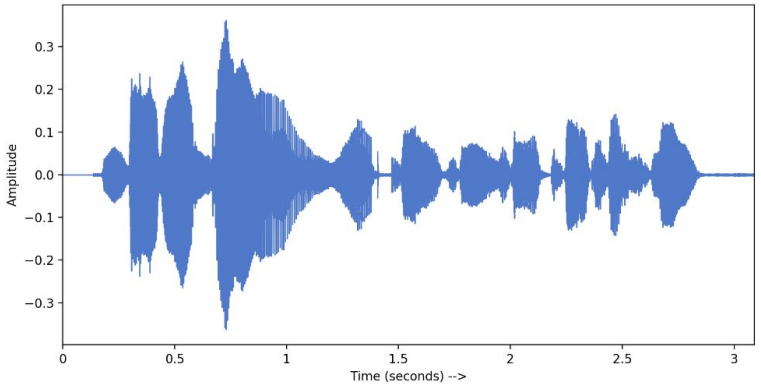
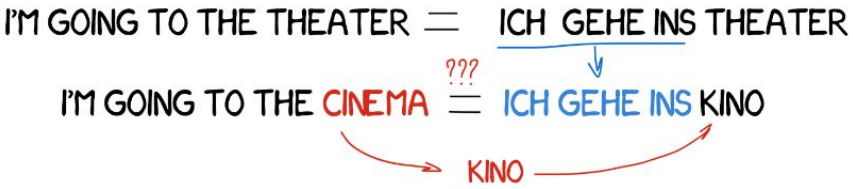
# APS360: Applied Fundamentals of Deep Learning

Week 8: Recurrent Neural Network - Part II

# Sequence Learning

Many real world problems deal with **inputs/outputs with varying sizes**




# Sequence Learning

In an image, we do not want to learn different weights for every pixel

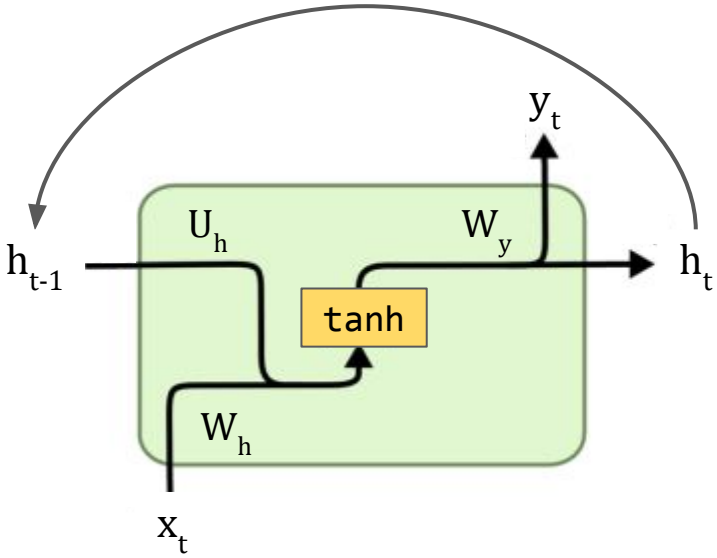
- CNNs use **convolutional filters** with **parameter sharing**
- CNN reuses convolutional filters for every pixel

In a sequence, we do not want to learn different weights for every token

- RNNs use a **shared neural network** to **update hidden state**
- Reuse the RNN module for every token in the sequence
- Keep the **context** of the previous tokens encoded in the hidden state ($\mathbf{h}$)

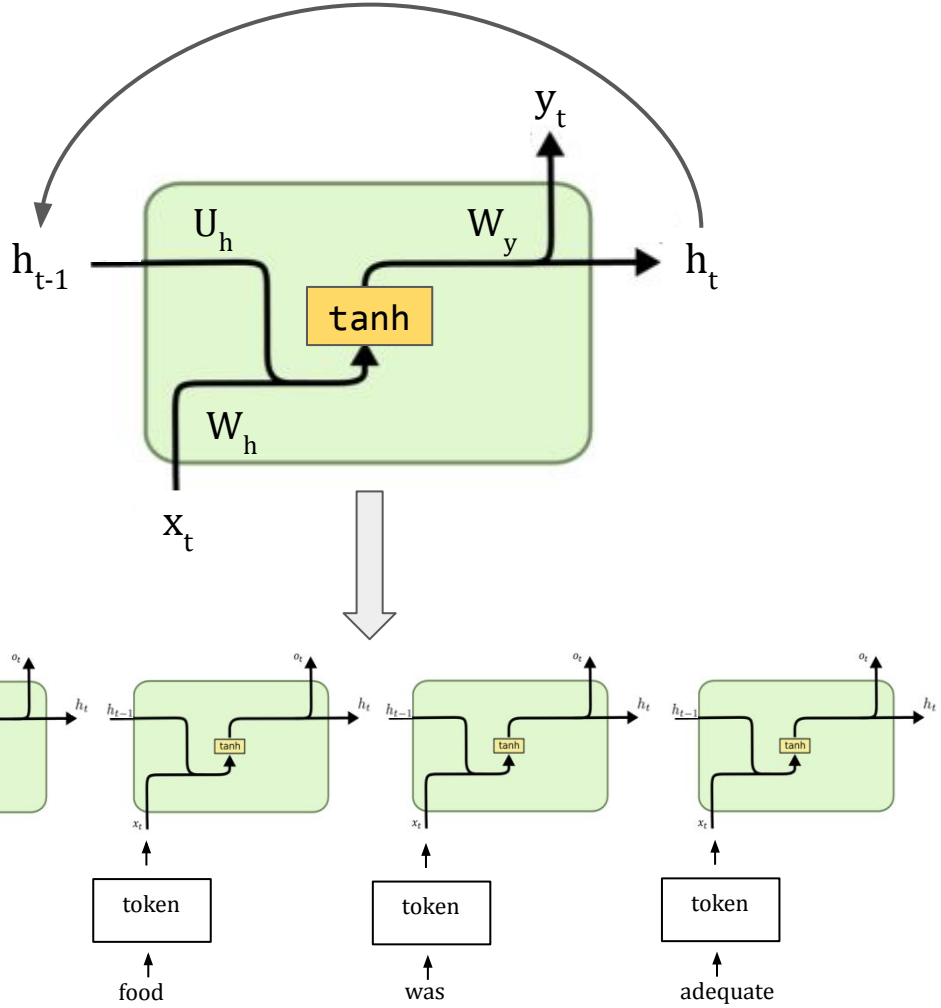
# RNN

If we consider the **concatenated input/hidden** and **output/hidden** vectors as simply input/output, forward path in RNN is simply a **fully-connected NN**

