CSE440: Natural Language Processing II

Farig Sadeque
Assistant Professor
Department of Computer Science and Engineering
BRAC University

Lecture 6: Neural Nets and RNN

Outline

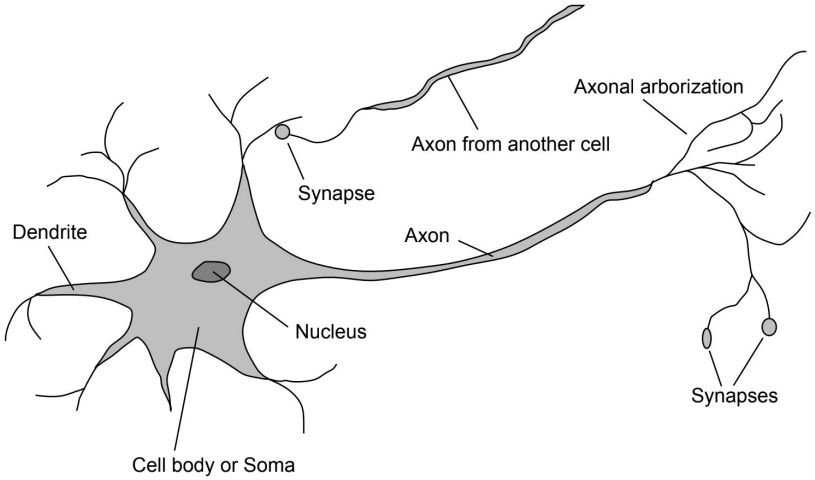
- Neural Networks (SLP 7 and lecture)
- Recurrent neural networks (SLP 9 and lecture)

Before starting learning sequence ….

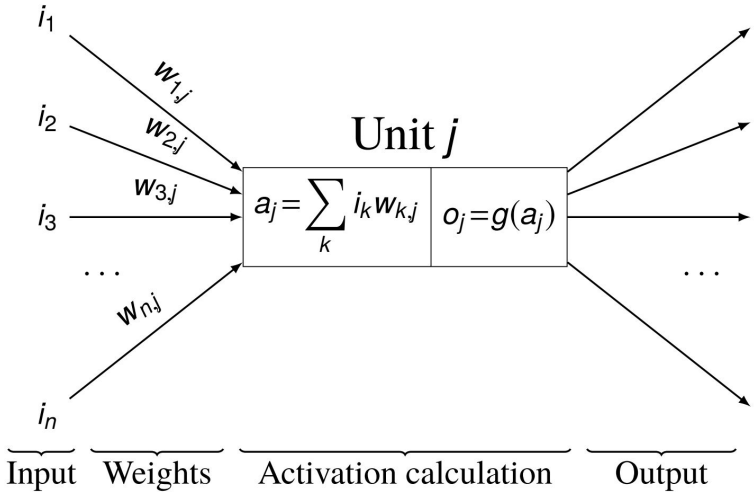
We need to remember some neural network basics.

