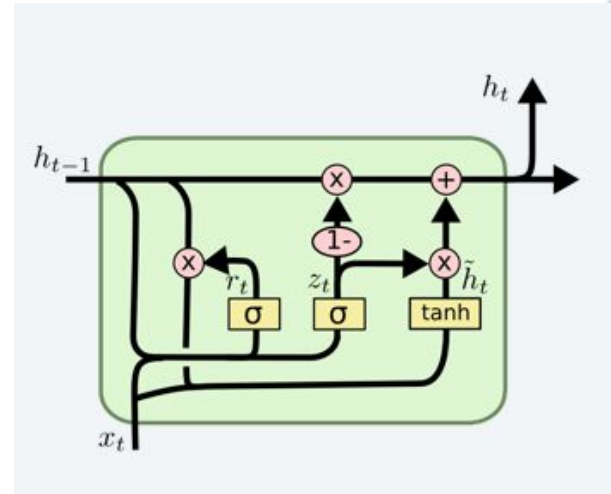
- Sepp Hochreiter and Jürgen Schmidhuber proposed the [long short term memory \(LSTM\) hidden unit in 1997](#)
- LSTMs selectively modify the inputs to produce “well-behaved” outputs, fixing the gradient issues
- Can model very long sequences without having the gradients vanish or explode

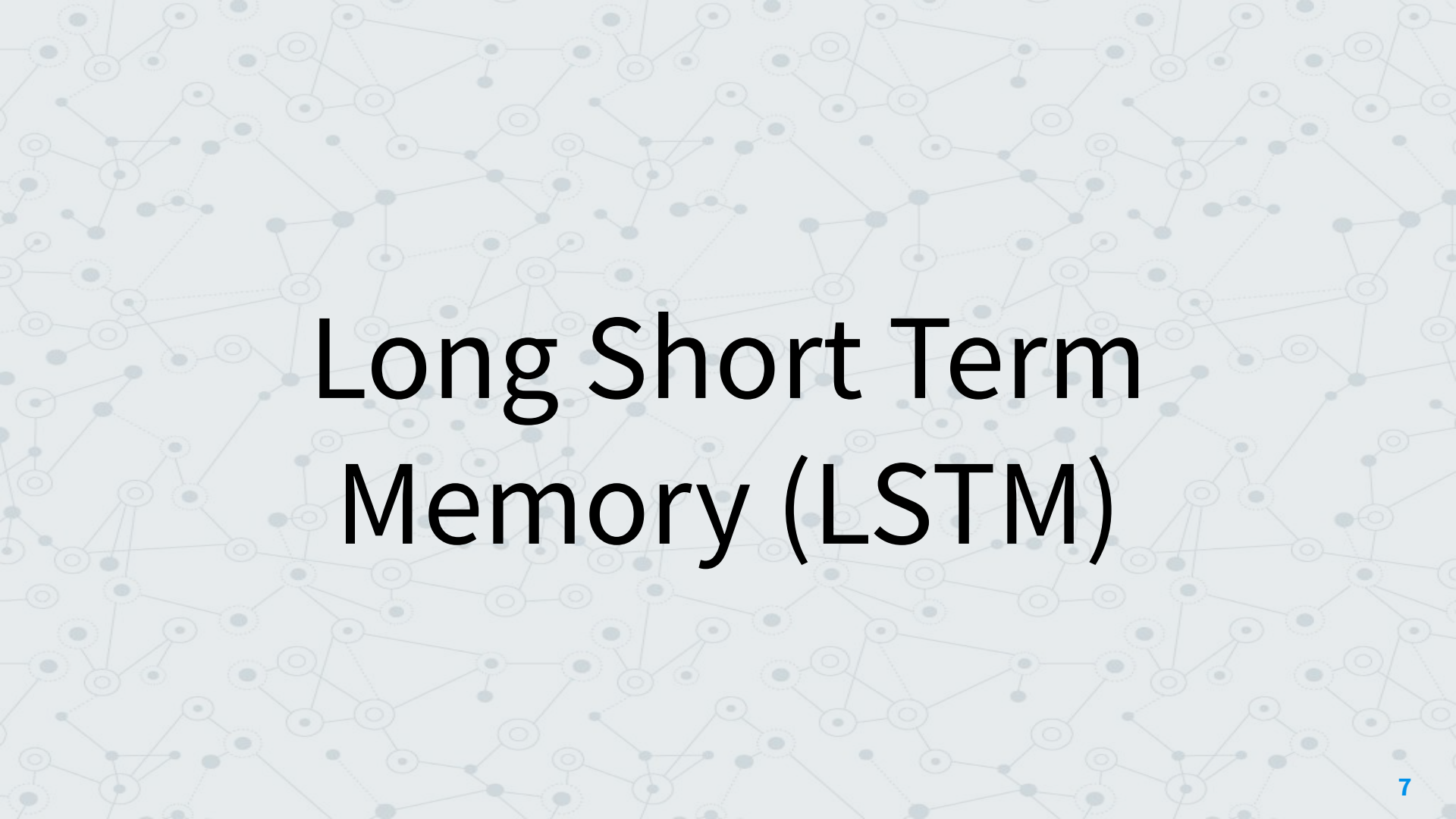


The repeating module in an LSTM contains four interacting layers.

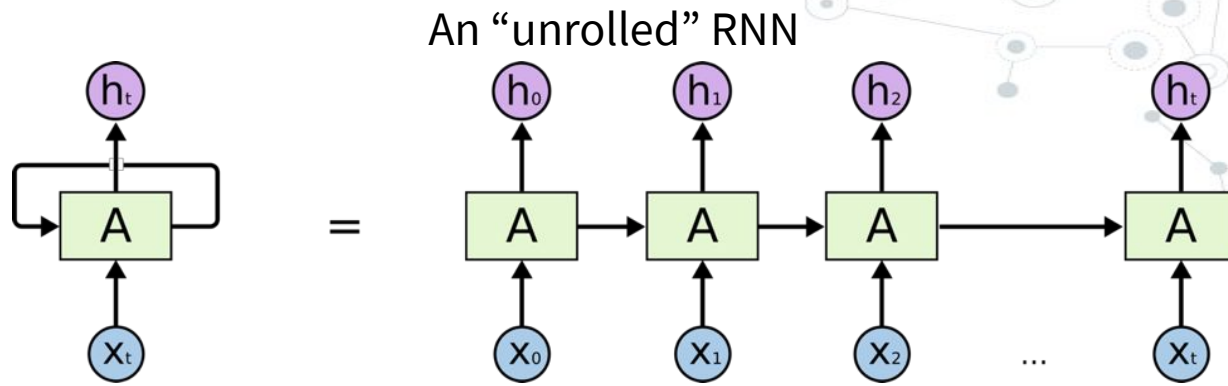
Fixing RNNs

- ◎ [Gated Recurrent Network](#) (GRU)
- ◎ Relatively new (2014), introduced by Cho et al.
- ◎ Combined aspects of the LSTM hidden unit
- ◎ Performance is on par with LSTM but computationally more efficient
- ◎ We'll dig into the details of these two new units



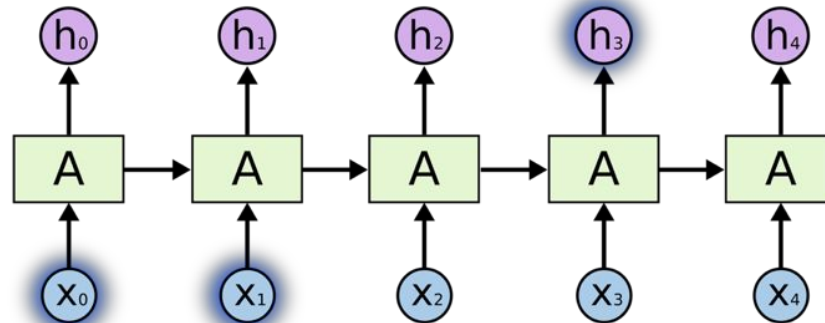


Long Short Term Memory (LSTM)