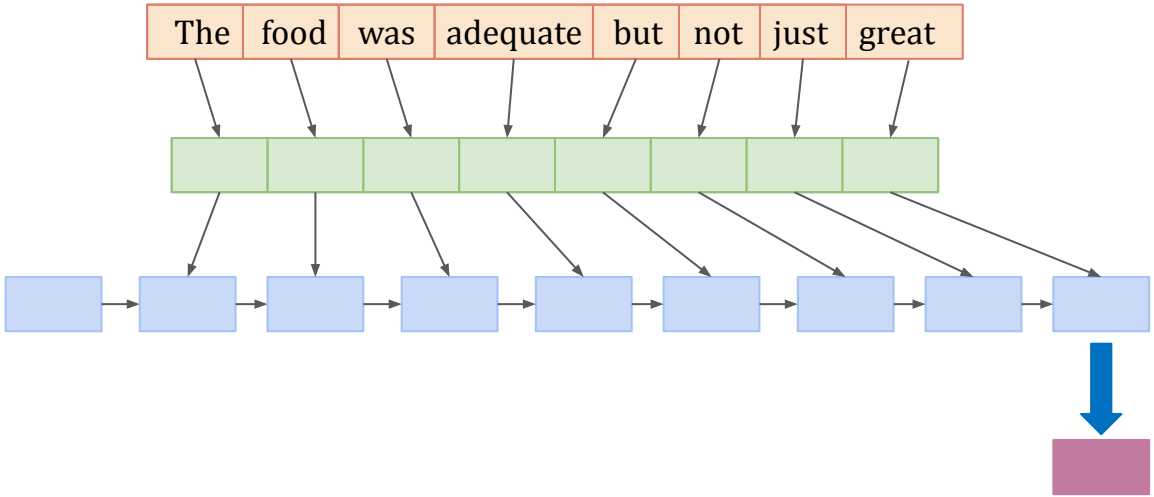



$$h_t = \sigma_h(W_h x_t + U_h h_{t-1} + b_h)$$
$$y_t = \sigma_y(W_y h_t + b_y)$$

# Unrolling RNNs



# Sequence-Level Predictions

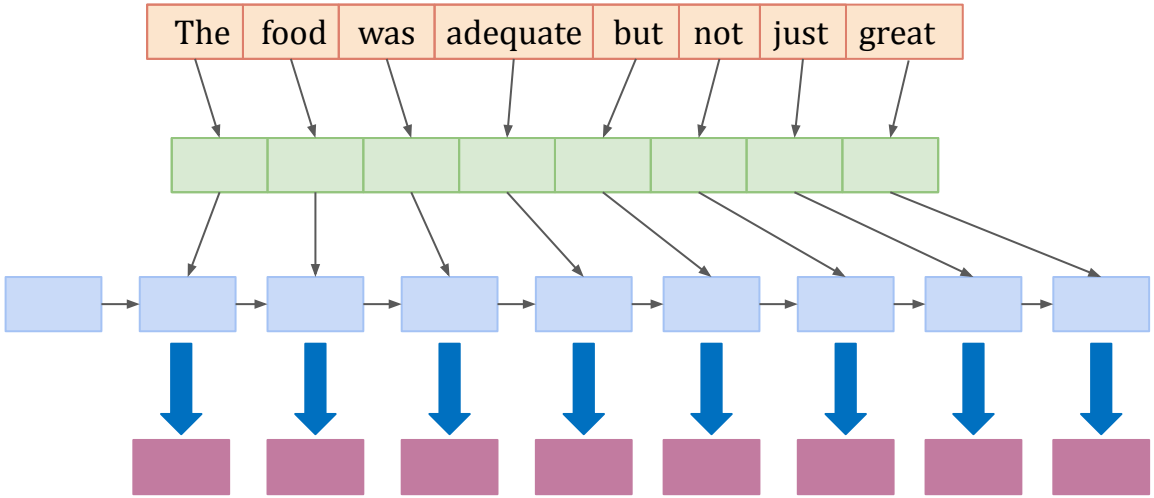



$x_n$ = GloVe(OneHot(input))

$h_n$ = Update($h_{n-1}$, $x_n$)

$y$ = predict($h_{last}$)

# Token-Level Predictions




$x_n$ = GloVe(OneHot(input))

$h_n$ = Update($h_{n-1}$, $x_n$)

$y_n$ = predict($h_n$)

# Limitations of Vanilla RNNs

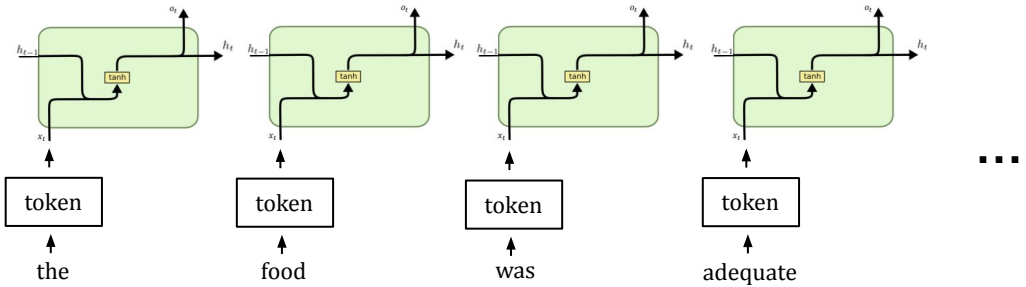
# The Problem

What happens to RNNs unrolled onto a **long sequence** ?

- RNNs can be very **deep** → Depth = Length of sequence

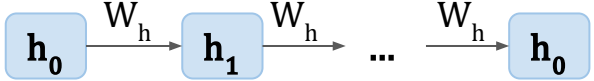
There are two related problems with vanilla RNNs:

- Not good at modelling **long-term dependencies**
- Hard to train due to **vanishing/exploding gradients**



# Exploding/Vanishing Gradients

Suppose update function is a simple linear model. For simplicity, let's ignore inputs:



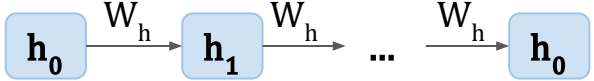

$$h_t = W_h h_{t-1}$$

We can write this for all time-steps as:

$$h_t = (W_h)^t h_0$$

# Exploding/Vanishing Gradients

Suppose update function is a simple linear model. For simplicity, let's ignore inputs:




$$h_t = W_h h_{t-1}$$

We can write this for all time-steps as:

$$h_t = (W_h)^t h_0$$

Then we have:

- Exploding gradients $h_t \to \infty \quad \text{if } |W_h| > 1$
- Vanishing gradient $h_t \to 0 \quad \text{if } |W_h| < 1$

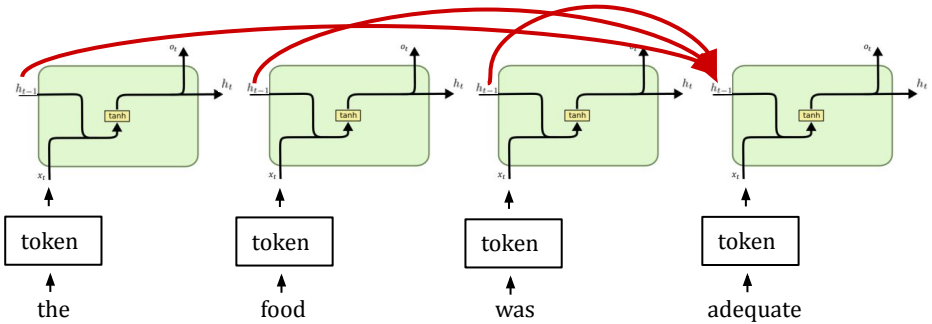
# Tackling Exploding/Vanishing Gradients

## Gradient clipping → Exploding Gradient

- If gradient is greater than a threshold, set the gradient to threshold

## Skip-connection → Vanishing Gradient

- Ideally we want skip connections to **all previous states** → Too expensive
- We could preserve the hidden state/context over the long term



# LSTMs & GRUs

# Gating Mechanism

- We can approximate skip-connections to all previous states by learning to **weight previous states differently** instead (soft skip-connections)
- We can use **gates** that learn to update the context **selectively**
- Gating mechanism controls how much informations flows through.
- Suppose X is a vector, then we can control how much of X to pass to next step by:

$$f(x) = X.\sigma(X)$$

Sigmoid or Tanh

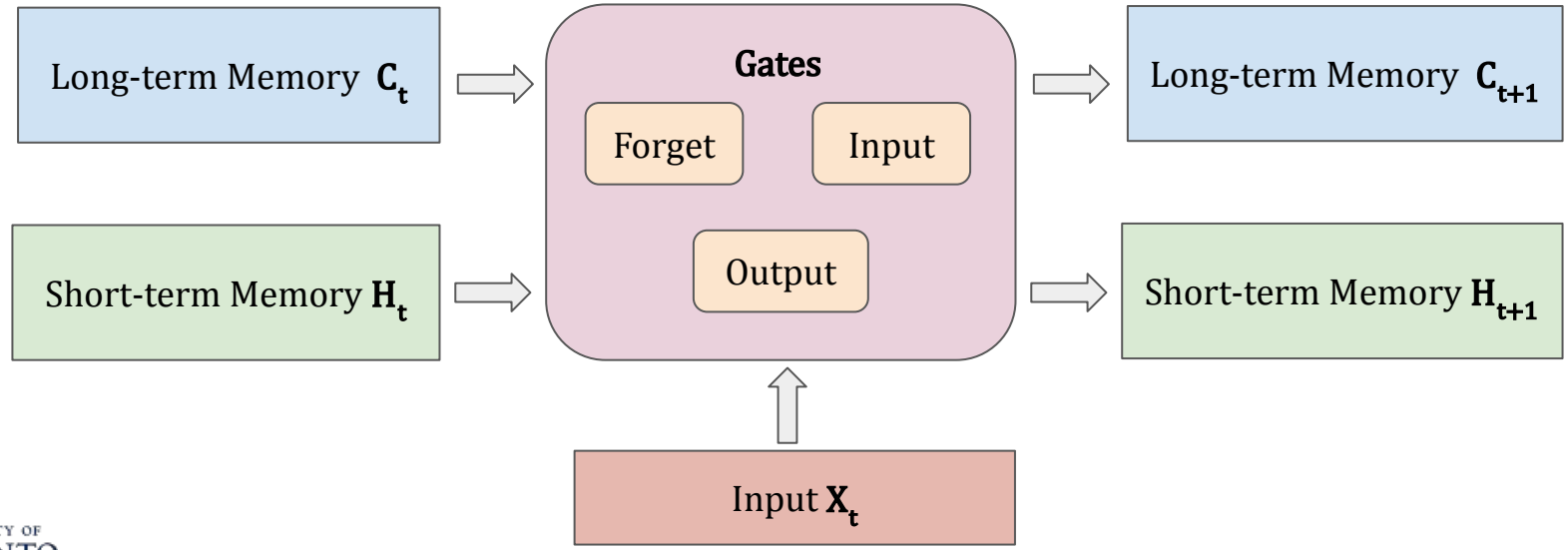
$$f(x) = X.\mathrm{NN}(X)$$

A neural network

# Long Short-Term Memory (LSTM)

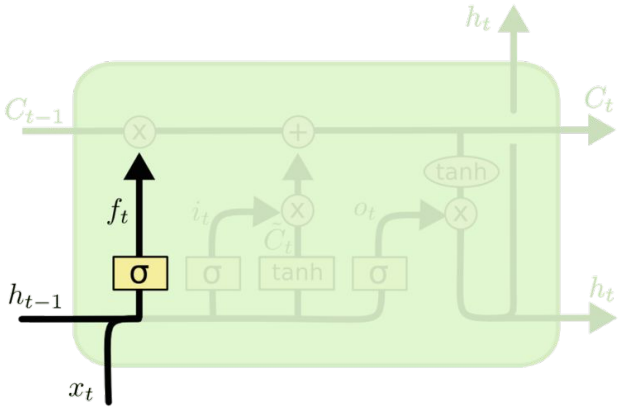
LSTMs consist of a **long-term memory** (cell state) and a **short-term memory** (context or hidden state)

They use three gates to update the memories



# LSTM

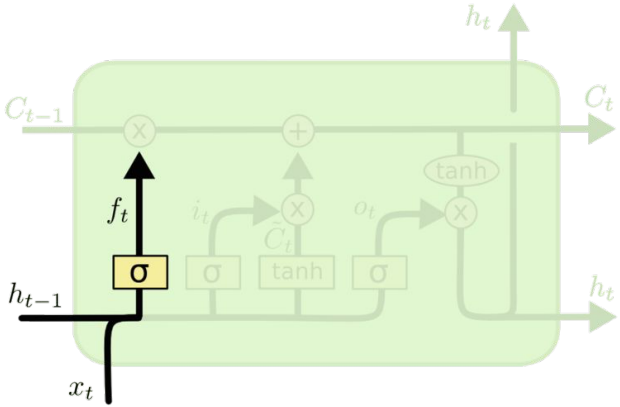
**Forget Gate** (long-term memory) → How much of the past memory should we forget?