Neural Networks

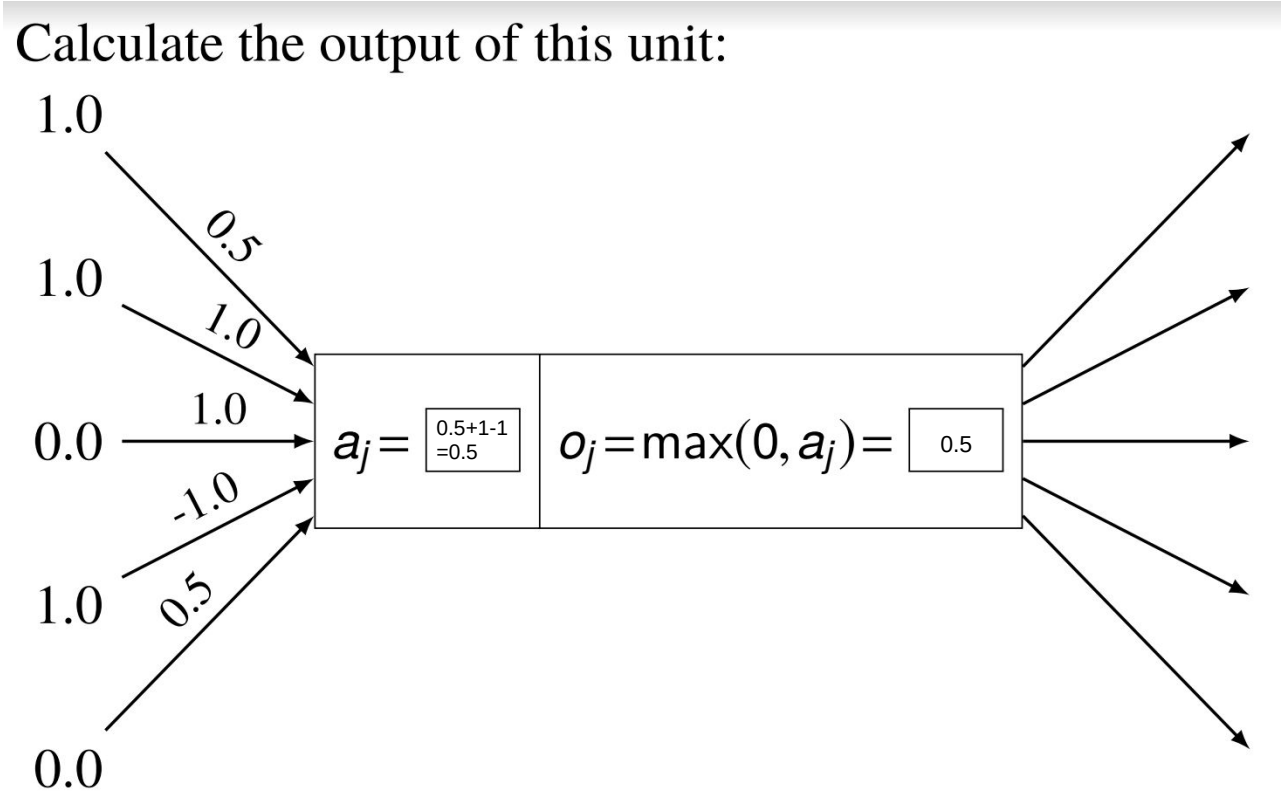
Neuron in a human brain



Neuron in an ML model

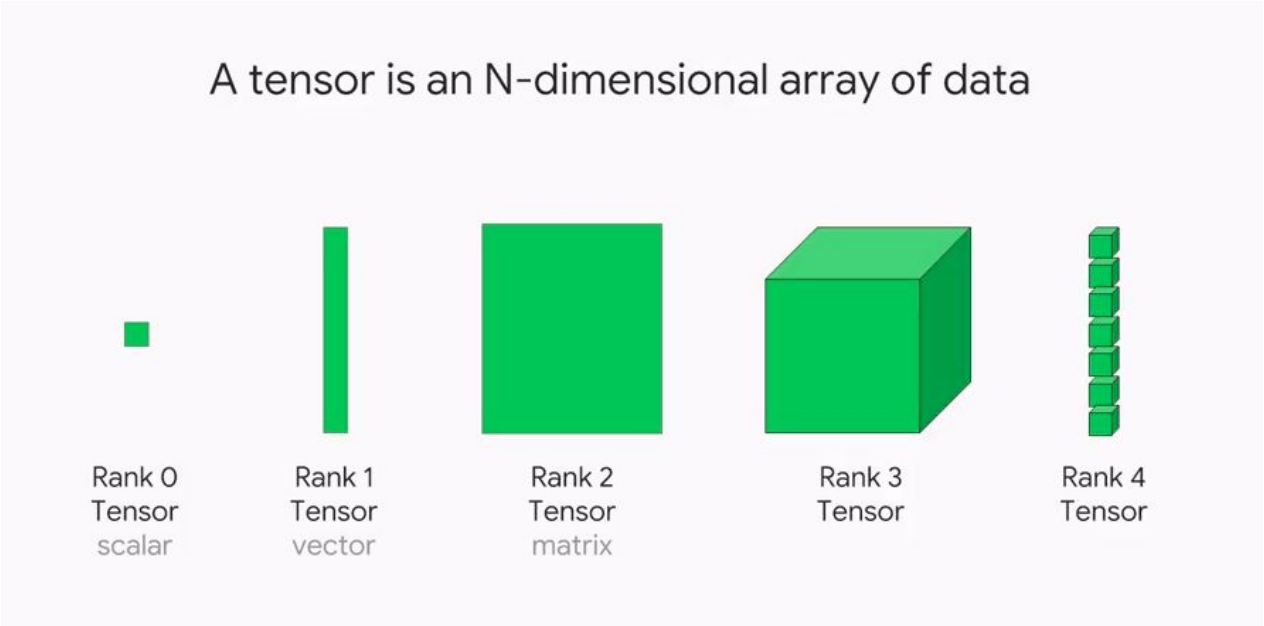


Class work


Calculate the output of this unit:
1.0
1.0
0.0
1.0
0.0
0.5
1.0
1.0
-1.0
0.5
$a_j =$
0.5+1-1 =0.5
$o_j = \max(0, a_j) =$
0.5

Unit calculations as tensor arithmetic

What is a tensor?



Unit calculations as tensor arithmetic

