# BST 261: Data Science II
# Lecture 10
## Recurrent Neural Networks (RNNs) Continued
## Transformers

Santiago Romero Brufau
Harvard T.H. Chan School of Public Health
Spring 2 2022

# Transformers

# Attention Is All You Need

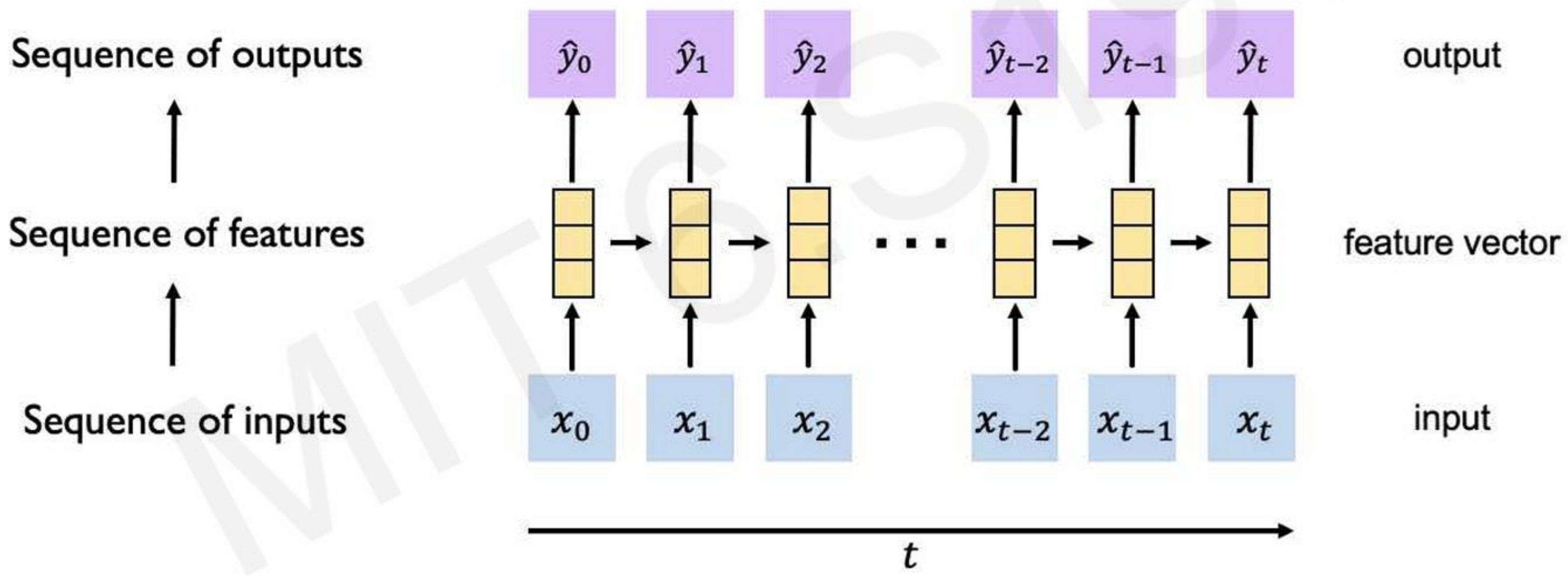
**Ashish Vaswani*** Google Brain avaswani@google.com
**Noam Shazeer*** Google Brain noam@google.com
**Niki Parmar*** Google Research nikip@google.com
**Jakob Uszkoreit*** Google Research usz@google.com
**Llion Jones*** Google Research llion@google.com
**Aidan N. Gomez*** † University of Toronto aidan@cs.toronto.edu
**Łukasz Kaiser*** Google Brain lukaszkaiser@google.com
**Illia Polosukhin*** ‡ illia.polosukhin@gmail.com

## Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.8 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature. We show that the Transformer generalizes well to other tasks by applying it successfully to English constituency parsing both with large and limited training data.

# Goal of Sequence Modeling

RNNs: recurrence to model sequence dependencies

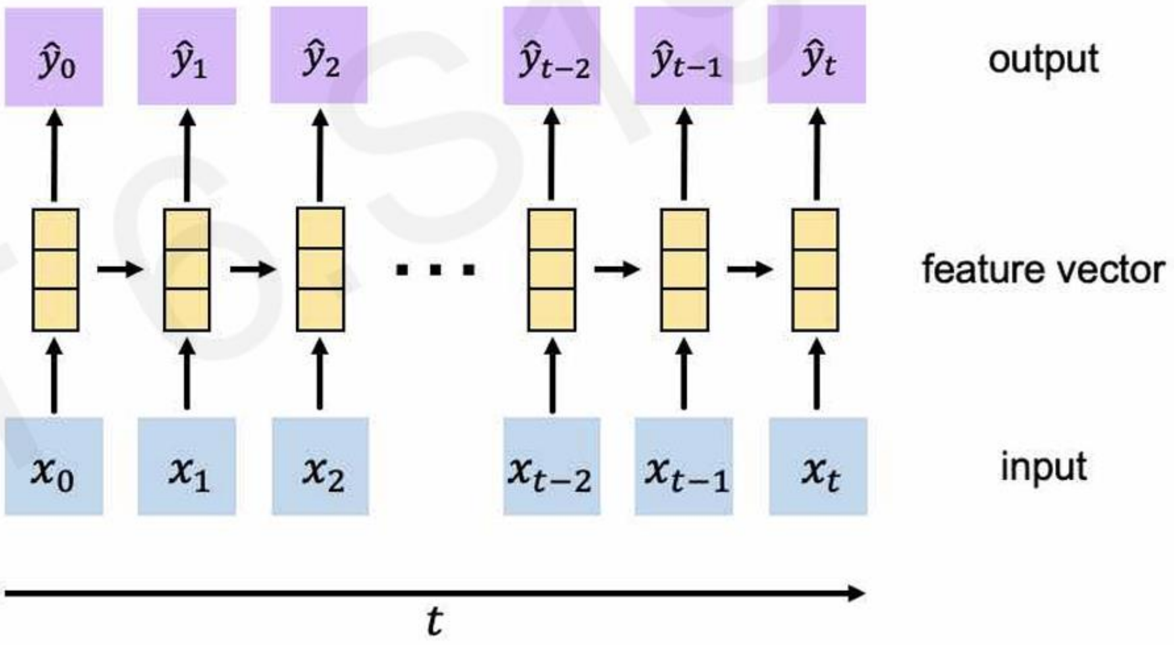


# Goal of Sequence Modeling

RNNs: recurrence to model sequence dependencies

## Limitations of RNNs

- Encoding bottleneck
- Slow, no parallelization
- Not long memory

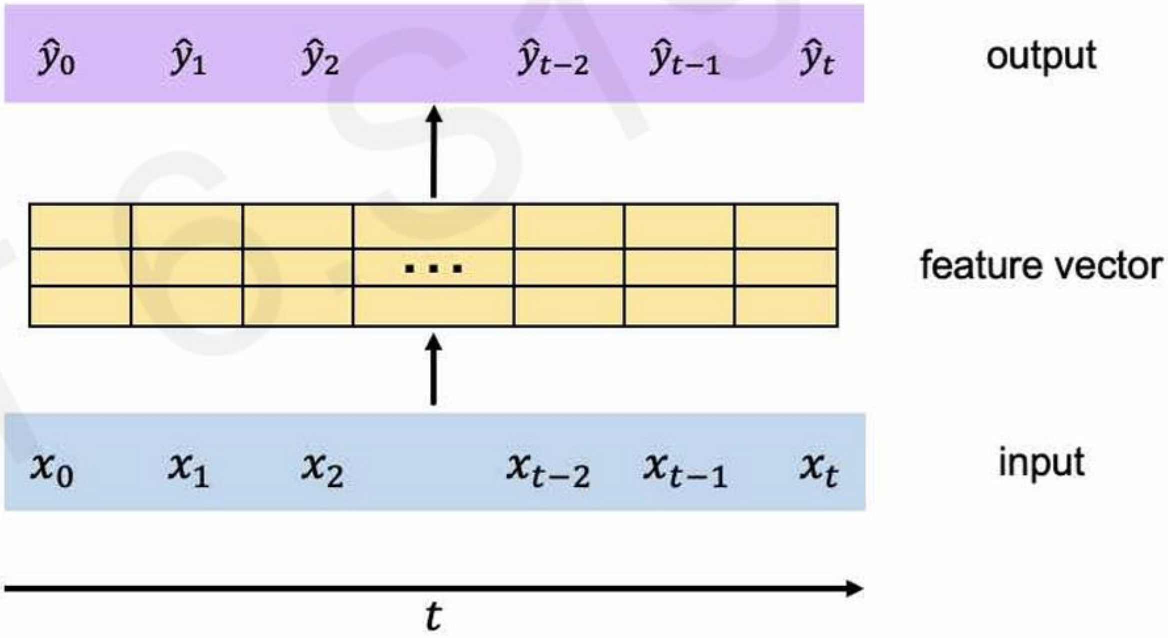


# Goal of Sequence Modeling

Can we eliminate the need for recurrence entirely?