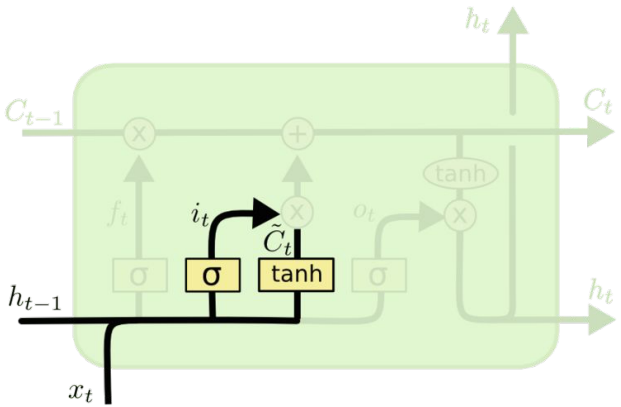



$$f_t = \sigma\left(W_f . [h_{t-1}, x_t] + b_f\right)$$

# LSTM

**Input Gate** (long-term memory)→ How much the current input should contribute to the memory?



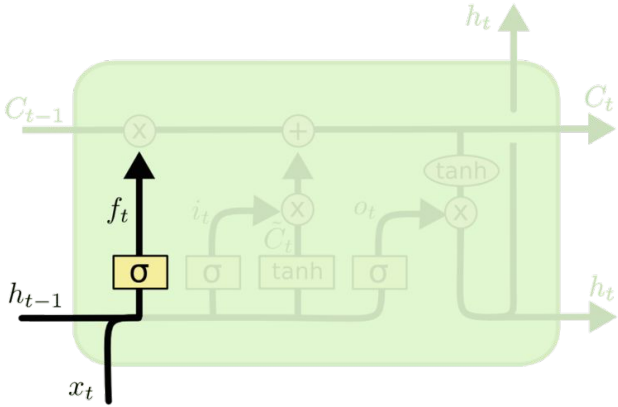

$$f_t = \sigma\left(W_f . [h_{t-1}, x_t] + b_f\right)$$

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

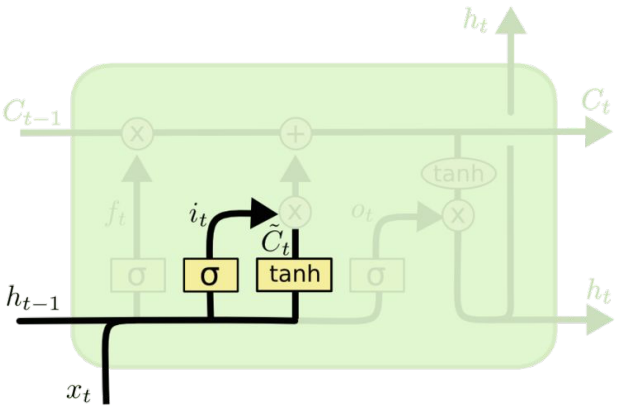
$$\tilde{C}_t = \tanh\left(W_C . [h_{t-1}, x_t] + b_C\right)$$

# LSTM

The **updated long-term memory** is the amount of past that is remembered (decided by forget gate) combined with the memory that was just created (decided by input gate)

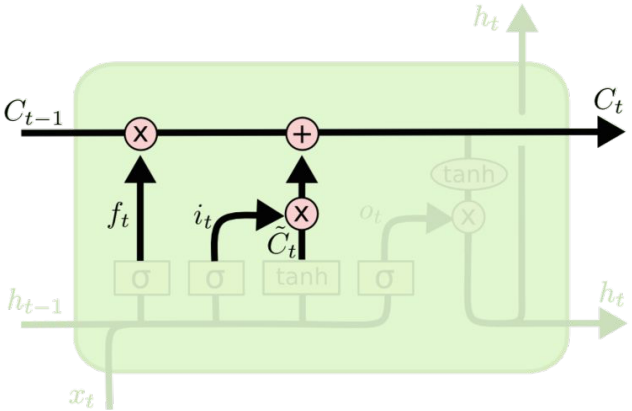



$$f_t = \sigma\left(W_f . [h_{t-1}, x_t] + b_f\right)$$




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

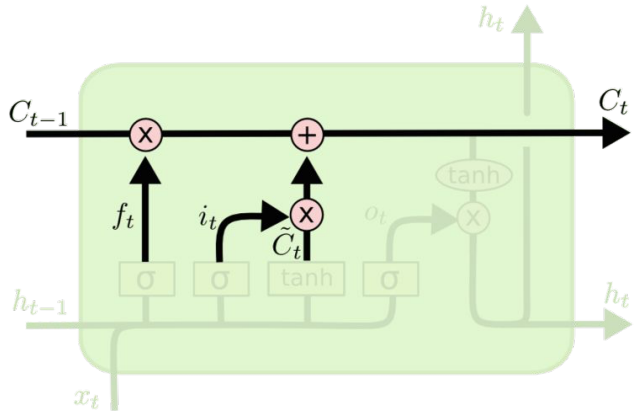
$$\tilde{C}_t = \tanh\left(W_C . [h_{t-1}, x_t] + b_C\right)$$