Summation for a single unit:

$$o_j = g\left(\sum_k i_k w_{k,j}\right)$$

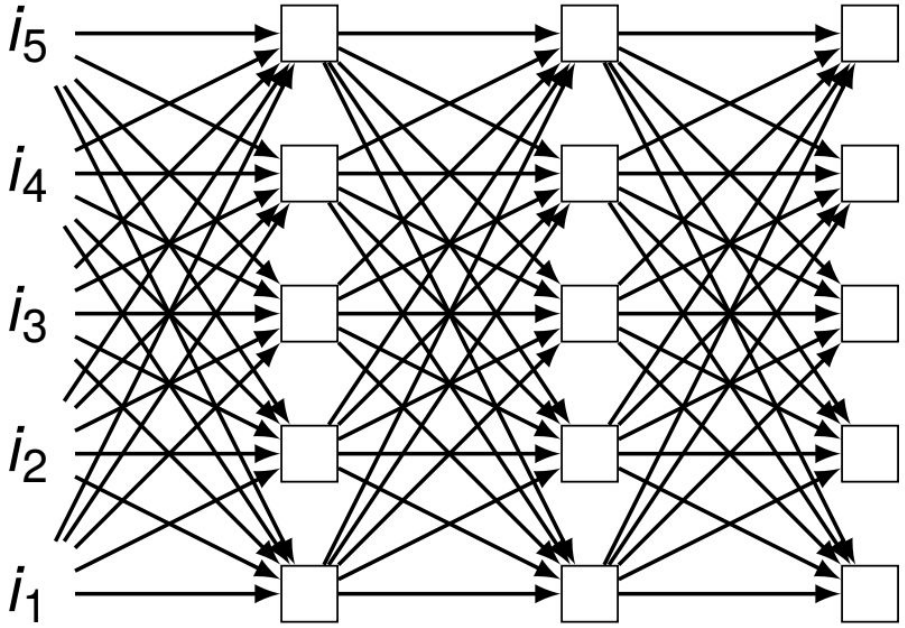
Vector arithmetic for a single unit:

$$o_j = g\left(\begin{bmatrix} i_1 & i_2 & \dots & i_n \end{bmatrix} \begin{bmatrix} w_{1,j} \\ w_{2,j} \\ \dots \\ w_{n,j} \end{bmatrix}\right) = g\left(\boldsymbol{i} \times \boldsymbol{w}_{*,j}\right)$$