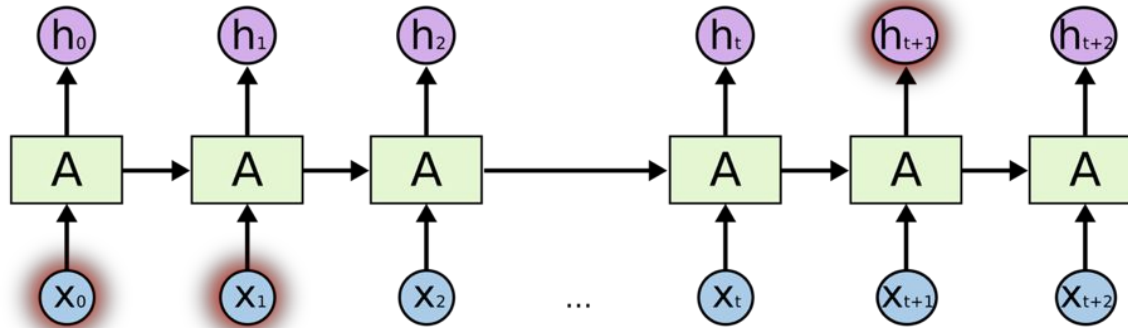
RNN where the output h_3 only depends on the input from x_0 and x_1
(The relevant information needed at h_3 comes from x_0 and x_1)

The gap between relevant information and the place it is needed is small

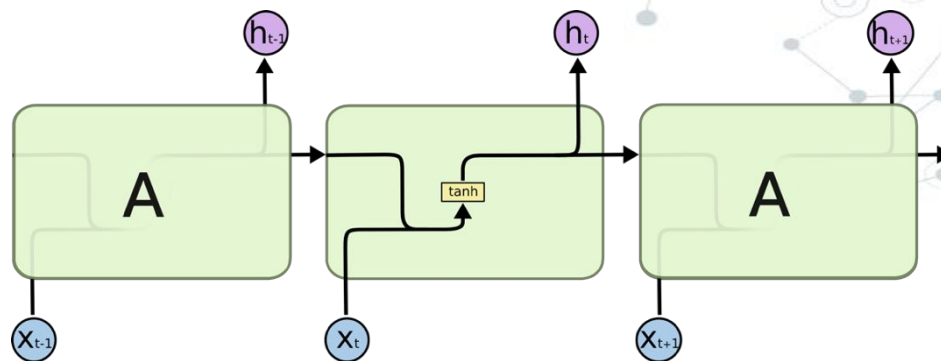


RNN where the output h_{t+1} is dependent on data inputs X_0 and X_1 that are too far for the gradient to carry

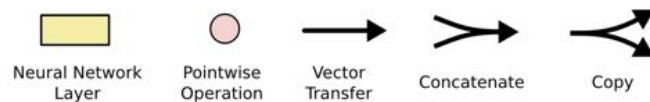
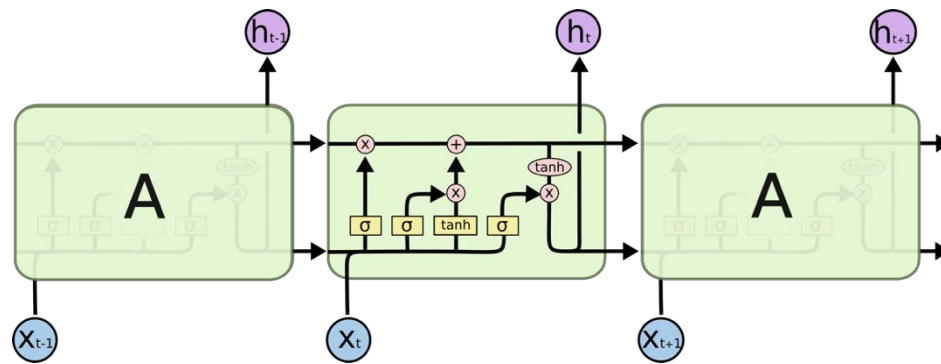
This is an example of a **long-term dependency** - RNNs struggle to learn to make connections when there are large gaps between the relevant information and where it is needed

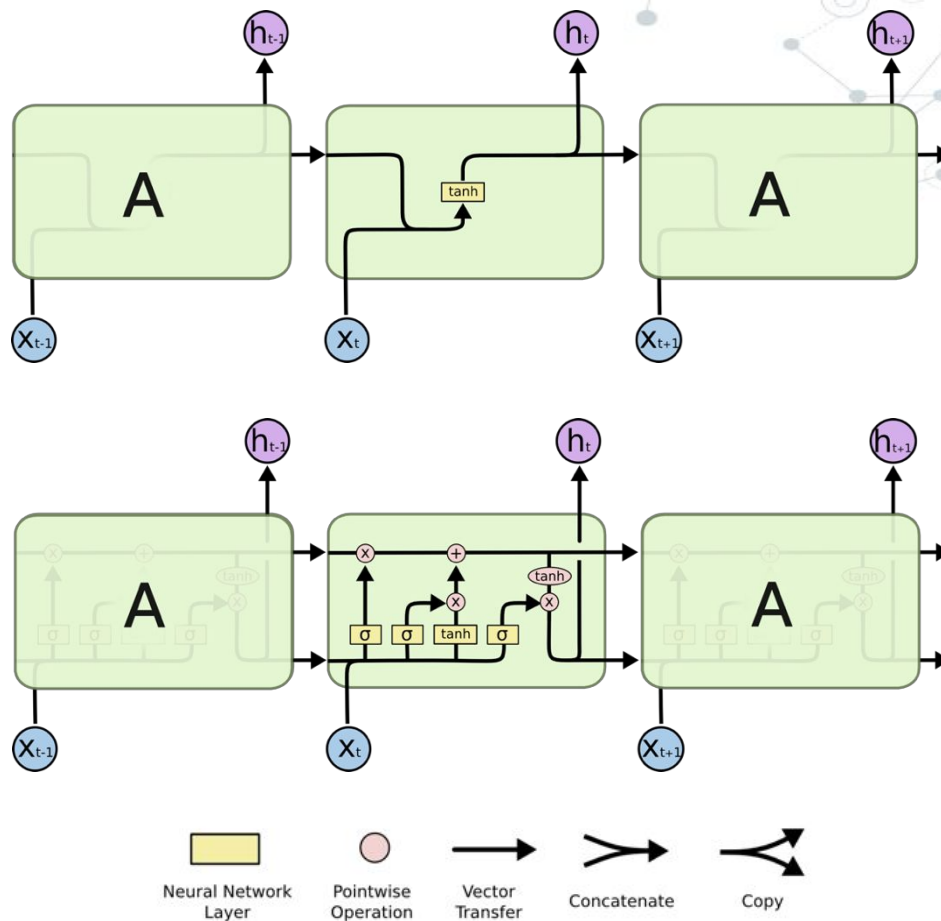
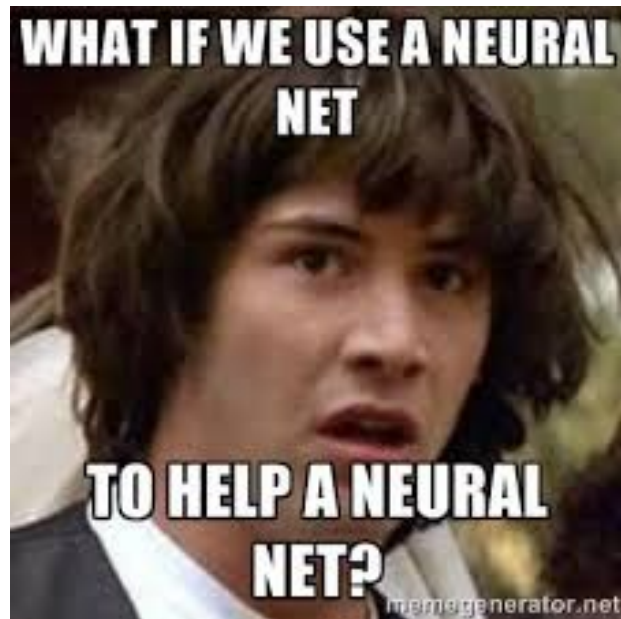


Simple, “vanilla” RNN:

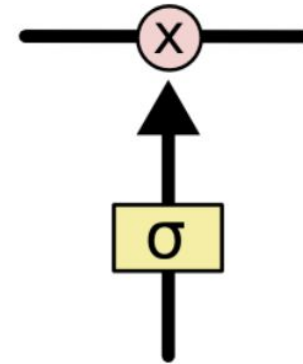
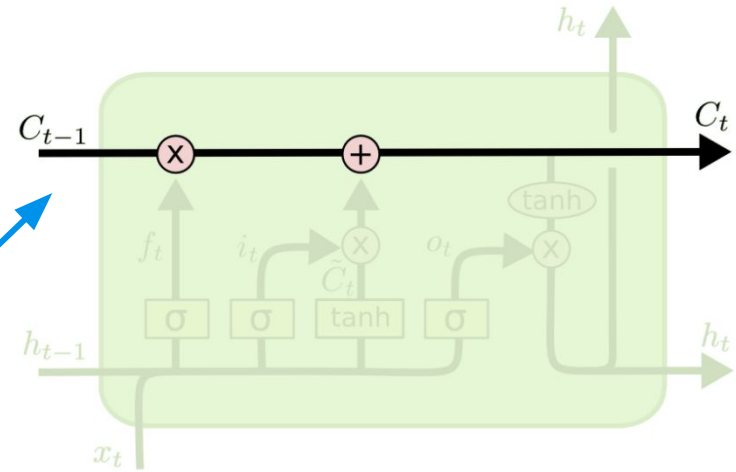


RNN with LSTM units:



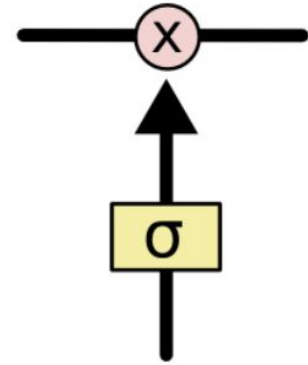


- ⊙ LSTMs were explicitly designed to avoid the long-term dependency problem
- ⊙ The key to LSTMs is the ability to let certain information through and carry it until it is deemed no longer useful (which may not happen)
- ⊙ Information is carried through the sequence in the **cell state**, which acts as a conveyor belt or highway of information (memory of the network)
- ⊙ Information is kept or forgotten by passing through **gates** (neural nets that regulate the flow of information from one time step to the next)

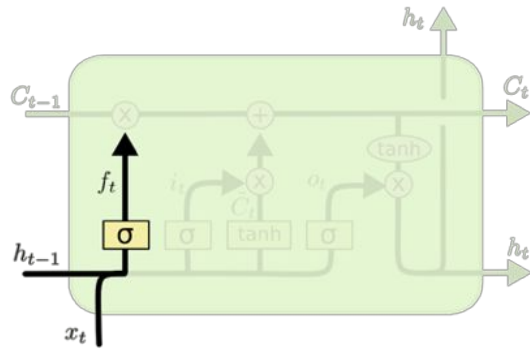


Gates

- ⦿ Gates control which information is let through
- ⦿ They are composed of a sigmoid neural net layer and a pointwise multiplication operation
- ⦿ The sigmoid layer outputs numbers between 0 and 1, representing how much information should be let through
- ⦿ 1 = all information, 0 = no information

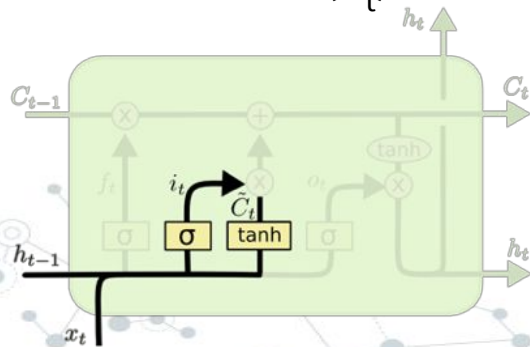


Step 1: Forget Gate - Determine how much of the previous state should affect the current state based on the current observed input x_t



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

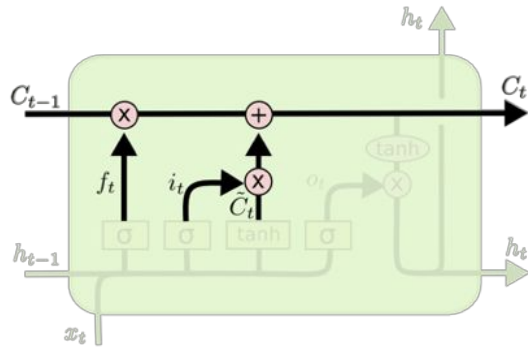
Step 2: Update Cell State - First determine which values we will update and by how much (gate i_t), then create a list of candidate values that we will add to the current state (C_t) based on the current input (x_t) and the previous output (h_{t-1}).



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

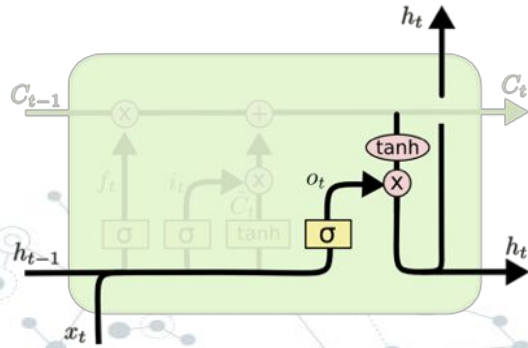
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

Step 3: Execute the Update - update the cell state C_{t-1} to C_t .



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

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



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

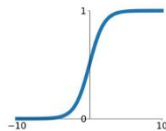
$$h_t = o_t * \tanh(C_t)$$

Why tanh?

- ◎ To overcome the vanishing/exploding gradient problem
- ◎ Forces values to be between -1 and 1

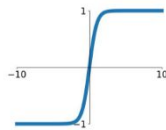
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



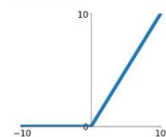
tanh

$$\tanh(x)$$



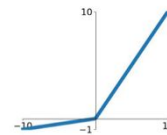
ReLU

$$\max(0, x)$$



Leaky ReLU

$$\max(0.1x, x)$$

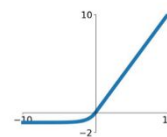


Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



LSTM Variants

- ◎ The steps we went through are for the standard, “normal” LSTM
- ◎ There are several variations - see blog post link from previous slide
- ◎ Encoder-decoder LSTMs led to the emergence of the

Attention Mechanism

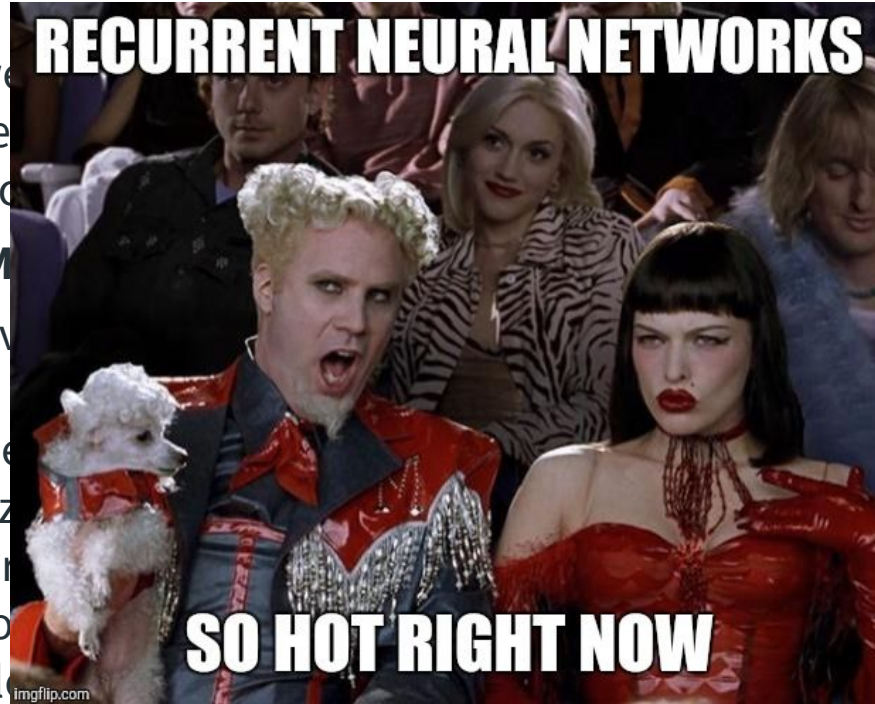
- Selectively concentrates on a few relevant things while ignoring others
- ◎ Think of an encoder as part of a neural net that reads in a sequence, tries to summarize it (encode a context vector), and passes it to the decoder
- ◎ The decoder translates the input from the encoder
- ◎ The Attention Mechanism overcame shortcomings of encoder-decoder LSTMs and led to huge breakthroughs in NLP