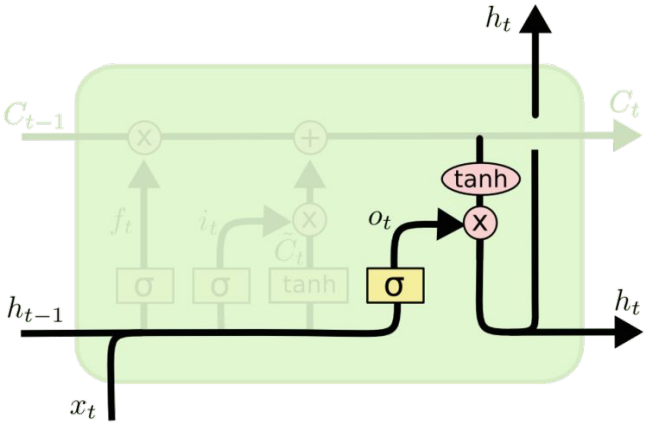



$$C_t = f_t . C_{t-1} + i_t . \tilde{C}_t$$

# LSTM

**Output Gate** (short-term memory) → How much of the updated long-term memory should construct the short-term memory?




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

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

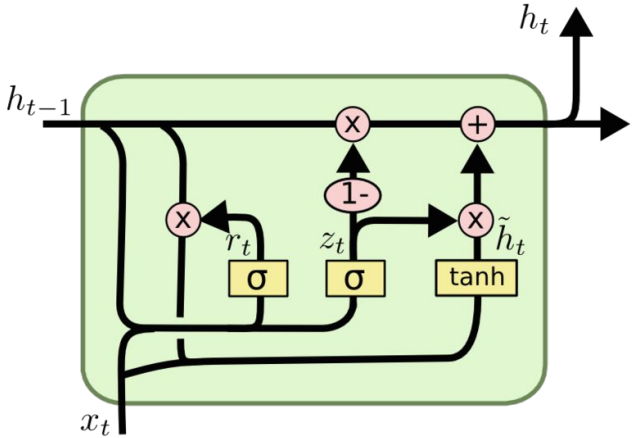
$$h_t = o_t . \tanh(C_t)$$

# Gated Recurrent Unit (GRU)

GRUs are more efficient than LSTMs while having a similar performance

They Combine **forget and input gates** into an **update gate**

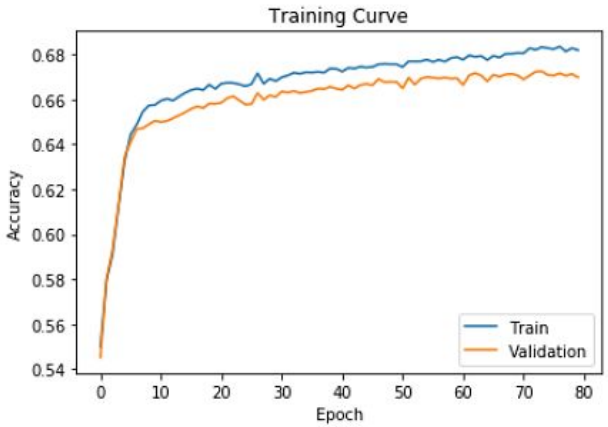
They also Merges cell state and hidden state



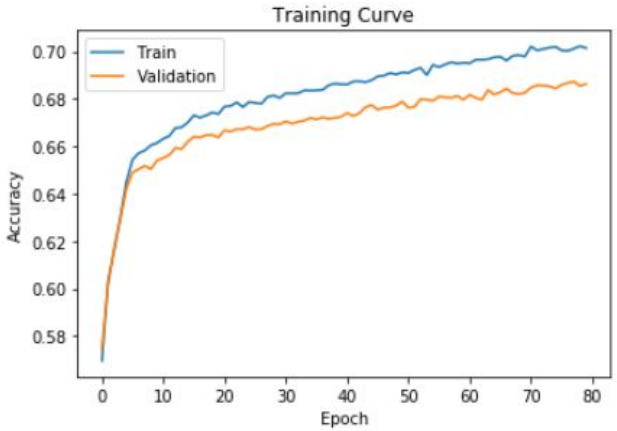

$$z_t = \sigma\left(W_z . [h_{t-1}, x_t] + b_z\right)$$
$$r_t = \sigma\left(W_r . [h_{t-1}, x_t] + b_r\right)$$
$$\tilde{h}_t = \tanh\left(W . [r_t . h_{t-1}, x_t] + b\right)$$
$$h_t = (1 - z_t) . h_{t-1} + z_t \tilde{h}_t$$

# LSTM/GRU vs RNN

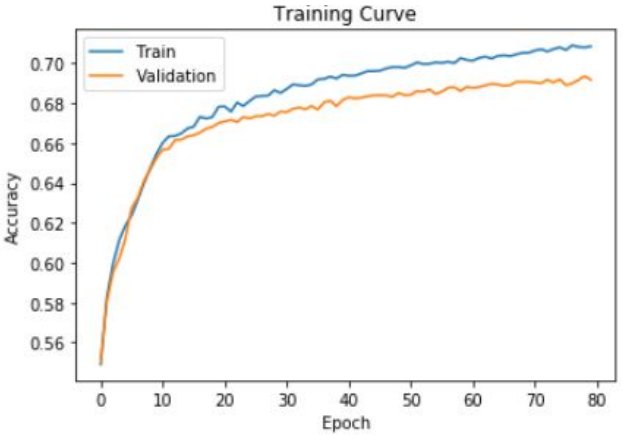
- LSTMs/GRUs can be trained on **longer sequences** and are much better at **learning long-term relationships**
- They are **easier to train** and achieve **better performance** than vanilla RNNs
- LSTM/GRU haven't finished converging here, with more training they do better




RNN




LSTM




GRU

# PyTorch: RNN

```
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, __ = self.rnn(x, h0)
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])
```

# PyTorch: GRU

```
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.GRU(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, __ = self.rnn(x, h0)
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])
```

# PyTorch: LSTM

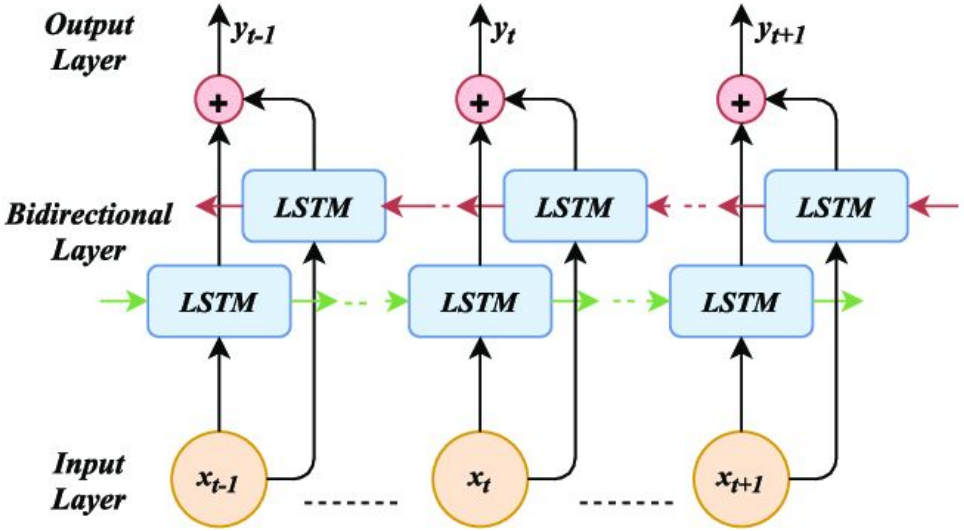
```
class TweetRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_class):
        super(TweetRNN, self).__init__()
        self.emb = nn.Embedding.from_pretrained(glove.vectors)
        self.hidden_size = hidden_size
        self.rnn = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_class)

    def forward(self, x):
        # Look-up the embeddings
        x = self.emb(x)
        # Set the initial hidden states
        h0 = torch.zeros(1, x.size(0), self.hidden_size)
        c0 = torch.zeros(1, x.size(0), self.hidden_size)
        # Forward propagate the RNN
        out, __ = self.rnn(x, (h0, c0))
        # Pass the output of the last step to the classifier
        return self.fc(out[:,-1,:])
```

# Deep & Bidirectional RNNs

# Bidirectional RNNs

A typical state in an RNN (RNN, GRU, or LSTM) relies on the past and the present.