## Desired Capabilities

- Continuous stream
- Parallelization
- Long memory



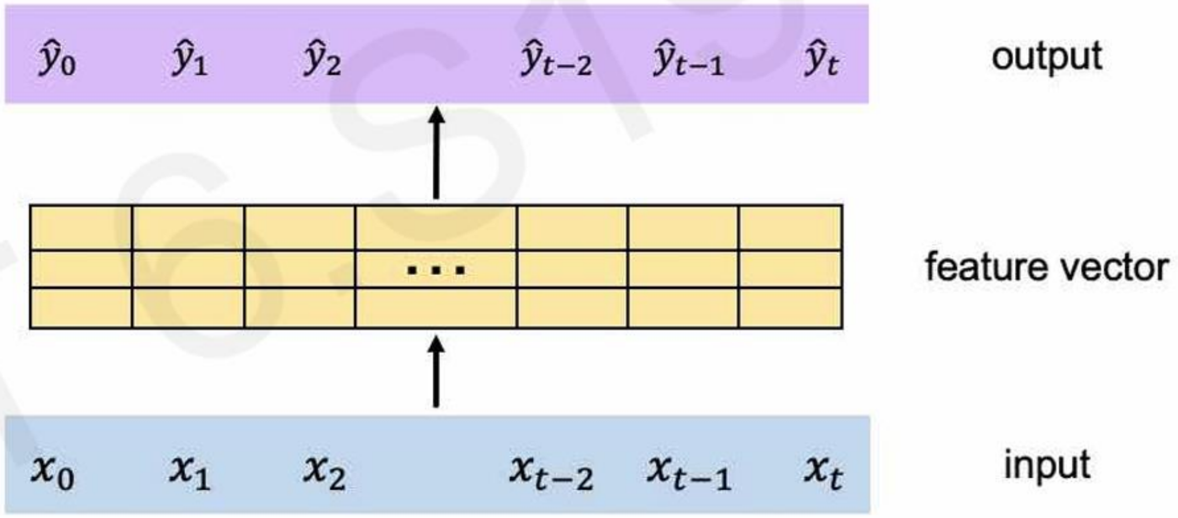
# Goal of Sequence Modeling

Idea 1: Feed everything into dense network

- ✓ No recurrence
- ✗ Not scalable
- ✗ No order
- ✗ No long memory

Idea: Identify and attend to what's important

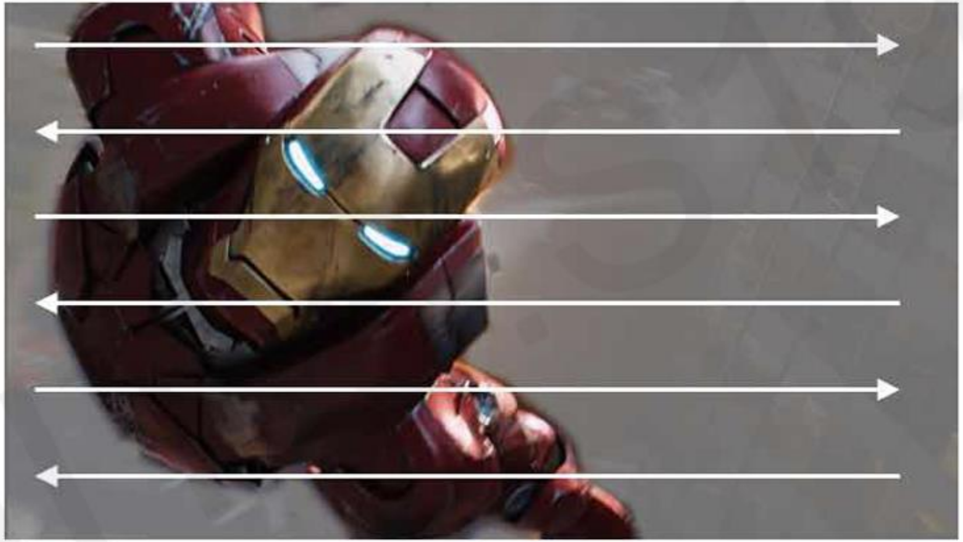
**Can we eliminate the need for recurrence entirely?**



# Attention Is All You Need

# Intuition Behind Self-Attention

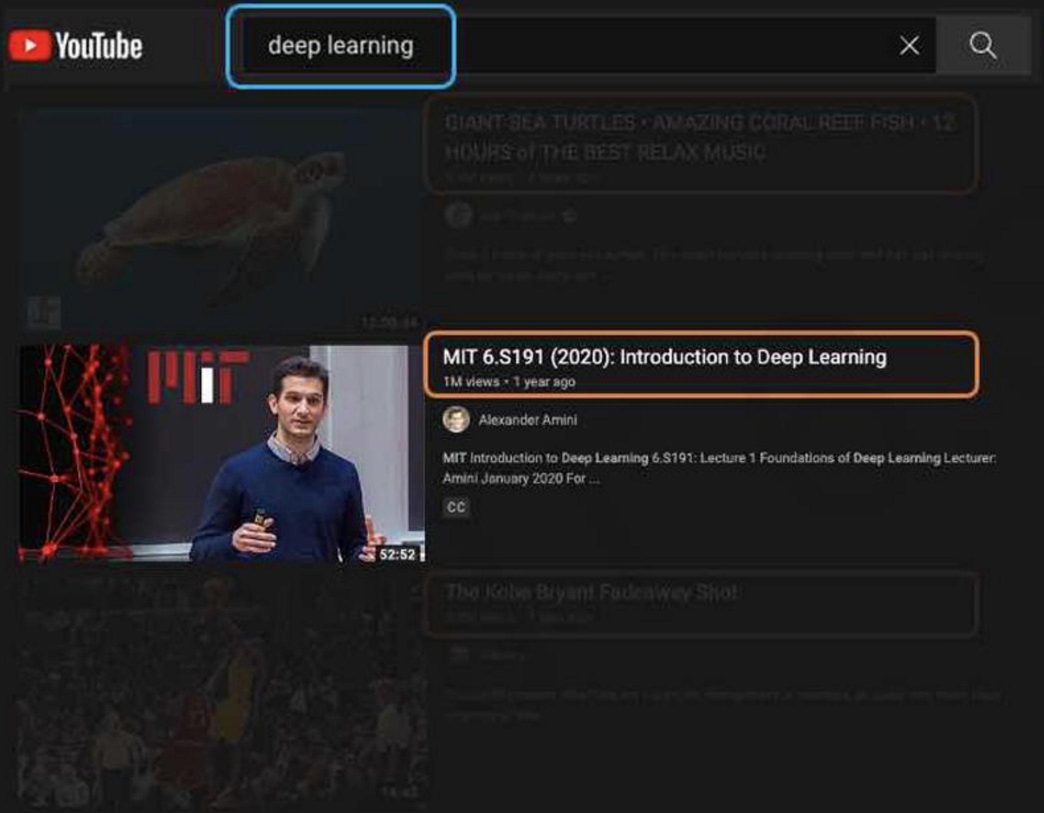
Attending to the most important parts of an input.



1. Identify which parts to attend to
2. Extract the features with high attention

Similar to a search problem!

# Understanding Attention with Search




Query (Q)

Key ($K_1$)

Key ($K_2$)

Key ($K_3$)

How similar is the key to the query?

1. **Compute attention mask:** how similar is each key to the desired query?

# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

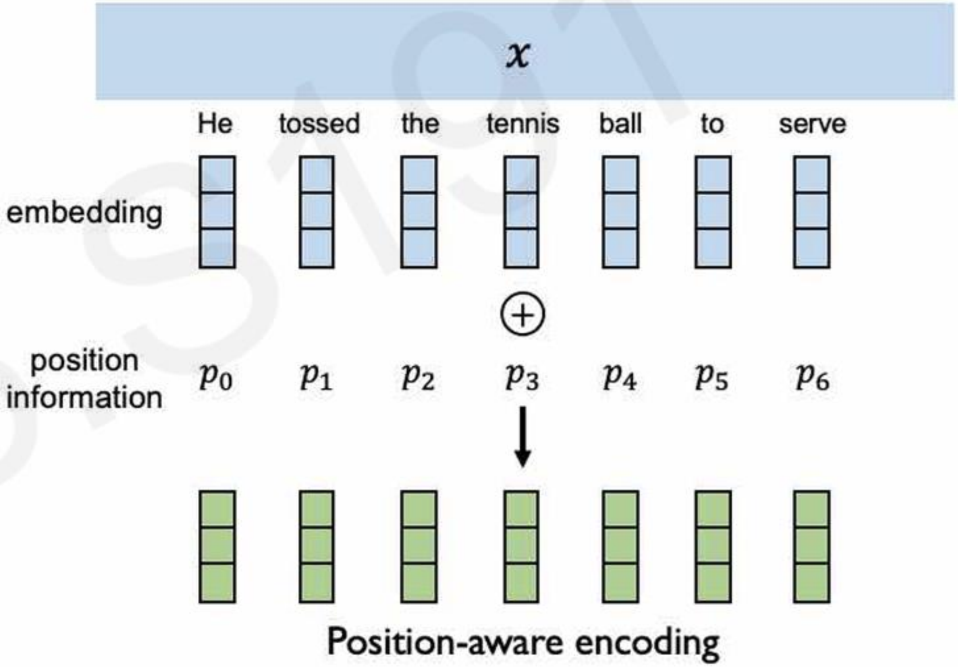
1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features with high attention**


**Data is fed in all at once! Need to encode position information to understand order.**

# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features with high attention**