LSTM Variants

- ◎ The steps we
- ◎ There are se
- ◎ Encoder-dec

Attention M

- Selectiv
others
- ◎ Think of an e
to summariz
- ◎ The decoder
- ◎ The Attention
LSTMs and l



” LSTM
vious slide

hile ignoring

a sequence, tries
to the decoder

coder-decoder

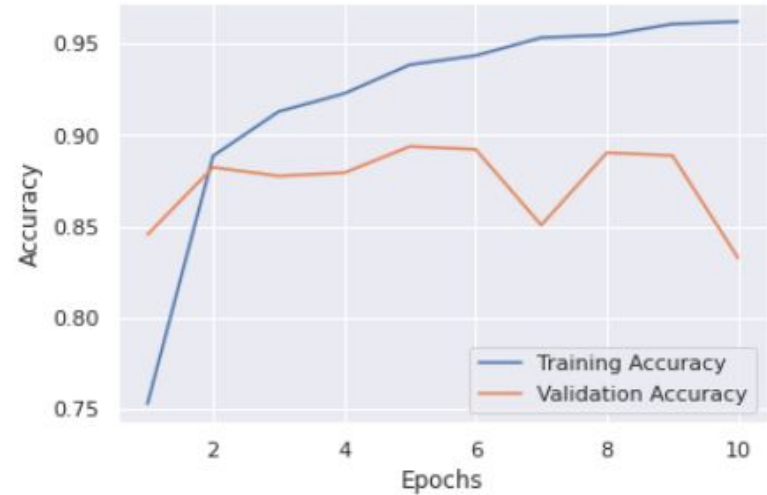
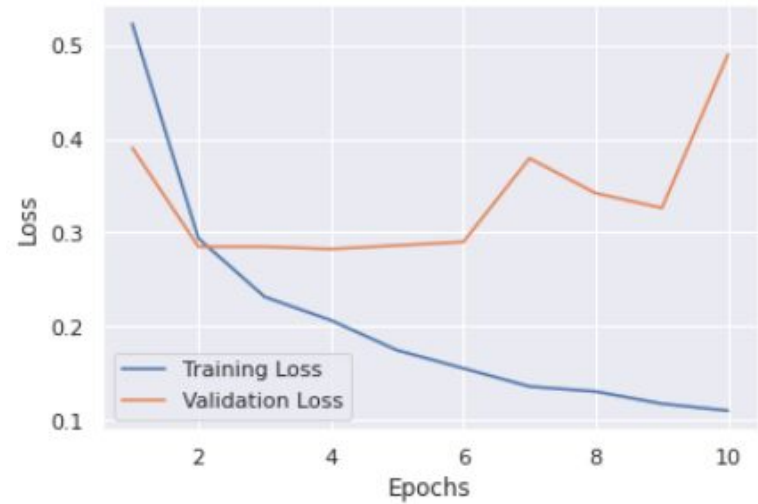
LSTM in Keras

- 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 lecture
- We only specify the output dimensionality of the LSTM layer, and leave every other argument (there's a lot) to the Keras defaults

```
1 model = tf.keras.models.Sequential([  
2     tf.keras.layers.Embedding(max_features, 32),  
3  
4     tf.keras.layers.LSTM(32),  
5  
6     tf.keras.layers.Dense(1, activation='sigmoid')  
7 ])  
8  
9 model.compile(optimizer = tf.keras.optimizers.RMSprop(),  
10               loss='binary_crossentropy',  
11               metrics=['accuracy'])  
12  
13 history = model.fit(input_train, y_train,  
14                     epochs=10,  
15                     batch_size=128,  
16                     validation_split=0.2)
```

LSTM in Keras

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



The background of the slide features a complex, light gray network pattern. It consists of numerous small circles, some of which are double-lined, connected by thin, intersecting lines that form a web-like structure across the entire page.

Gated Recurrent Unit (GRU)

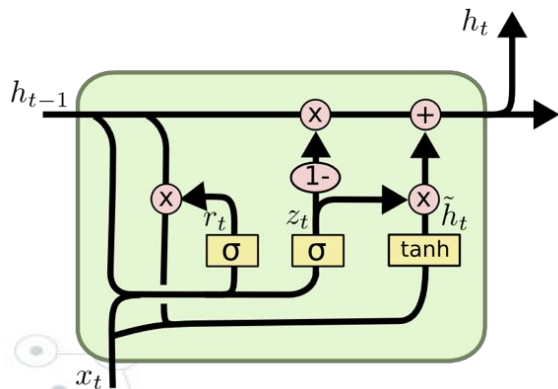
Gated Recurrent Unit (



)

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 (due to fewer tensor operations)



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

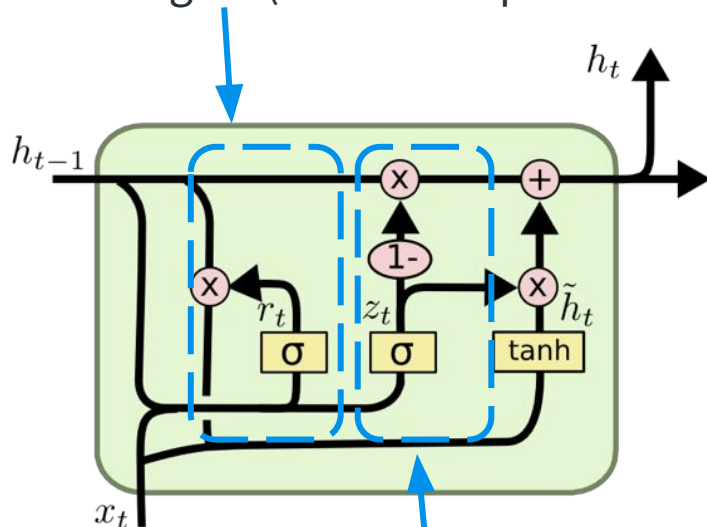
$$r_t = \sigma (W_r \cdot [h_{t-1}, x_t])$$

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

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

GRU

Reset gate (how much past information to forget)



Update gate (decides which new information to throw away and which to add)

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

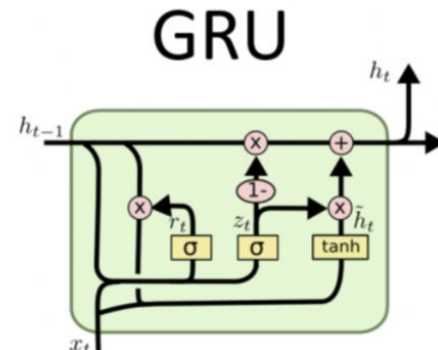
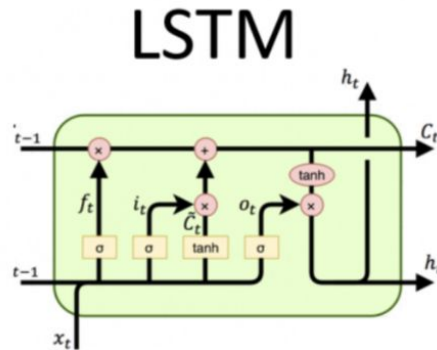
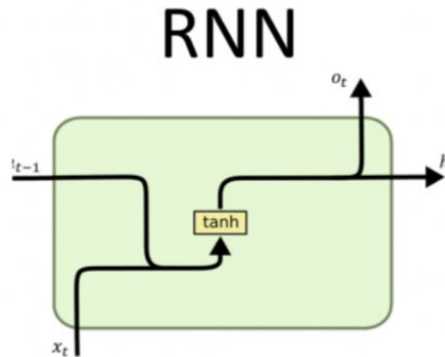
$$r_t = \sigma(W_r \cdot [h_{t-1}, x_t])$$

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

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

LSTM vs GRU

- ⊙ LSTM performs better on long sequences
- ⊙ GRUs are faster to train
- ⊙ GRUs are simpler to understand and modify
- ⊙ A nice [video explanation/post](#) of LSTM and GRU



The background of the slide features a complex, light blue network pattern. It consists of numerous small circles, some solid and some hollow, connected by thin, intersecting lines that form a web-like structure across the entire page.