Matrix arithmetic for multiple units:

$$\boldsymbol{o} = g\left(\boldsymbol{i} \times \boldsymbol{W}\right)$$

A feedforward network as composition



$$\boldsymbol{o} = g\left(g\left(g\left(\boldsymbol{i} \times \boldsymbol{W}^1\right) \times \boldsymbol{W}^2\right) \times \boldsymbol{W}^3\right)$$

Activation functions

- Why do we need activation functions?
- What types of activation functions do we have?

Learning the XOR

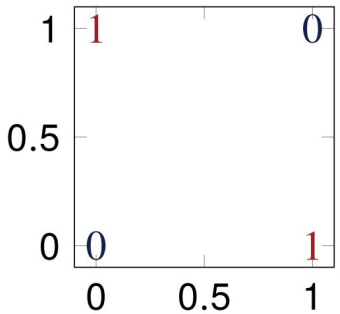
$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \qquad \boldsymbol{y} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Can you solve it with linear regression? No

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{b}$$

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \boldsymbol{W} + \boldsymbol{b}$$

No such weights exist!



$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} ? \\ ? \end{bmatrix} + ?$$

Solving XOR with NNs

$$f(\boldsymbol{X}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \max(0, \boldsymbol{X}\boldsymbol{W} + \boldsymbol{c})\,\boldsymbol{w} + b$$

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \max\left(0, \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \boldsymbol{W} + \boldsymbol{c}\right) \boldsymbol{w} + b$$

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \max\left(0, \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} ? & ? \\ ? & ? \end{bmatrix} + \begin{bmatrix} ? & ? \end{bmatrix}\right) \begin{bmatrix} ? \\ ? \end{bmatrix} + ?$$

Solving XOR with NNs

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \max\left(0, \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 0 & -1 \end{bmatrix}\right) \begin{bmatrix} 1 \\ -2 \end{bmatrix} + 0$$

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ -2 \end{bmatrix} + 0$$

Common activation functions in hidden units

We have: affine transformation of input x, followed by nonlinear activation function

$$\boldsymbol{h} = g(\boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$$

g could be just about anything! It can be linear, but a linear function is not preferred. Why? Cause linear function can't solve multiclass or non-linear problems like XOR problem.

Considerations:

- What specific behavior is needed? Non-linear
- How will the gradients behave? Linear functions have constant derivatives that never change during backpropagation
  Non linear functions solves vanishing or exploding gradient problem

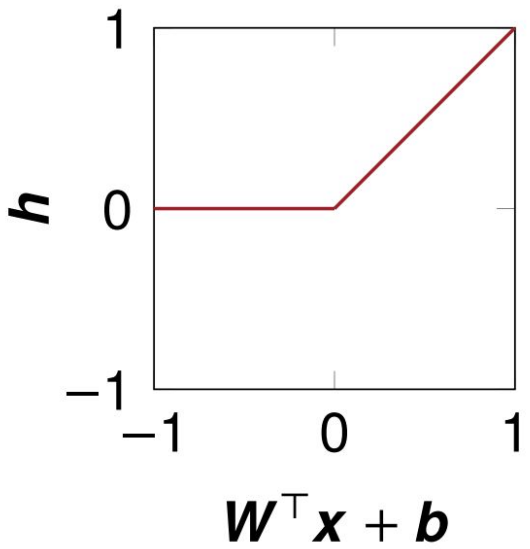
Why linear functions are not preferred?

Because of the considerations.

- We need complex mappings between the inputs and the outputs
- All linear layers will translate the input linearly to output– that is, multiple linear transformation is basically one giant linear transformation
- Cannot use backpropagation as the derivative is constant

ReLU

$$\boldsymbol{h} = \max(0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$$



Behavior?