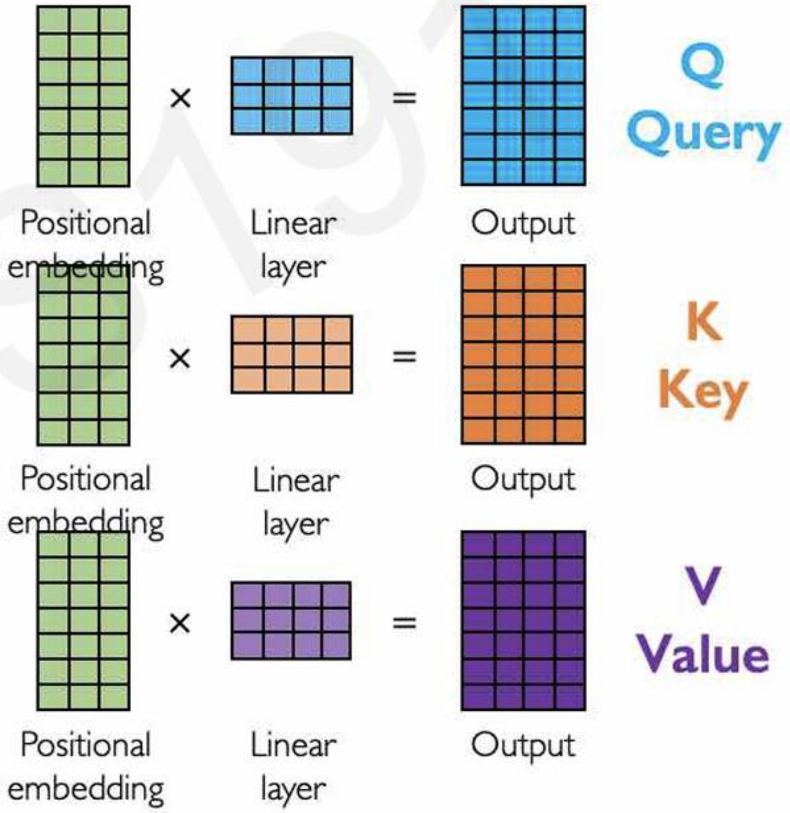



**Data is fed in all at once! Need to encode position information to understand order.**

# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features** with high **attention**



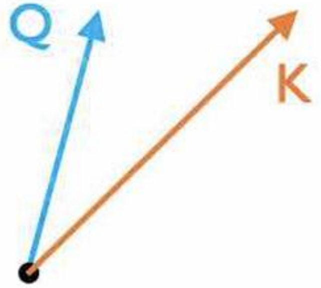
# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract features with high attention

**Attention score**: compute pairwise similarity between each query and key

**How to compute similarity between two sets of features?**




Dot product → Scale

$$\boxed{\frac{Q \cdot K^T}{\text{scaling}}}$$

Similarity metric

Also known as the "cosine similarity"

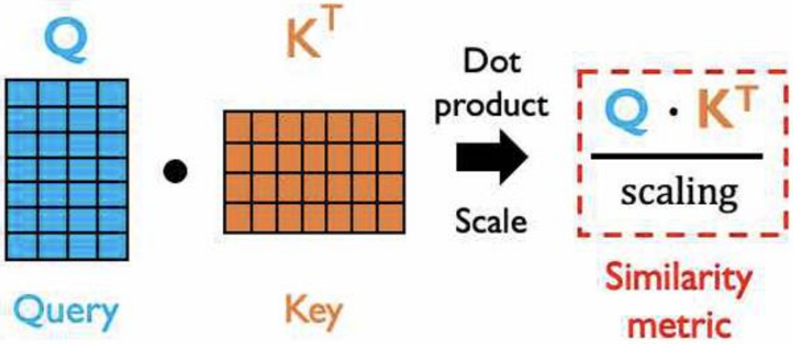
# Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract features with high attention

**Attention score**: compute pairwise similarity between each query and key

How to compute similarity between two sets of features?




Also known as the "cosine similarity"

Remember embeddings are based on which words typically appear together.

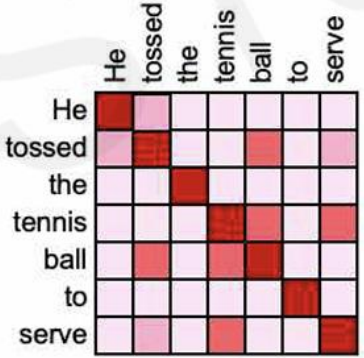
So, computing the “similarity” between two words is actually estimating how “related” they are, in the sense of how often they tend to appear with similar words.

# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract features with high attention

**Attention weighting: where to attend to!**
**How similar is the key to the query?**




$$softmax\left(\frac{Q \cdot K^T}{scaling}\right)$$

Attention weighting

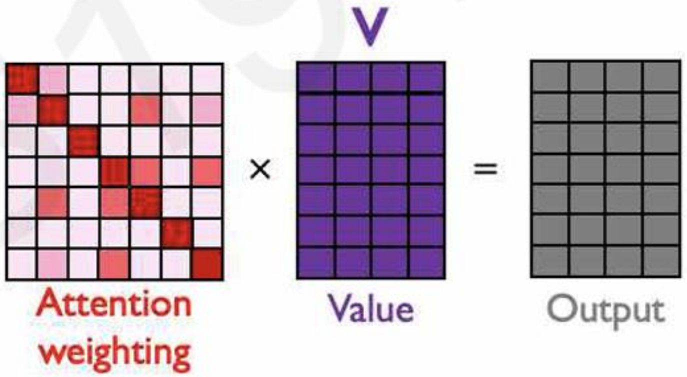
That “similarity matrix” now includes information about the relationship between different words.

# Learning Self-Attention with Neural Networks

Goal: identify and attend to most important features in input.

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features with high attention**

Last step: self-attend to extract features



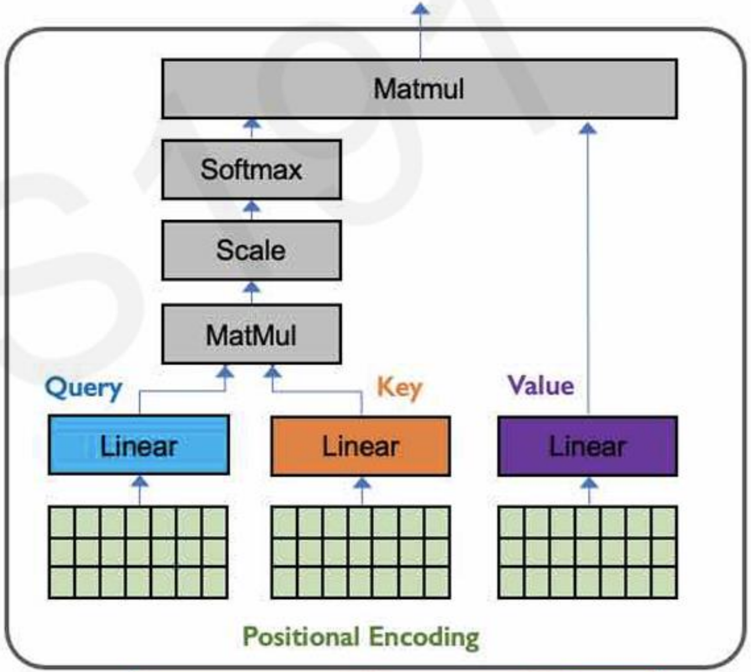

$$softmax\left(\frac{Q \cdot K^T}{scaling}\right) \cdot V = A(Q, K, V)$$

# Learning Self-Attention with Neural Networks

**Goal: identify and attend to most important features in input.**

1. Encode **position** information
2. Extract **query, key, value** for search
3. Compute **attention weighting**
4. Extract **features with high attention**

**These operations form a self-attention head that can plug into a larger network. Each head attends to a different part of input.**




$$softmax\left(\frac{\boxed{Q \cdot K^T}}{scaling}\right) \cdot V$$

# Recipe of the Day!

Pink Champagne Cake

# Administrivia

Last Lab this Friday!

PSet2, Q1: just after padding

# Problems with RNNs

# Problems with RNNs

- Recall the formula for a generic RNN:

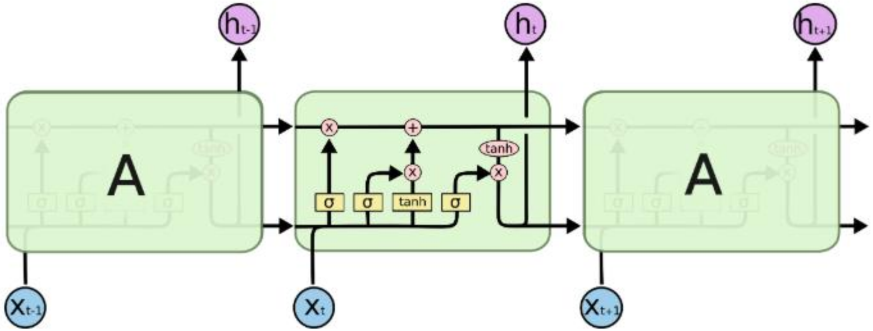
$$h_t = f(X_tW + h_{t-1}U + b)$$

- 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

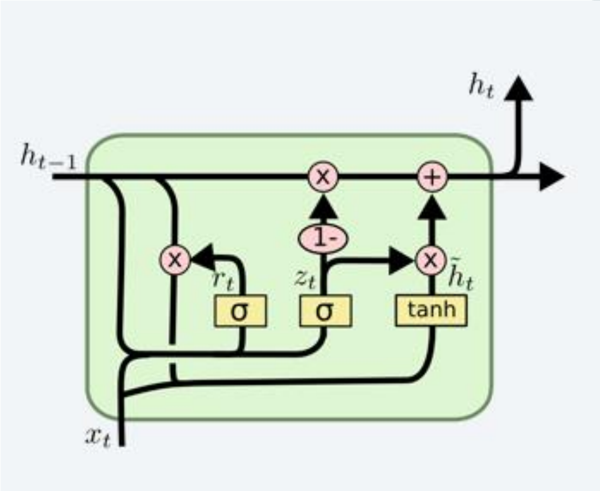
- Sepp Hochreiter and Jürgen Schmidhuber proposed the [long short term memory (LSTM) hidden unit in 1997]
- LSTMs selectively modify the inputs to produce “well-behaved” outputs, fixing the gradient issues
- Can model very long sequences without having the gradients vanish or explode



The repeating module in an LSTM contains four interacting layers.