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** dataset recorded at the [Weather Station at Max Planck Institute for Biochemistry](#) in Jena, Germany



Example: temperature forecasting

- ◎ 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
- ◎ [Colab notebook](#)
- ◎ [Data file](#)


```
1 fname = 'path/jena_climate_2009_2016.csv'
2 f = open(fname)
3 data = f.read()
4 f.close()
5
6 lines = data.split('\n')      # Each line is 1 recording
7 header = lines[0].split(',')  # Variable names are separated by commas
8 lines = lines[1:]            # Drop first line (it's a header)
9
10 print(header)
11 print(len(lines))
```

```
['Date Time', 'p (mbar)', 'T (degC)', 'Tpot (K)', 'Tdew (degC)', 'rh (%)',
420551
```

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

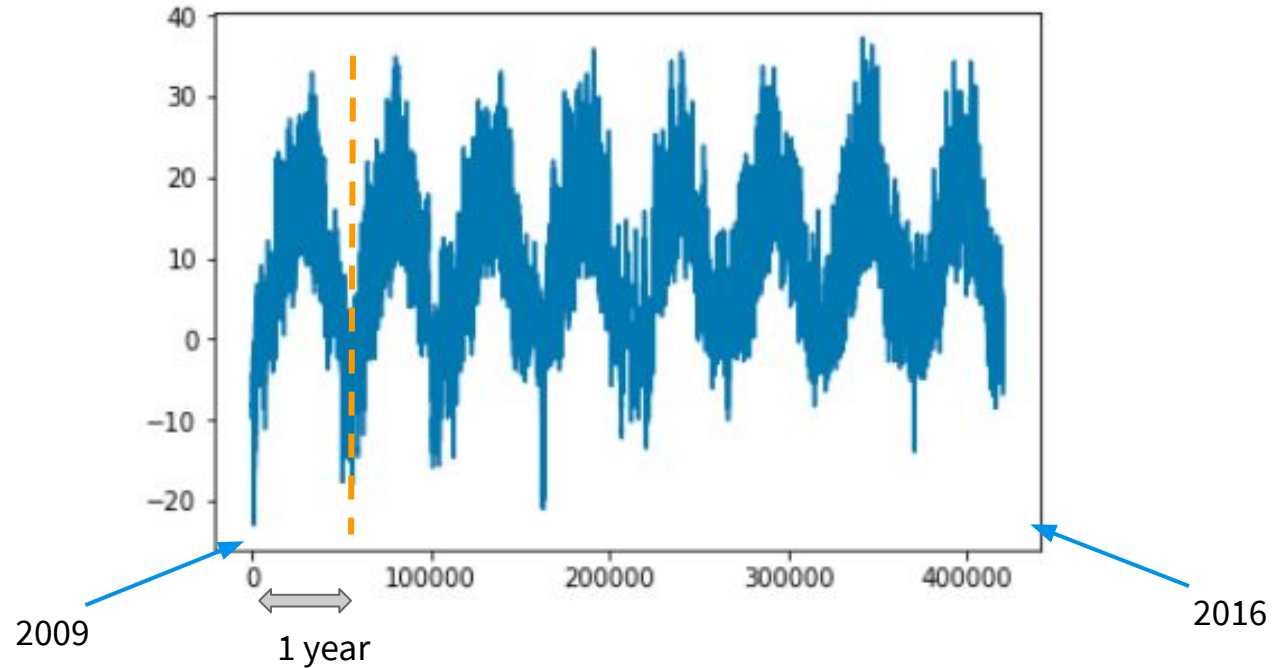
Number of rows
(observations)

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

Drop first column (the
unnecessary date/time)

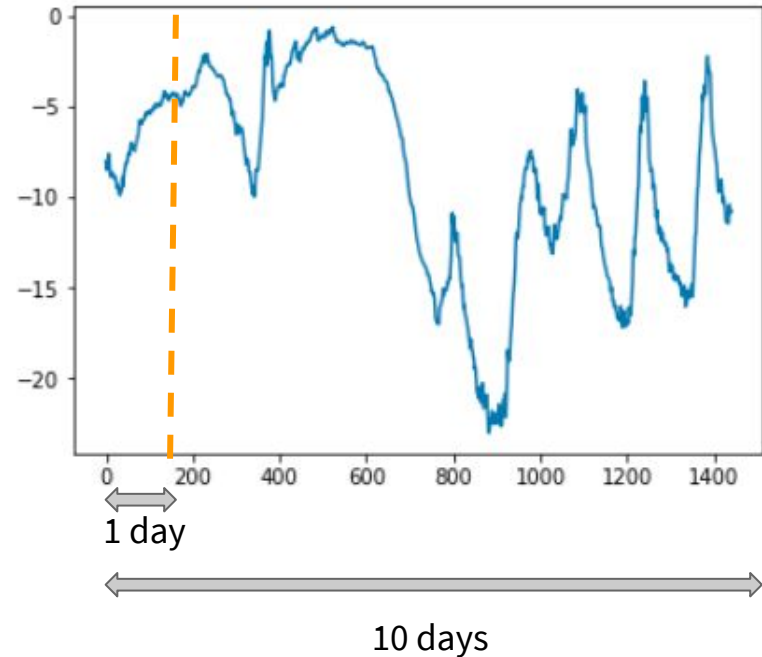
```
(420551, 14)
```

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 standard deviation 1

Temperature Forecasting in Keras

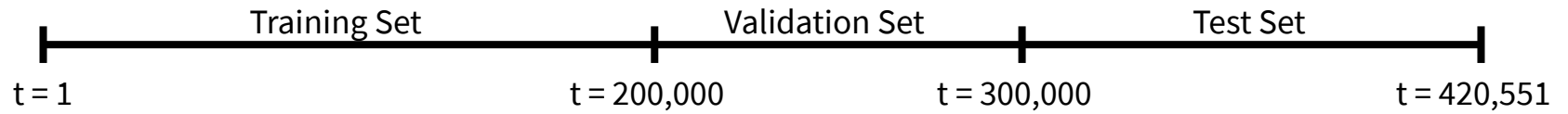
Generate samples

- **data:** The original array of floating point data
- **lookback:** How many timesteps back should our input data go
- **delay:** How many timesteps in the future should our target be

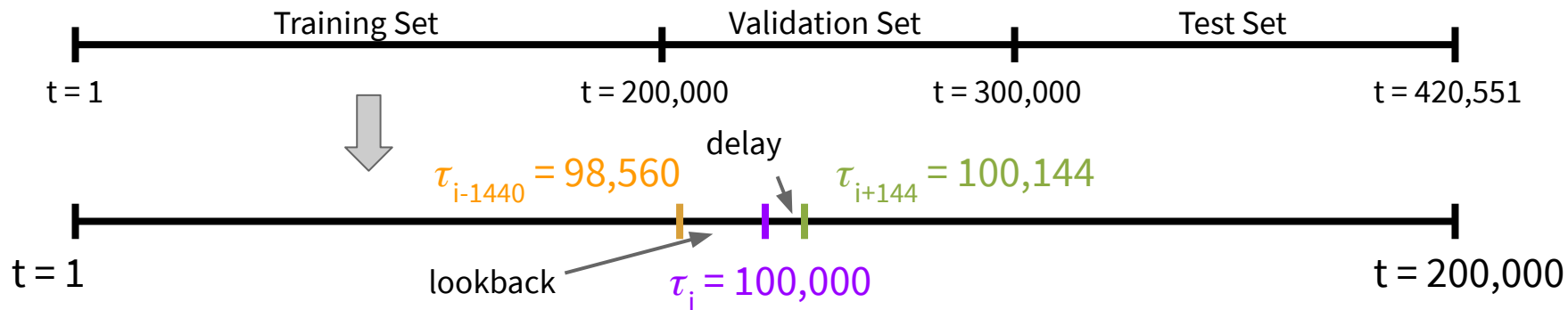
Temperature Forecasting in Keras

- **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 to 6 in order to draw one data point every hour