In tasks such as machine translation, where a prediction depends on the past, present, and future, we can exploit the future to improve performance.

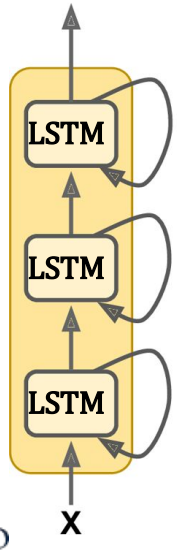



$$h = \left[\overrightarrow{h} \,||\, \overleftarrow{h}\right]$$

# Deep RNNs

We can also stack RNN layers to learn more abstract representations.

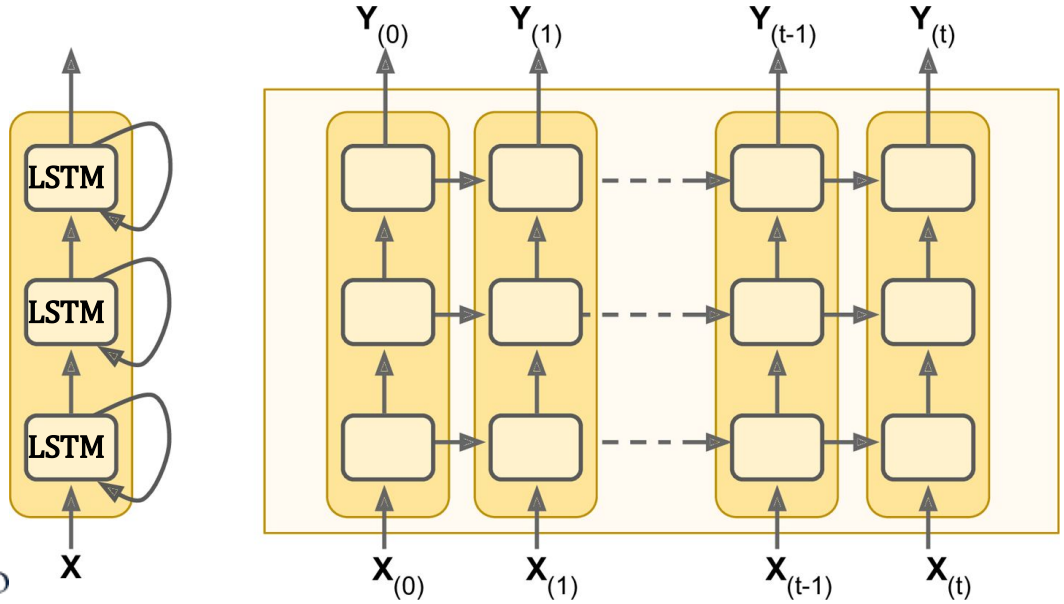
Representations in first layers are better for syntactic tasks while representations in last layers perform better on semantic tasks.



# Deep RNNs

We can also stack RNN layers to learn more abstract representations.

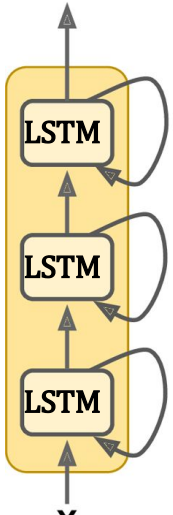
Representations in first layers are better for syntactic tasks while representations in last layers perform better on semantic tasks.

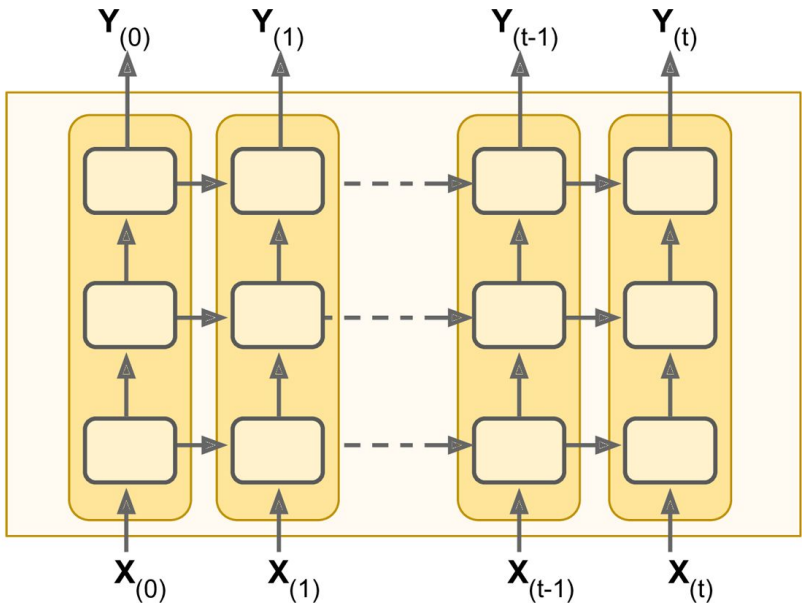


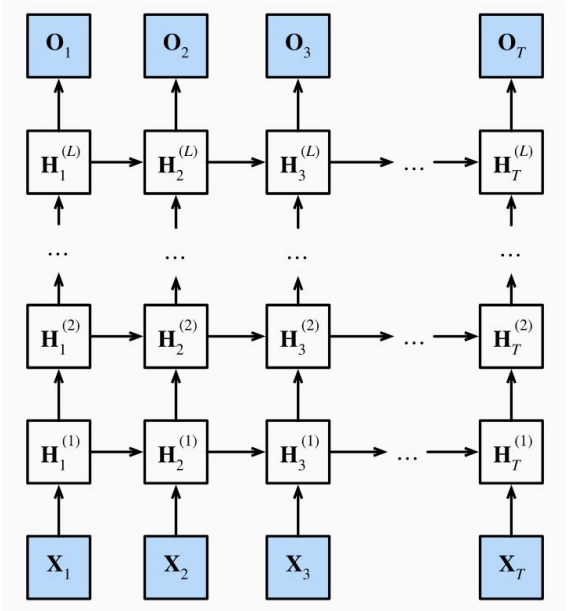
# Deep RNNs

We can also stack RNN layers to learn more abstract representations.