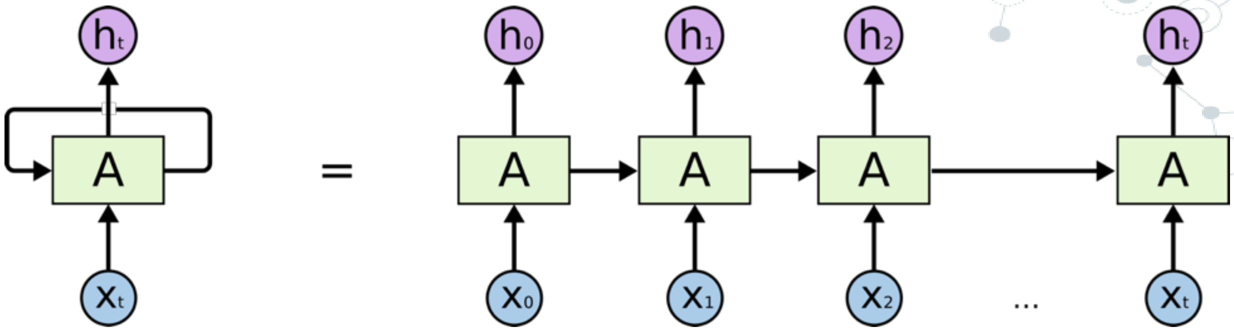
# Fixing RNNs

- 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

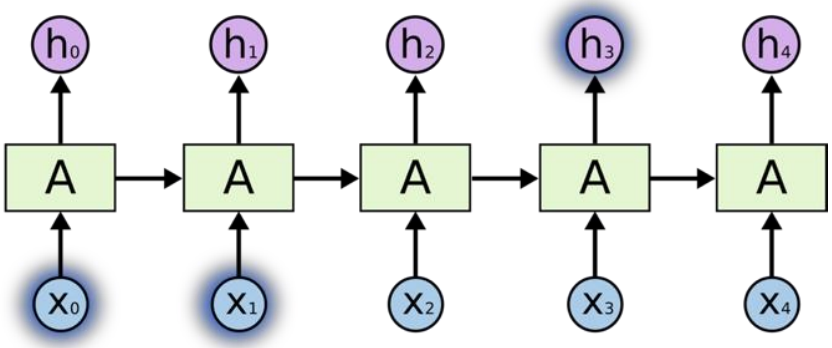


# Long Short Term Memory (LSTM)

An “unrolled” RNN



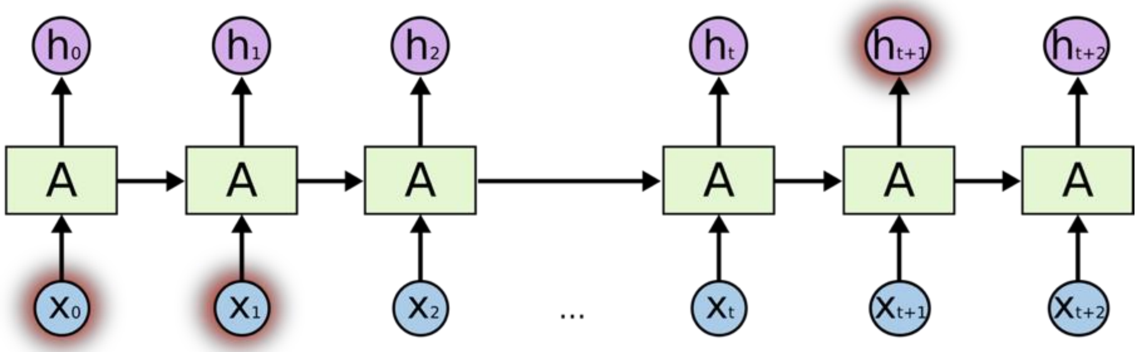

RNN where the output $h_3$ only depends on the input from $X_0$ and $X_1$
(The relevant information needed at $h_3$ comes from $X_0$ and $X_1$)
The gap between relevant information and the place it is needed is small



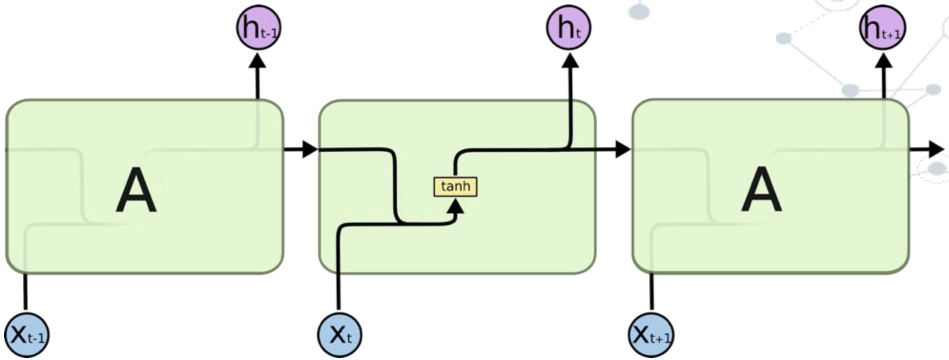

http://colah.github.io/posts/2015-08-Understanding-LSTMs/

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

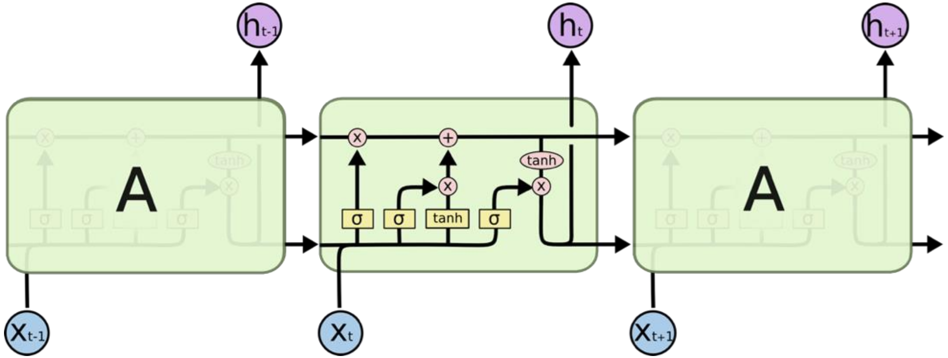
This is an example of a **long-term dependency** - RNNs struggle to learn to make connections when there are large gaps between the relevant information and where it is needed

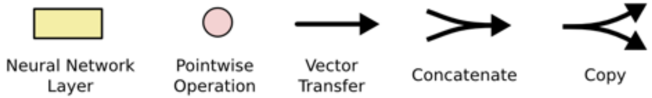


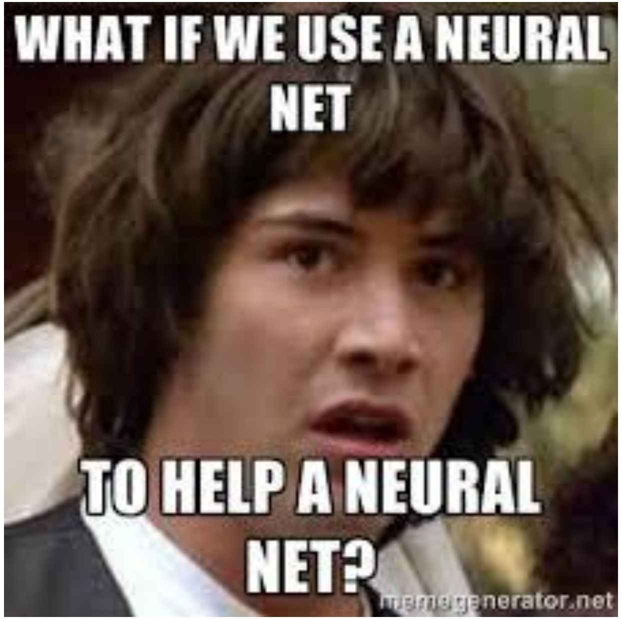
Simple, “vanilla” RNN:

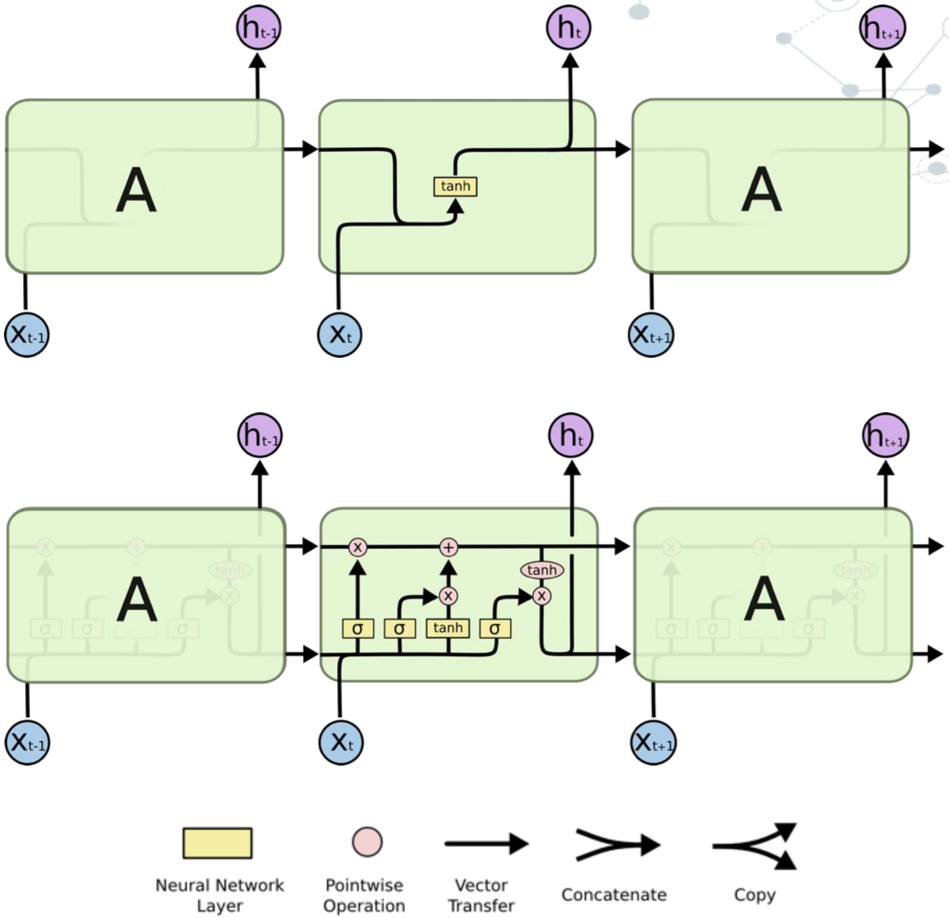



RNN with LSTM units:

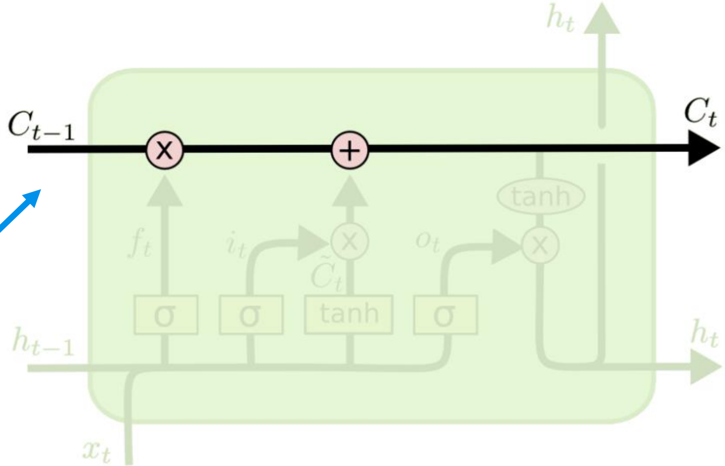


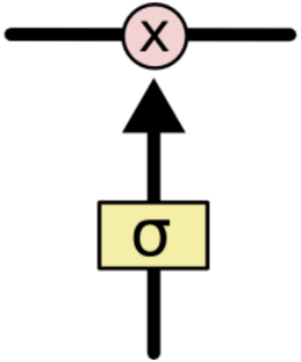




WHAT IF WE USE A NEURAL NET
TO HELP A NEURAL NET?
memegenerator.net


A
tanh
σ
Neural Network Layer
Pointwise Operation
Vector Transfer
Concatenate
Copy

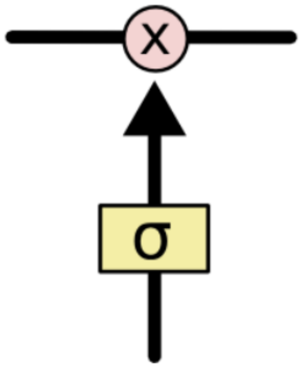
- LSTMs were explicitly designed to avoid the long-term dependency problem
- The key to LSTMs is the ability to let certain information through and carry it until it is deemed no longer useful (which may not happen)
- Information is carried through the sequence in the **cell state**, which acts as a 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



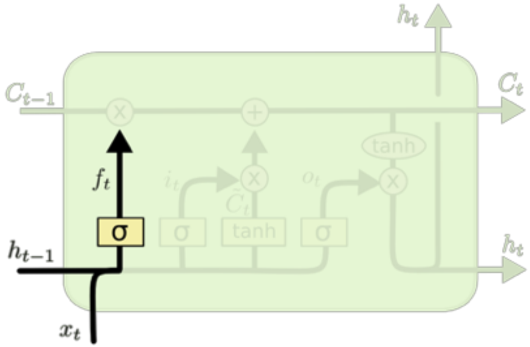



# Gates

- 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

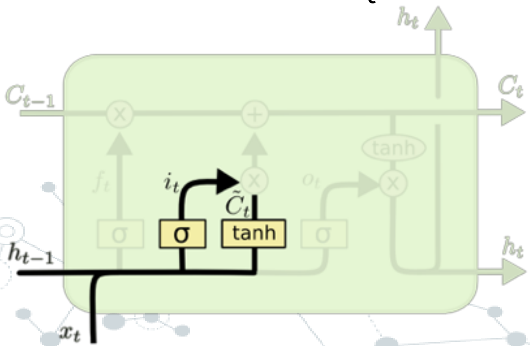


**Step 1: Forget Gate** - Determine how much of the previous state should affect the current state based on the current observed input $x_t$



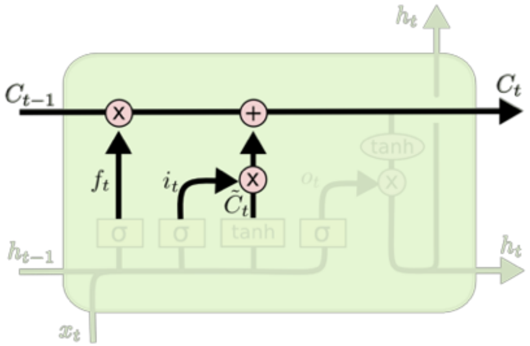

$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \ + \ b_f\right)$$

**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 ($h_{t-1}$).



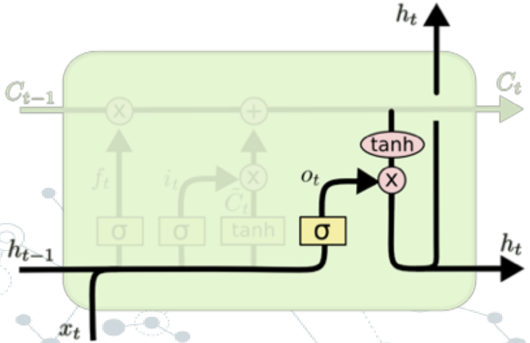

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \ + \ b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \ + \ b_C)$$

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




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

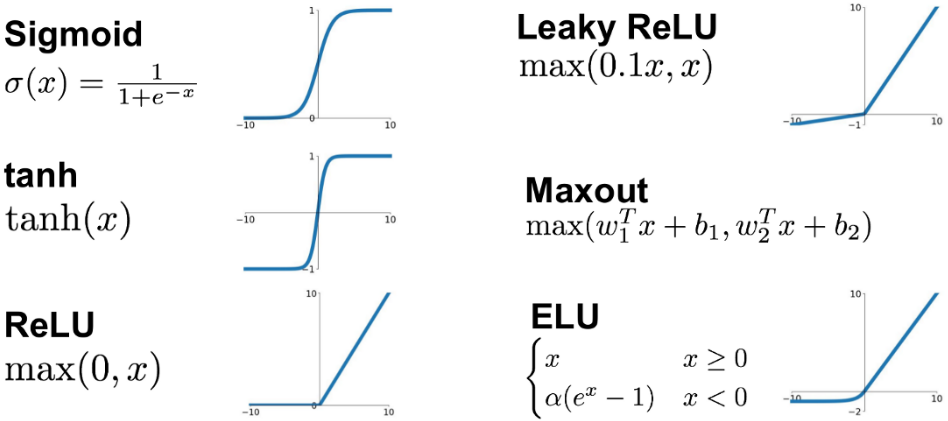
**Step 4: Compute Unit Output** - determine which parts of the cell state will be used as unit output. Output is a filtered version of the cell state.