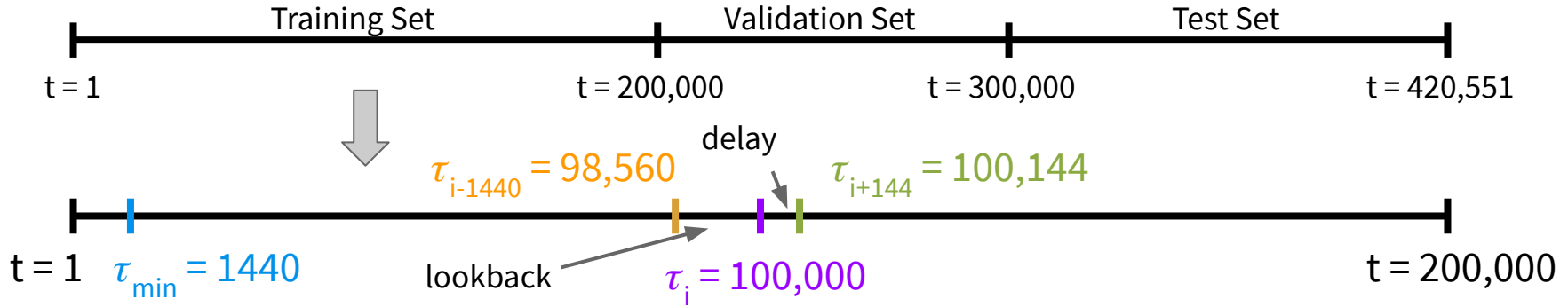
Timeline



Timeline

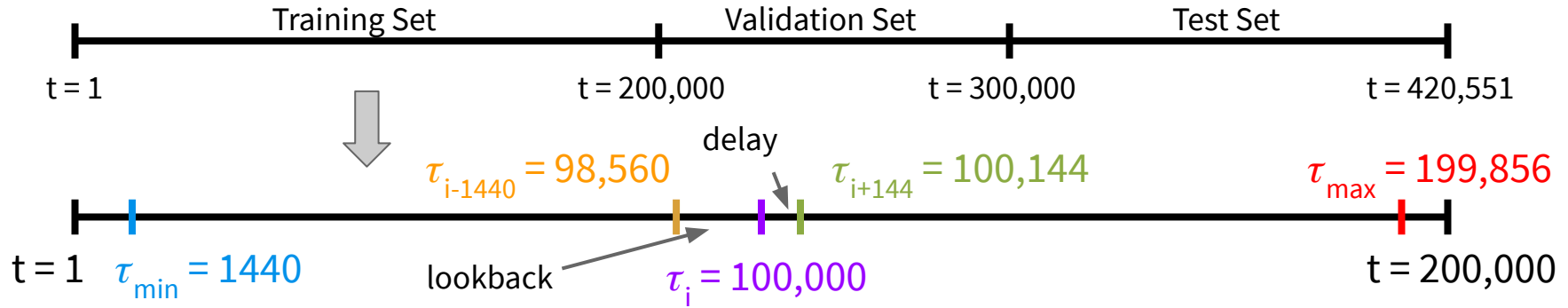


Timeline



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 $\tau_i < 1441$, we won't have enough prior data to make a prediction.

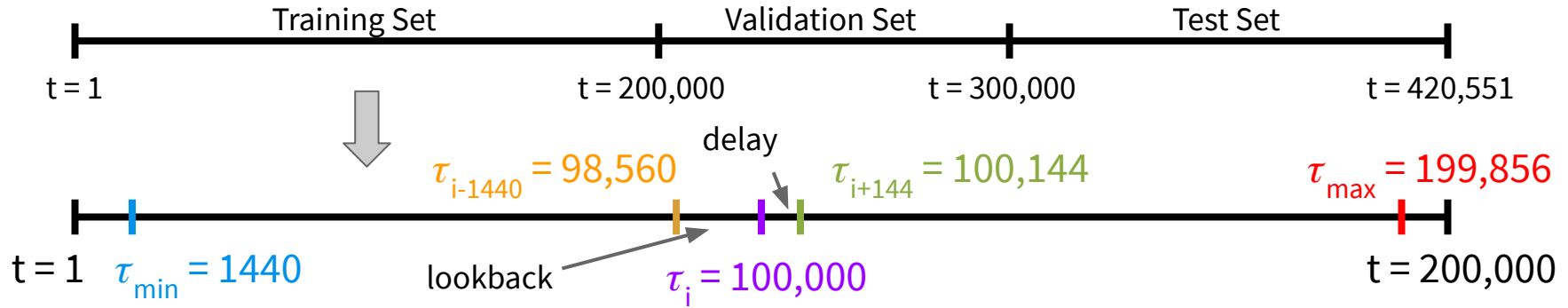
Timeline



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 $\tau_i < 1441$, 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 $\tau_i > 199,856$, 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 $\tau_i < 1441$, 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 $\tau_i > 199,856$, 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

```
1 def generator(data, lookback, delay, min_index, max_index,  
2               shuffle=False, batch_size=128, step=6):  
3     if max_index is None:  
4         max_index = len(data) - delay - 1  
5     i = min_index + lookback  
6     while 1:  
7         if shuffle:  
8             rows = np.random.randint(  
9                 min_index + lookback, max_index, size=batch_size)  
10        else:  
11            if i + batch_size >= max_index:  
12                i = min_index + lookback  
13            rows = np.arange(i, min(i + batch_size, max_index))  
14            i += len(rows)  
15  
16            samples = np.zeros((len(rows),  
17                               lookback // step,  
18                               data.shape[-1]))  
19            targets = np.zeros((len(rows),))  
20            for j, row in enumerate(rows):  
21                indices = range(rows[j] - lookback, rows[j], step)  
22                samples[j] = data[indices]  
23                targets[j] = data[rows[j] + delay][1]  
24            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.

```
1 lookback = 1440
2 step = 6
3 delay = 144
4 batch_size = 128
5
6 train_gen = generator(float_data,
7                       lookback=lookback,
8                       delay=delay,
9                       min_index=0,
10                      max_index=200000,
11                      shuffle=True,
12                      step=step,
13                      batch_size=batch_size)
14 val_gen = generator(float_data,
15                    lookback=lookback,
16                    delay=delay,
17                    min_index=200001,
18                    max_index=300000,
19                    step=step,
20                    batch_size=batch_size)
21 test_gen = generator(float_data,
22                     lookback=lookback,
23                     delay=delay,
24                     min_index=300001,
25                     max_index=None,
26                     step=step,
27                     batch_size=batch_size)
28
29 # This is how many steps to draw from `val_gen`
30 # in order to see the whole validation set:
31 val_steps = (300000 - 200001 - lookback) // batch_size
32
33 # This is how many steps to draw from `test_gen`
34 # in order to see the whole test set:
35 test_steps = (len(float_data) - 300001 - lookback) // batch_size
36
37 print(val_steps)
38 print(test_steps)
```

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

```
1 def evaluate_naive_method():
2     batch_maes = []
3     for step in range(val_steps):
4         samples, targets = next(val_gen)
5         preds = samples[:, -1, 1]
6         mae = np.mean(np.abs(preds - targets))
7         batch_maes.append(mae)
8     print(np.mean(batch_maes))
9
10 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°C.

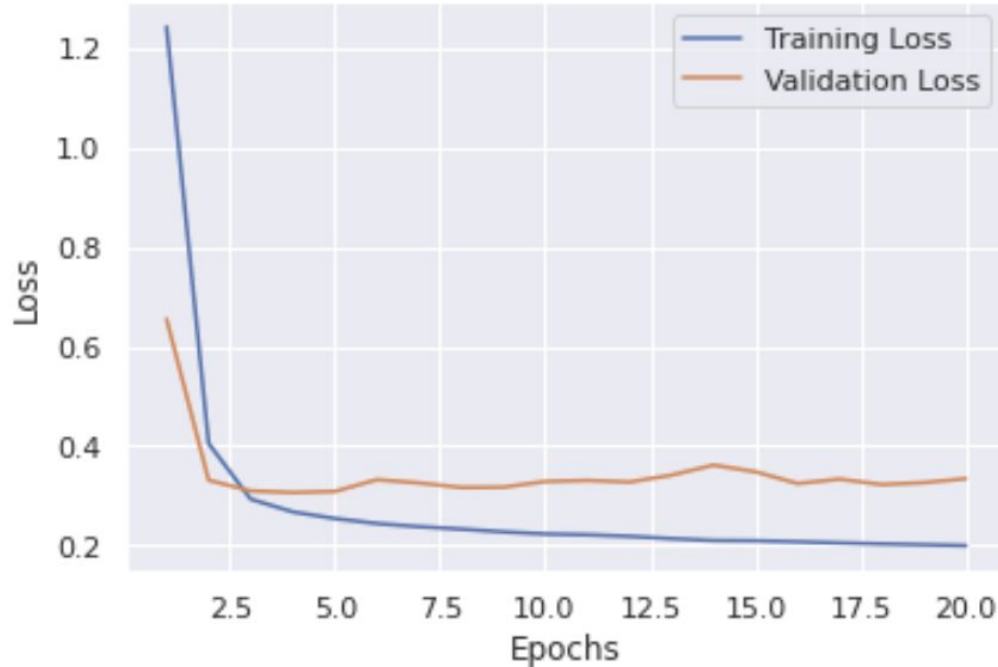
That's a fairly large average absolute error – now the task is to leverage our knowledge of deep learning to do better.