Representations in first layers are better for syntactic tasks while representations in last layers perform better on semantic tasks.



# PyTorch Details

$N$ = batch size
$T$ = sequence length
$D$ = 2 if bidirectional=True otherwise 1
$H_{in}$ = input_size
$H_{out}$ = hidden_size

Output

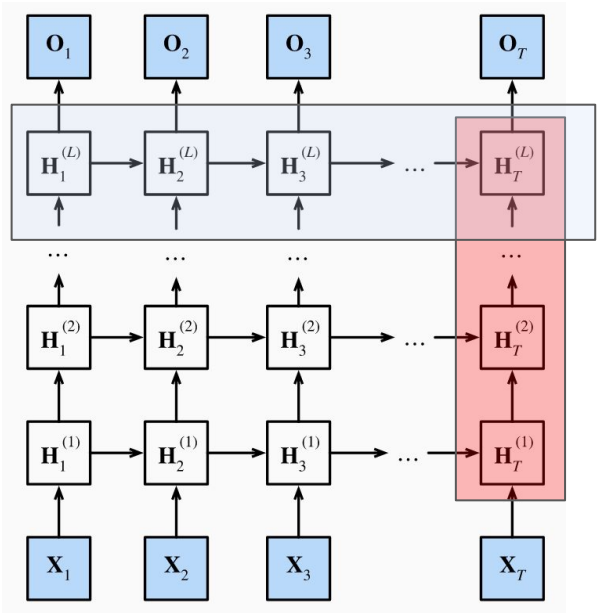
h_n

```
self.rnn = nn.GRU(input_size=64,
                  hidden_size=256,
                  batch_first=True,
                  num_layers=4,
                  bidirectional=True)

h0 = torch.zeros(2x4, x.size(0), self.hidden_size)

output, h_n = self.rnn(x, h0)
```

Dxnum_layer

tensor of shape (N, T, DxH$_{out}$) containing the output features ($h_t$) from the last layer of the GRU, for each t

tensor of shape (N, Dxnum_layers, H$_{out}$) containing the final hidden state for the input sequence.



# Sequence-to-Sequence Models

# Sequence-to-Sequence RNNs

Learning to generate new sequences requires addressing some problems:

- How do we generate variable-length sequences → how do we know when to **stop/finish a generated sequence** !
- Training-time behaviour must be changed (**teacher-forcing** )
- Inference-time behaviour also changes (**sampling and temperature scaling** )

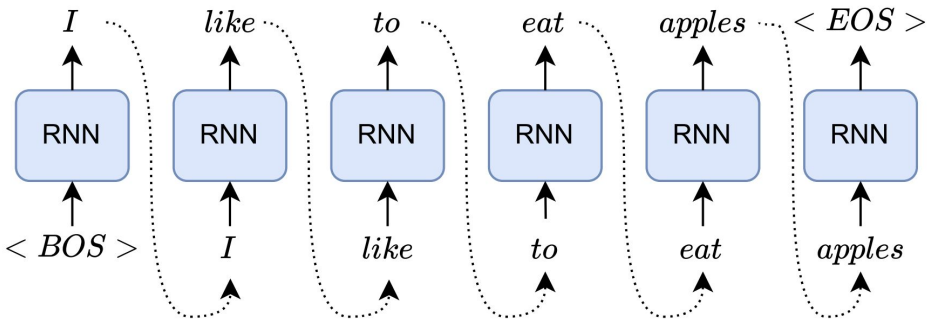
# During Training

How do we know when to **stop/finish a generated sequence** ?

Let's use dedicated control symbols to define the Beginning of Sequence **<BOS>** and End of Sequence **<EOS>**

<BOS> I like to eat apples <EOS>

Once the RNN generates <EOS> we will know it is done generating!

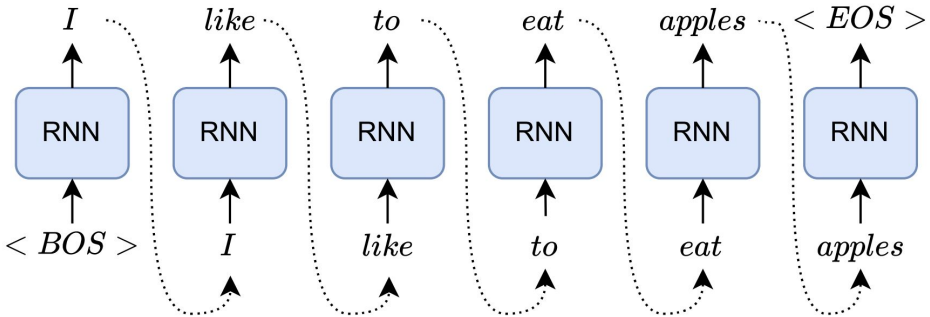


# During Training

How to define the **ground-truth and loss** ?

RNN is trained to generates one particular sequence in the training set:

- We feed the RNN with **<BOS>** and compare its prediction with **I** (**Cross-Entropy** )
- We then feed it with **I** and compare the prediction with **like**
- …
- Finally, we feed it with **apples** and compare the prediction with **<EOS>**

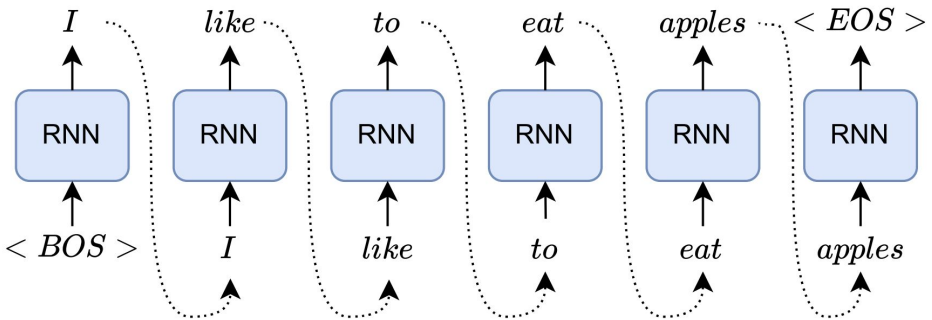


# Teacher Forcing

In each step, we compute the loss by comparing ground-truth and predicted tokens.

In order to make training more **efficient** , we force the RNN to stay close to the ground-truth sequence.