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

# Why tanh?

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



# 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

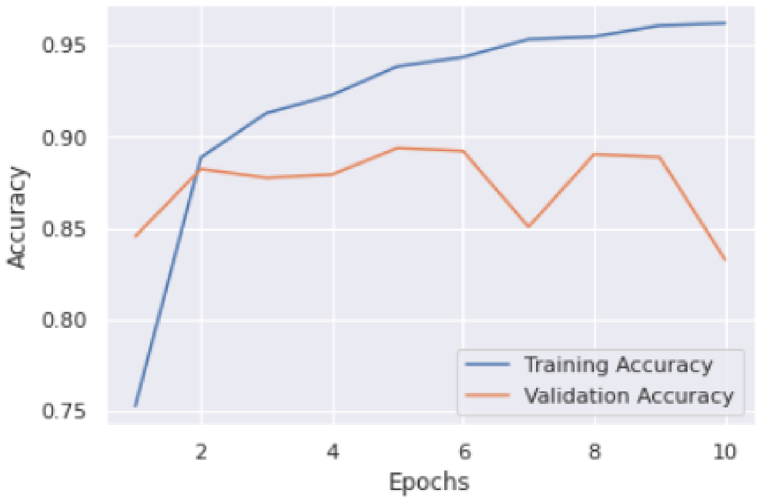
# LSTM in Keras

- 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

```
model = tf.keras.models.Sequential([
  tf.keras.layers.Embedding(max_features, 32),

  tf.keras.layers.LSTM(32),

  tf.keras.layers.Dense(1, activation='sigmoid')
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='binary_crossentropy',
              metrics=['accuracy'])

history = model.fit(input_train, y_train,
                    epochs=10,
                    batch_size=128,
                    validation_split=0.2)
```

# LSTM in Keras

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

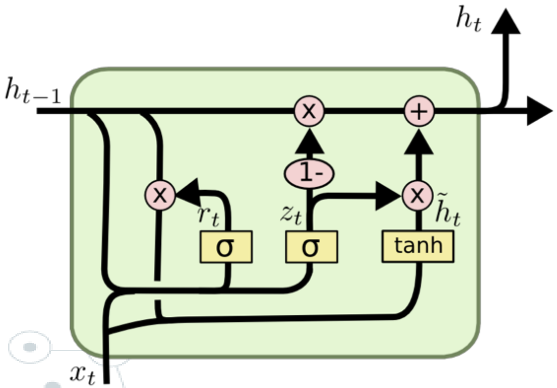



# Gated Recurrent Unit (GRU)

# 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)




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

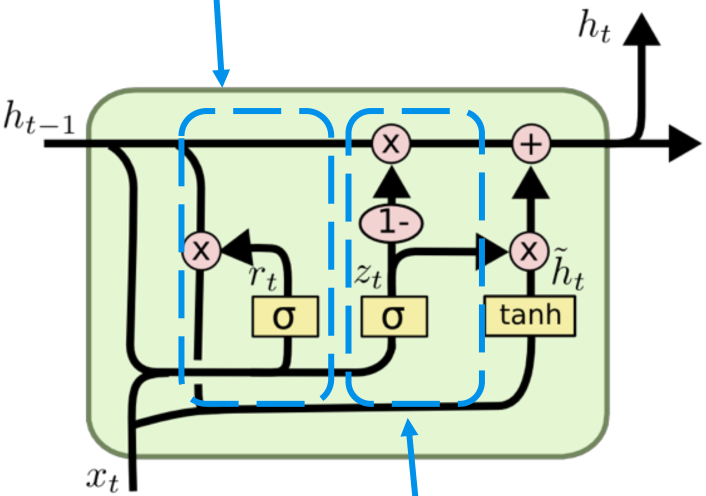
$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

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

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

# GRU

Reset gate (how much past information to forget)




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

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

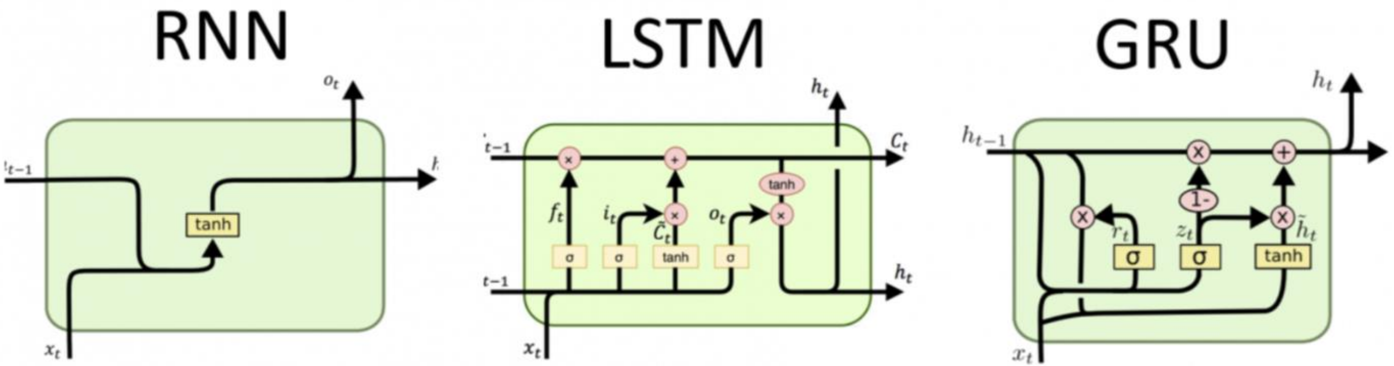
$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

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

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

# 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



# 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

```python
fname = 'path/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 (%)"',
420551
```

Number of rows (observations)

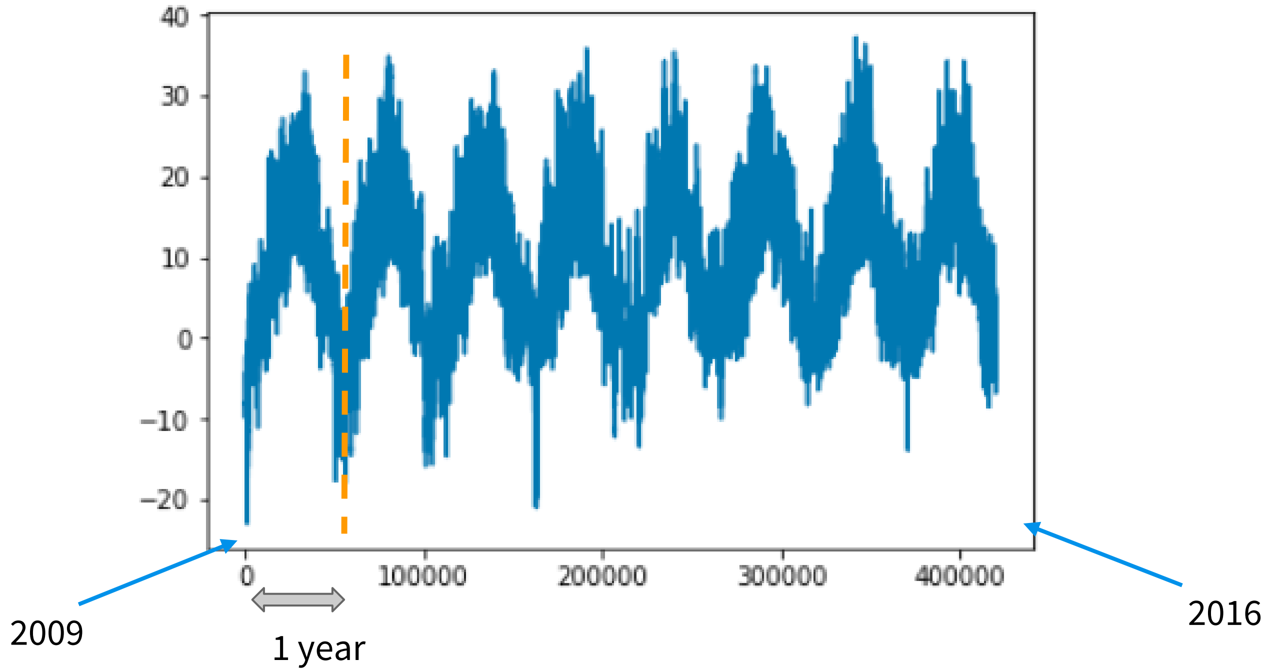
Number of columns (-1 for unnecessary 1st column: date/time)

```python
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)
```

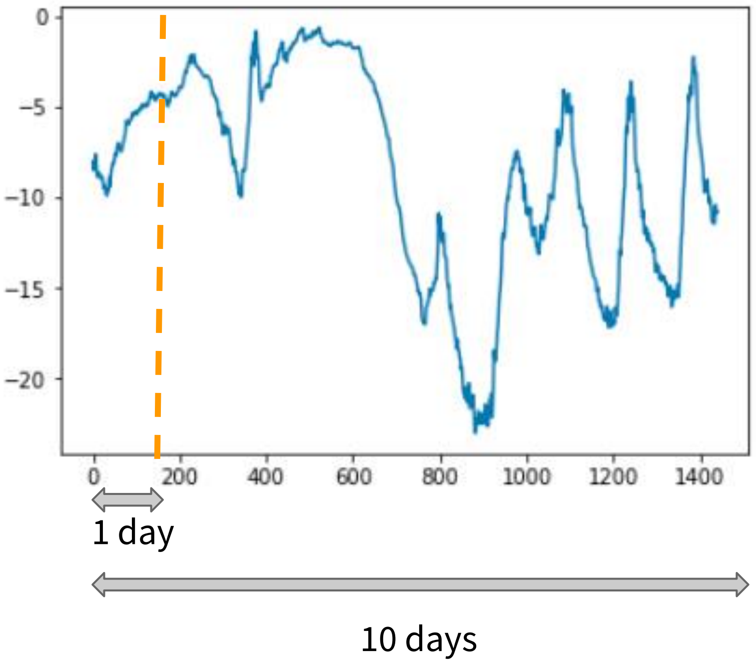
# Temperature over time



# 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 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

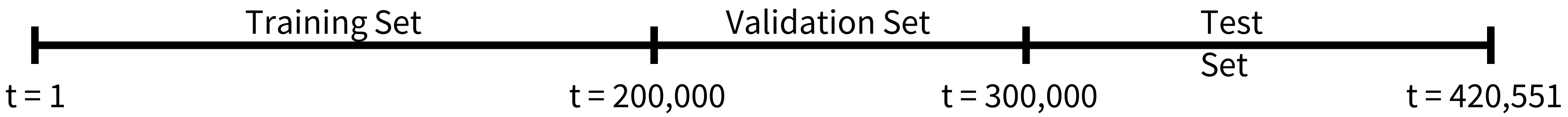
- **min_index** and **max_index**: Indices in the data array that delimit which timesteps to draw from. This is useful for keeping a segment of the data for validation and another one for testing.