Temperature Forecasting - Simple Model

- Let's first try a simple model (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

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


Temperature Forecasting - Simple Model



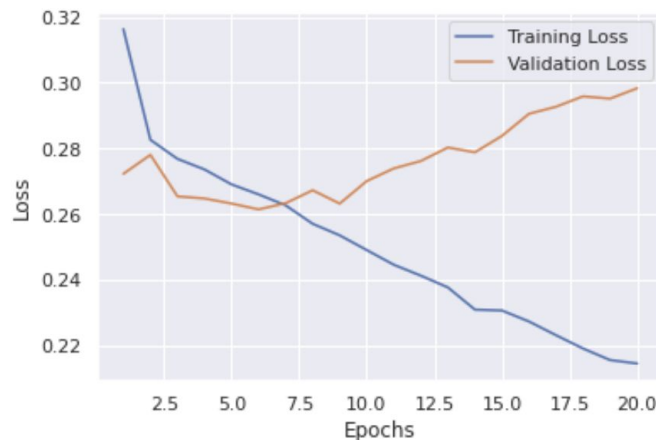
We get MAEs above 0.3 for the validation loss - worse than our benchmark.

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

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.

```
1 model = keras.Sequential([
2     layers.GRU(32, input_shape=(None, float_data.shape[-1])),
3
4     layers.Dense(1)
5 ])
6
7 model.compile(optimizer = tf.keras.optimizers.RMSprop(),
8               loss='mae')
9
10 history = model.fit(train_gen,
11                     steps_per_epoch=500,
12                     epochs=20,
13                     validation_data=val_gen,
14                     validation_steps=val_steps)
```



The background of the slide is a light gray network diagram. It consists of numerous small circular nodes, some of which are solid gray and others are hollow with a gray outline. These nodes are interconnected by a web of thin, light gray lines, creating a complex, interconnected pattern that resembles a neural network or a data graph.

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**

Recurrent Dropout

- ◎ 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

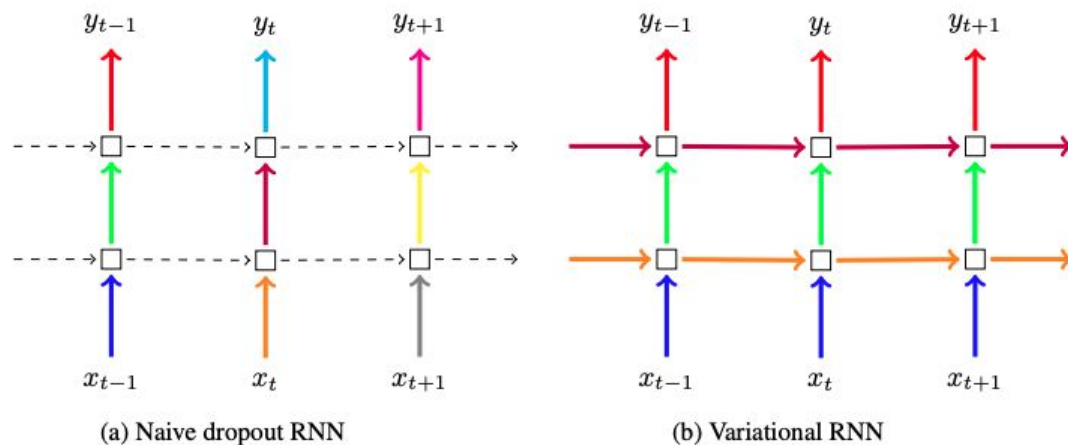
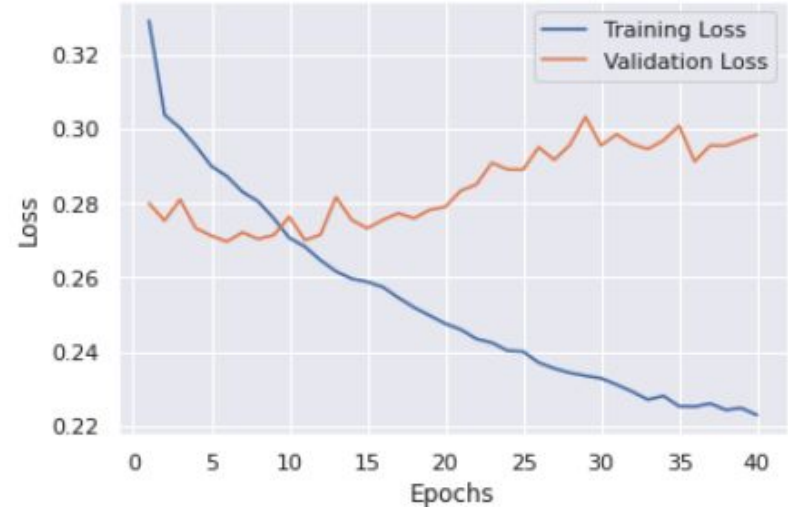


Figure 1: **Depiction of the dropout technique following our Bayesian interpretation (right) compared to the standard technique in the field (left).** Each square represents an RNN unit, with horizontal arrows representing time dependence (recurrent connections). Vertical arrows represent the input and 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

```
1 model = keras.Sequential([
2     layers.GRU(32,
3         dropout = 0.2,
4         recurrent_dropout=0.2,
5         input_shape=(None, float_data.shape[-1])),
6
7     layers.Dense(1)
8 ])
9
10 model.compile(optimizer = tf.keras.optimizers.RMSprop(),
11               loss='mae')
12
13 history = model.fit(train_gen,
14                     steps_per_epoch=500,
15                     epochs=40,
16                     validation_data=val_gen,
17                     validation_steps=val_steps)
```



This helps a little with overfitting - increasing the dropout percentage might help more.

We have more stable evaluation scores, but our best scores are not much lower than they were previously