We do this by passing the **ground-truth label as the next input** instead of current prediction → This trick is known as **Teacher Forcing**



# During Inference

Unlike in a classification problem, always selecting the token with the **highest probability** won't work well:

- When using a generative model, we want **diversity** not deterministic behavior.
- In practice, this greedy approach results in lots of **grammatical errors** .

To address this, we **sample from the predicted distributions** . We will address 3 sampling strategies:

- Greedy search
- Beam search
- Softmax Temperature Scaling

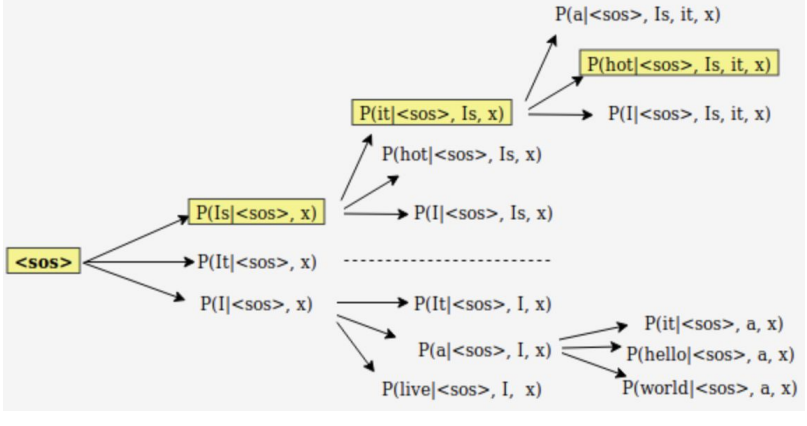
# During Inference

**Greedy Search** selects the token with highest probability as the generated token

$$\max p(t_1, t_2, \ldots t_n) = \max p(t_1) \times p(t_2) \times \cdots \times p(t_n)$$

**Beam Search** looks for a sequence of tokens with the highest probability within a window.

$$\max p(t_1, t_2, \cdots t_n) = \max p(t_1) \times p(t_2|t_1) \times \cdots p(t_n|t_{n-1}, \cdots, t_2, t_1)$$



# During Inference

**Softmax Temperature Scaling** helps with the problem of over-confidence in neural networks by by scaling the input logits to the softmax with a temperature.

**Low Temperature** (larger logits, more confident)

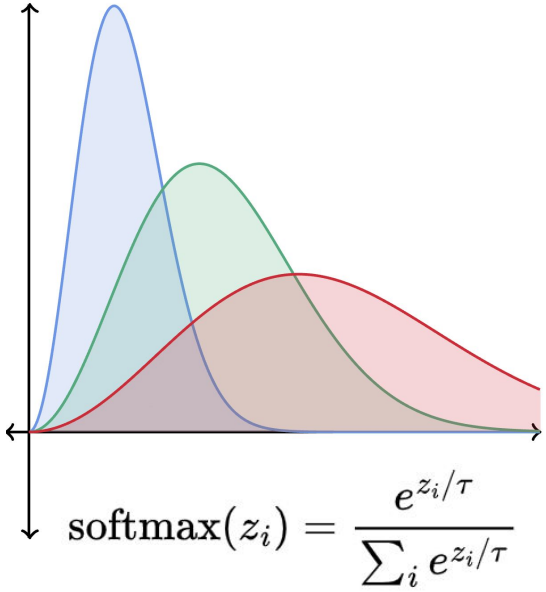
- Higher quality samples, less variety

**High Temperature** (smaller logits, less confident)

- Lower quality samples, more variety

```
print(sample_sequence(model, temperature=0.8))
print(sample_sequence(model, temperature=1.0))
print(sample_sequence(model, temperature=1.5))
print(sample_sequence(model, temperature=2.0))
print(sample_sequence(model, temperature=5.0))
```

```
God Bless the people of Venezuela!
God Bless the people of Venezuela!
os Bless the peopleeof Venezuela!
God Bdess Bpeooeop e ofzuenezeela!
auodlflBdptfh oe!on!!lhlpfeVeGhfpup!f
```



# Example: Generate Presidential Tweets

Dataset: ~20,000 Trump Tweets from 2018

At most 140 characters!

Remove tweets that starts with “http” (tweet with link only)

# PyTorch: Text Generator

```
class TextGenerator(nn.Module):
    def __init__(self, vocab_size, hidden_size, n_layers=1):
        super(TextGenerator, self).__init__()
        # Identity matrix for generating 1-hot vectors
        self.ident = torch.eye(vocab_size)
        # Recurrent neural network
        self.rnn = nn.GRU(vocab_size, hidden_size, batch_first=True)
        # A FC layer outputting a distribution over the next token
        self.decoder = nn.Linear(hidden_size, vocab_size)

    def forward(self, inp, hidden=None):
        # Generate 1-hot vectors of input
        inp = self.ident[inp]
        # Get the next output and hidden state
        output, hidden = self.rnn(inp, hidden)
        # Predict distribution over next tokens
        output = self.decoder(output)
        return output, hidden
```

# PyTorch: Training Text Generator

```
def train(model, data, batch_size=1, num_epochs=1, lr=0.01):
    optimizer = torch.optim.Adam(model.parameters(), lr=lr)
    criterion = nn.CrossEntropyLoss()
    data_iter = torchtext.legacy.data.BucketIterator(data,
                                                     batch_size=batch_size,
                                                      sort_key= lambda x: len(x.text),
                                                      sort_within_batch=True)
    for __ in range(num_epochs):
        Avg_loss = 0
        for (tweet, lengths), label in data_iter:
            target = tweet[:, 1:]
            inp = tweet[:, :-1]
            optimizer.zero_grad()
            output, __ = model(inp)
            loss = criterion(output.reshape(-1, vocab_size), target.reshape(-1))
            loss.backward()
            optimizer.step()
```

# PyTorch: Sampling Text Generator

```
def sample(sample, max_len=100, temperature=0.8):
    generated_sequence = ''
    inp = torch.Tensor([vocab_stoi['<BOS>']]).long()
    hidden = None
    for p in range(max_len):
        output, hidden = model(inp.unsqueeze(0), hidden)

        # Sample from the model as a multinomial distribution
        output_dist = output.data.view(-1).div(temperature).exp()
        top_i = int(torch.multinomial(output_dist, 1)[0])

        # Add predicted character to string and use as next input
        predicted_char = vocab_itos[top_i]
        if predicted_char == '<EOS>':
            break
        generated_sequence += predicted_char
        inp = torch.Tensor([top_i]).long()
    return generated_sequence
```

# Questions?