- **shuffle**: Whether to shuffle our samples or draw them in chronological order

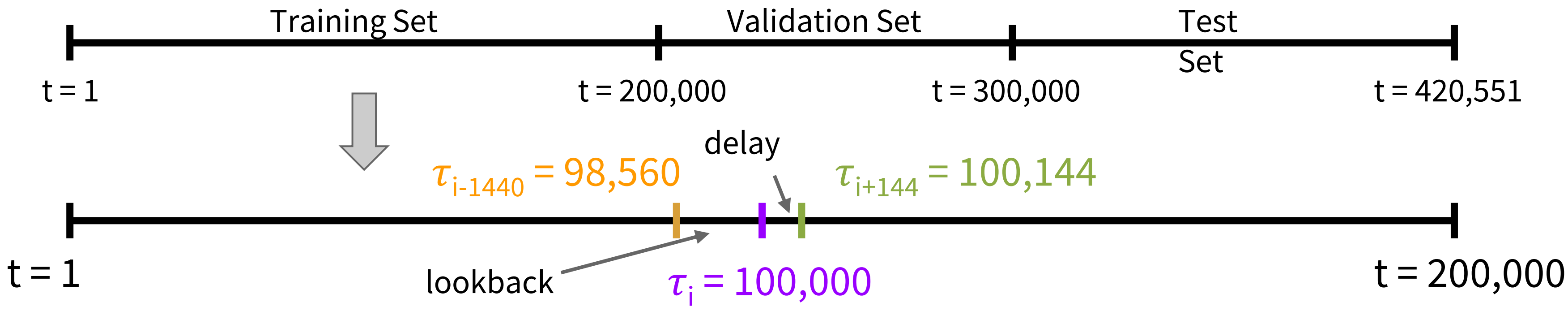
- **batch_size**: The number of samples per batch

- **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

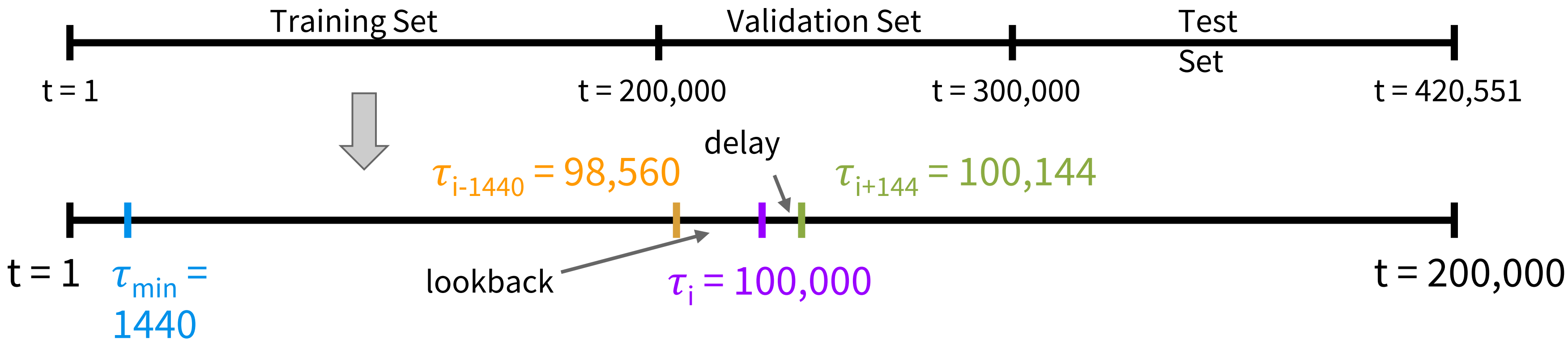

Training Set
Validation Set
Test Set
t = 1
t = 200,000
t = 300,000
t = 420,551

# Timeline



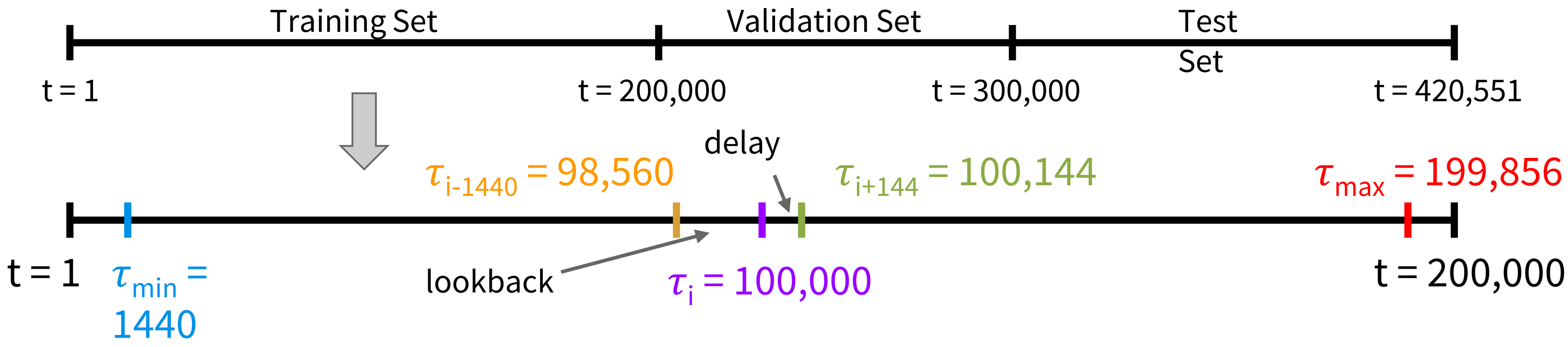
Training Set
Validation Set
Test Set
t = 1
t = 200,000
t = 300,000
t = 420,551
delay
$\tau_{i-1440} = 98{,}560$
$\tau_{i+144} = 100{,}144$
t = 1
lookback
$\tau_i = 100{,}000$
t = 200,000

# Timeline




$\tau_{min}$: we need all 10 days worth of past data to predict the temperature for the next time point. Thus, the minimum value $\tau_i$ 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.

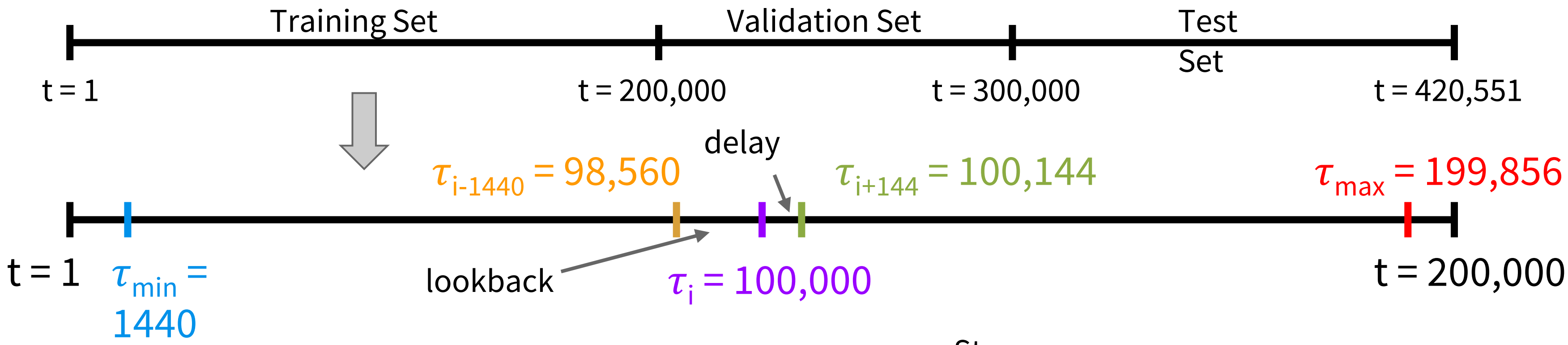
# Timeline




$\tau_{min}$: we need all 10 days worth of past data to predict the temperature for the next time point. Thus, the minimum value $\tau_i$ 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.