- Active only when input is positive

Gradients?

- 1 when positive
- 0 when negative

ReLU is non-differentiable

ReLU at z = 0:

- left derivative = 0
- right derivative = 1

So ReLU is not differentiable at z = 0!

A few non-differentiable points are not a problem:

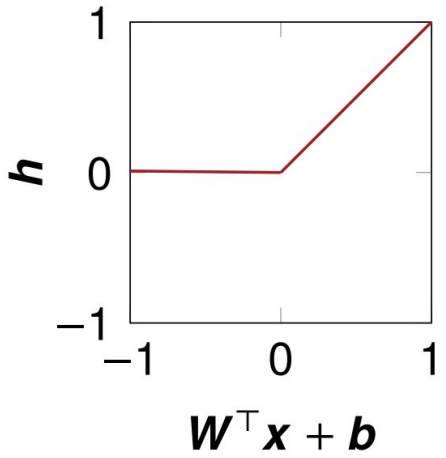
- Training rarely reaches a point with gradient 0 anyway
- Software simply returns either left or right derivative

GeLU better

Leaky ReLU

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

$$\boldsymbol{h} = \max(0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b}) + 0.01 \min(0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{b})$$



ReLU has a “dead neuron” problem!

Behavior?

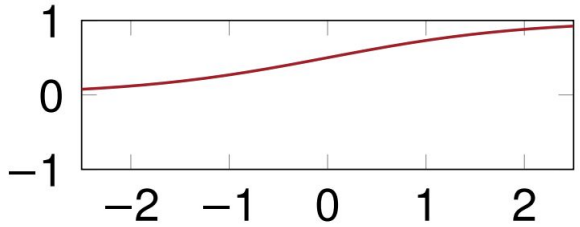
- Strong positive activation when positive
- Very weak negative activation when negative

Gradients?

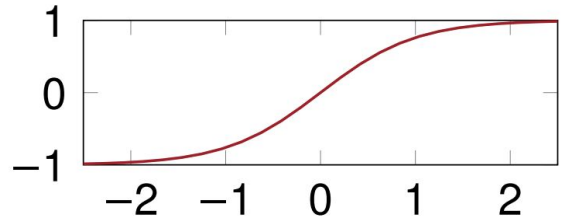
- 1 when positive
- 0.1 when negative

Sigmoid and tanh

Sigmoid: $f(x) = \frac{1}{1+e^{-x}}$



Tanh: $f(x) = \frac{\left(e^{x}-e^{-x}\right)}{\left(e^{x}+e^{-x}\right)}$



Behavior?

- Sigmoid: 0/1 switch
- Tanh: -1/+1 switch

Gradients?

- Saturate across most of their domain
- Tanh optimizes slightly better since it is similar to the identity function near 0

Hyperparameters

- Network parameters are the ones that are being learned throughout the training process (e.g. weights)
- There are parameters that we can control to facilitate this learning: they are called hyperparameters
- Activation function is one such hyperparameter
- We have others …

Optimizers

- Algorithms used to update the learnable parameters in order to reduce loss
- Common optimizers:
  - Gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent
  - Momentum
  - Adagrad
  - RMSProp
  - AdaDelta
  - Adaptive moment estimation
- GD methods maintain a single learning rate (with/without decay for all parameters and are known as first order optimizers (works with the first order derivative)
- Adaptive methods like Adagrad provide learning rates for each parameter, thus improving the learnability, but are computationally expensive
- RMSProp not only provides LR for each parameter, it adapts based on the mean of recent magnitudes of the gradients for the weight → first order. AdaDelta is similar but works with squared gradients (second order)

Adaptive moment estimation

- Popularly known as Adam (not ADAM)
- Came out of OpenAI and University of Toronto
- Most likely the highest cited paper in recent history
- Why is it so popular?
  - Straightforward to implement
  - Little memory requirements
  - Well suited for problems that are large in terms of data and/or parameters
  - Needs little to no manual tuning