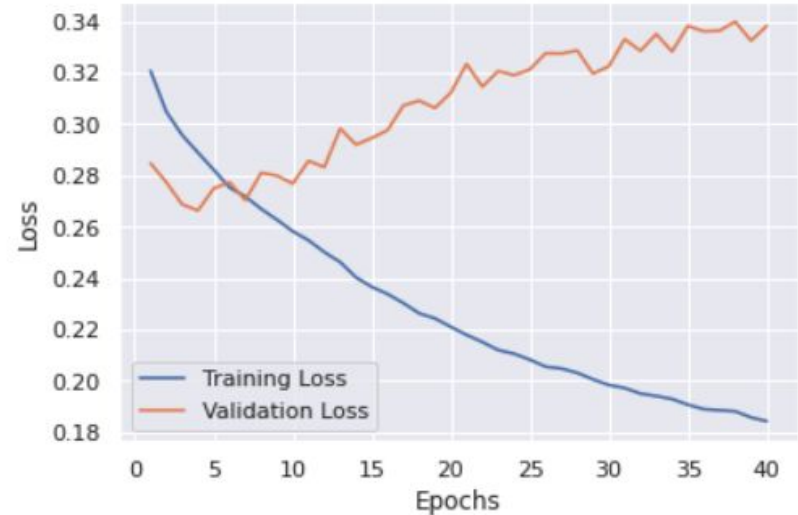
Stacking Recurrent Layers

Stacking Recurrent Layers

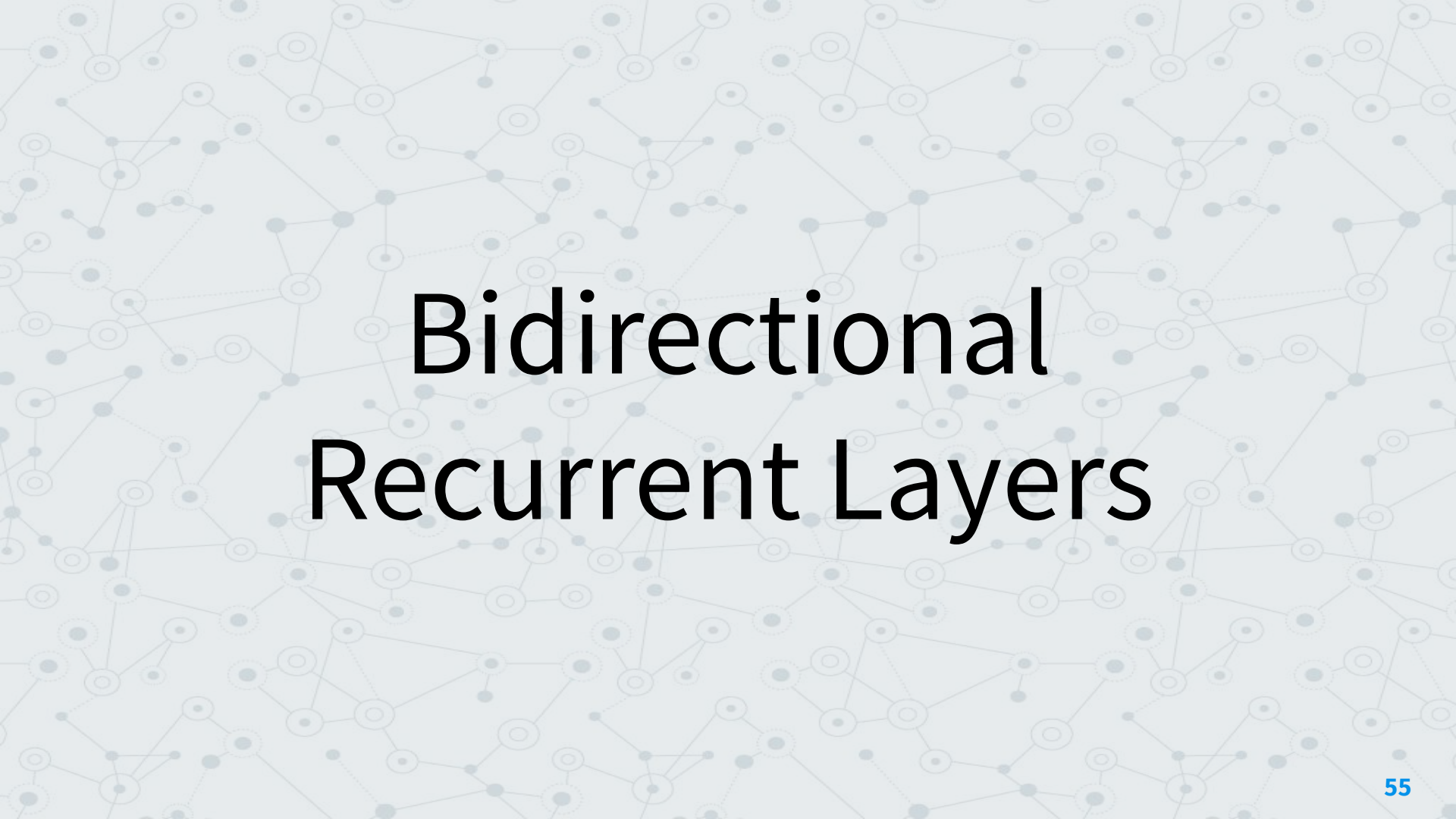
- ◎ Because we have hit a performance bottleneck, we should consider increasing the capacity of the network - make the model more complex
- ◎ Increasing network capacity is typically done by increasing the number of units in the layers or adding more layers.
- ◎ Recurrent layer stacking is a classic way to build more-powerful recurrent networks: for instance, what currently powers the Google Translate algorithm is a stack of 8 large LSTM layers—that's huge!

Stacking Recurrent Layers in Keras

```
1 model = keras.Sequential([
2     layers.GRU(32,
3         dropout = 0.1,
4         recurrent_dropout=0.5,
5         return_sequences=True,
6         input_shape=(None, float_data.shape[-1])),
7     layers.GRU(64, activation='relu',
8         dropout = 0.1,
9         recurrent_dropout=0.5),
10    layers.Dense(1)
11 ])
12
13 model.compile(optimizer = tf.keras.optimizers.RMSprop(),
14     loss='mae')
15
16 history = model.fit(train_gen,
17     steps_per_epoch=500,
18     epochs=40,
19     validation_data=val_gen,
20     validation_steps=val_steps)
```



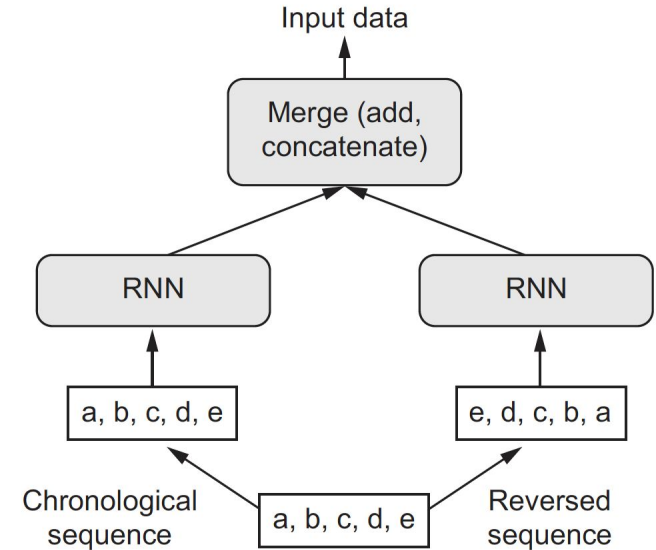
- The overfitting becomes worse, signaling the network capacity is too high, i.e. the model is too complex and has too many parameters.
- It would probably be best to drop the added layer and increase the number of nodes in the first GRU layer

The background of the slide is a light gray network diagram. It consists of numerous small circular nodes, some of which are highlighted with a darker blue or gray fill. These nodes are interconnected by a web of thin, light gray lines, creating a complex, organic-looking structure that resembles a neural network or a data graph. The overall aesthetic is clean and technical.

Bidirectional Recurrent Layers

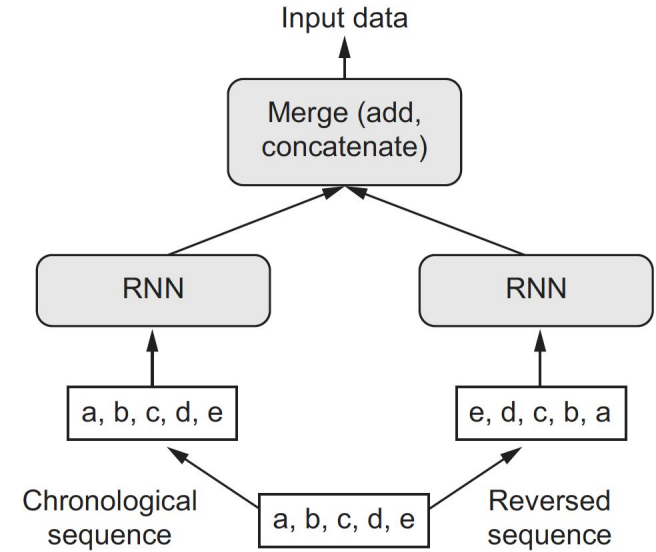
Bidirectional RNNs

- ◎ 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



Bidirectional RNNs

- Uses 2 regular RNNs, each of which processes the input sequence in one direction (chronologically and anti chronologically), and then merges their representations
- Catches patterns that may be overlooked by a regular RNN



Temperature Forecasting with a BRNN

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).

```
1 model = keras.Sequential([
2     layers.GRU(32, input_shape=(None, float_data.shape[-1])),
3
4     layers.Dense(1)
5 ])
6
7 model.compile(optimizer = tf.keras.optimizers.RMSprop(),
8               loss='mae')
9
10 history = model.fit(train_gen_reverse,
11                     steps_per_epoch=500,
12                     epochs=20,
13                     validation_data=val_gen_reverse,
14                     validation_steps=val_steps)
```

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 (or any optimizer)
 - Try a bigger densely connected classifier on top of the recurrent layers