$\tau_{max}$: 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 $\tau_i$ 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




$\tau_{min}$: we need all 10 days worth of past data to predict the temperature for the next time point. Thus, the minimum value $\tau_i$ 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.

$\tau_{max}$: 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 $\tau_i$ 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, $\tau_i$, between $\tau_{min}$ and $\tau_{max}$
2. Keep 10 days of data prior to $\tau_i$ and 24 hours after $\tau_i$.
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

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

```python
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℃.

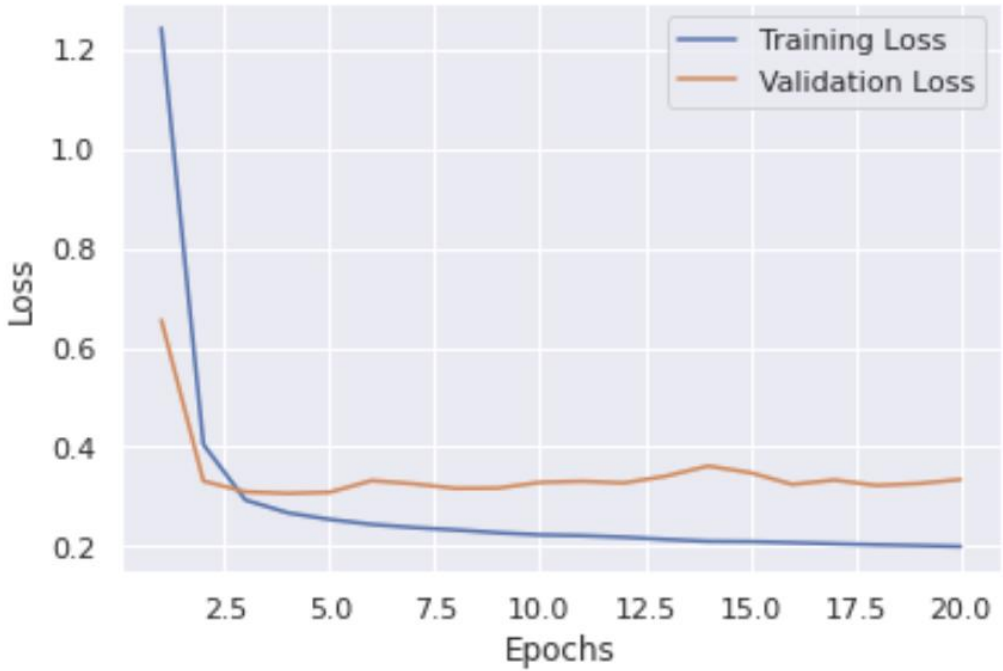
That's a fairly large average absolute error – now the task is to leverage our knowledge of deep learning to do better.

# 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

```
model = keras.Sequential([
  layers.Flatten(input_shape=(lookback // step, float_data.shape[-1])),

  layers.Dense(32, activation='relu'),
  layers.Dense(1)
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='mae')

history = model.fit(train_gen,
                    steps_per_epoch=500,
                    epochs=20,
                    validation_data=val_gen,
                    validation_steps=val_steps)
```

# Temperature Forecasting - Simple Model



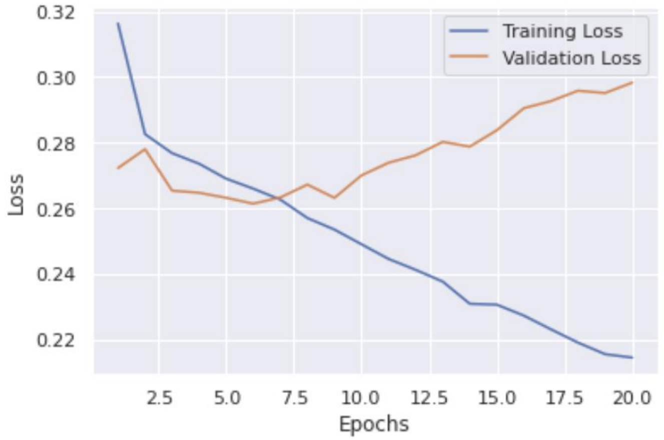

We get MAEs above 0.3 for the validation loss - worse than our benchmark.

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

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.

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

  layers.Dense(1)
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='mae')

history = model.fit(train_gen,
                    steps_per_epoch=500,
                    epochs=20,
                    validation_data=val_gen,
                    validation_steps=val_steps)
```



# 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

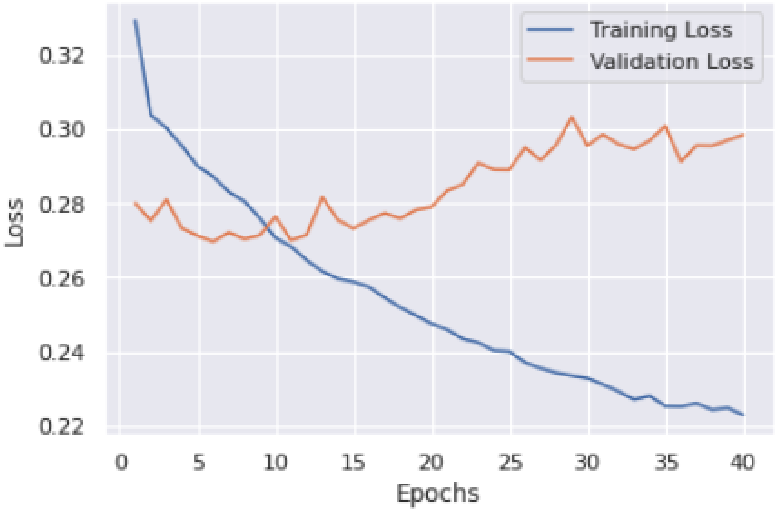
- This allows the network to properly propagate its learning error rate through time - a temporally random dropout pattern would disrupt the error signal and hinder the learning process

- 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

# Recurrent Dropout in Keras

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

  layers.Dense(1)
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
               loss='mae')

history = model.fit(train_gen,
                    steps_per_epoch=500,
                    epochs=40,
                    validation_data=val_gen,
                    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

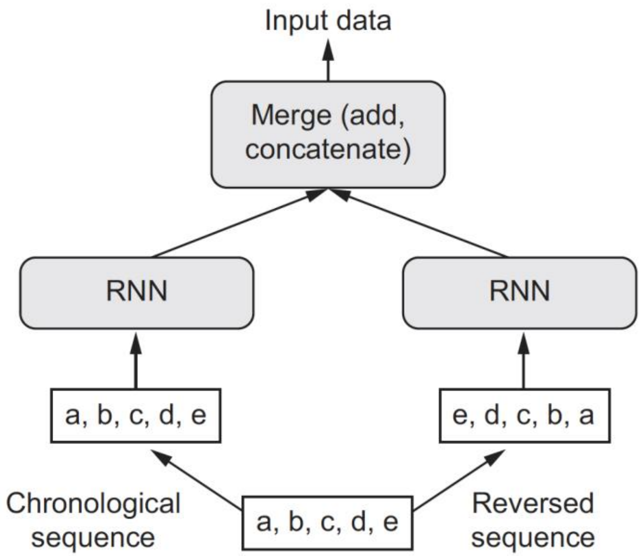
```
model = keras.Sequential([
  layers.GRU(32,
             dropout = 0.1,
             recurrent_dropout=0.5,
             return_sequences=True,
              input_shape=(None, float_data.shape[-1])),
  layers.GRU(64, activation='relu',
             dropout = 0.1,
             recurrent_dropout=0.5),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

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='mae')

history = model.fit(train_gen,
                    steps_per_epoch=500,
                    epochs=40,
                    validation_data=val_gen,
                    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

# Bidirectional Recurrent Layers

# Bidirectional RNNs

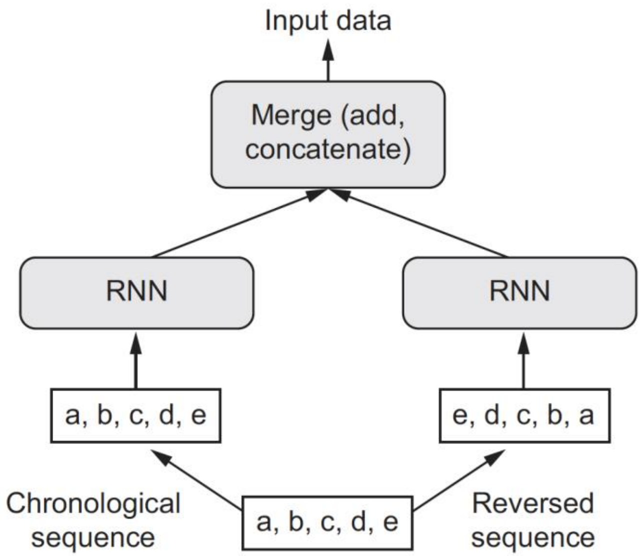
- A bidirectional RNN (BRNN) can offer greater performance on certain tasks
- Frequently used in natural-language processing (NLP)
- BRNNs exploit the order sensitivity of RNNs



# 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).

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

  layers.Dense(1)
])

model.compile(optimizer = tf.keras.optimizers.RMSprop(),
              loss='mae')

history = model.fit(train_gen_reverse,
                    steps_per_epoch=500,
                    epochs=20,
                    validation_data=val_gen_reverse,
                    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