Adam

- Adam works with momentums of first and **second** order
- Instead of adapting the parameter learning rates based on the average first moment (the mean), m_t as in RMSProp, Adam also makes use of the average of the second moments of the gradients (the uncentered variance) v_t

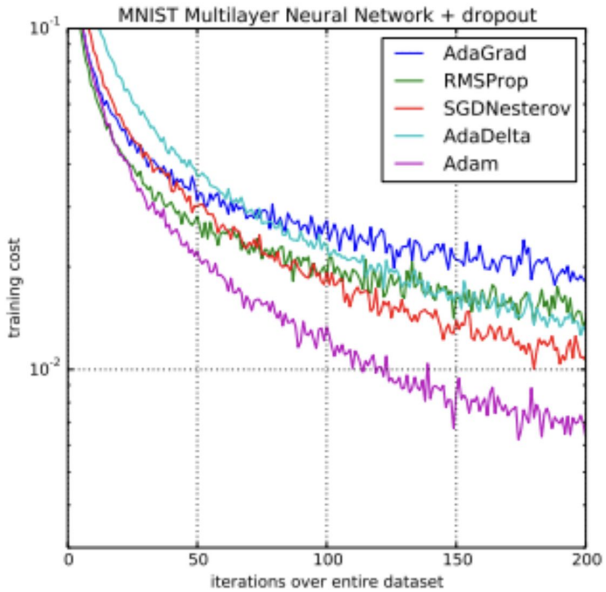
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

- The values for β1 is 0.9 , 0.999 for β2 and epsilon is an extremely small number to avoid zero division

Adam's popularity



Sebastian Ruder: "... RMSprop, Adadelta, and Adam are very similar algorithms that do well in similar circumstances. ….. Adam might be the best overall choice."
Andrej Karpathy: "In practice Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp."

Regularization

- A set of strategies used in Machine Learning to reduce the generalization error
- Why?
- Bias-variance tradeoff
- Bias: error from wrong assumptions in the learning algorithm
  - High bias can cause an algorithm to miss the relevant relations between features and target outputs → Observations don't matter
  - Underfitting
- Variance: error from sensitivity to small fluctuations in the training set
  - High variance may result in modeling the random noise in the training data → focusing too much on observations
  - Overfitting

Regularization

- A good model needs to balance bias and variance
- Regularization helps us do that

Regularization techniques

- Introduce regularization term to the loss function
  - Introduces a small amount of bias to counter variance → reduce overfitting
  - Loss function: negative log likelihood or binary cross entropy
  - Most common terms:
    - L2 regularizer: Ridge regression → adds the "squared magnitude" of the coefficient as the penalty term to the loss function
    - L1 regularizer: Lasso regression → adds the "absolute value of magnitude" of the coefficient as a penalty term to the loss function
    - Uses λ that controls the sensitivity of the model to the input → less sensitive, less likely to overfit
    - L2 focuses on larger weights, so higher λ means L2 penalizes higher value weights more than lower ones → no feature essentially goes away
    - L1 has equal focus, so, higher λ → low weight features go away → feature selection

Regularization techniques

- Dropout: Drops out (ignore) a layer's output with a probability *p*
  - Choice of p depends on the architecture
- Early stopping: Stops when performance gets saturated
  - Stops model from being overfit
  - Returns the best current model
- Data augmentation
  - Introduce new training data with variation
  - Injects noise
  - Add bias

Other common hyperparameters

- Learning rate: how quickly a network updates its parameters
  - Low learning rate slows down the learning process but converges smoothly
  - Larger learning rate speeds up the learning but may not converge
  - Decaying learning rate is preferred
- Epochs: How many times the entire training dataset has passed through the model
- Batch size: (Mini) batch size refers to a subset of the training data. Weights are updated after each mini batch training
  - Small mini batch → too many updates
  - Large mini batch → too few updates, too many epochs to converge

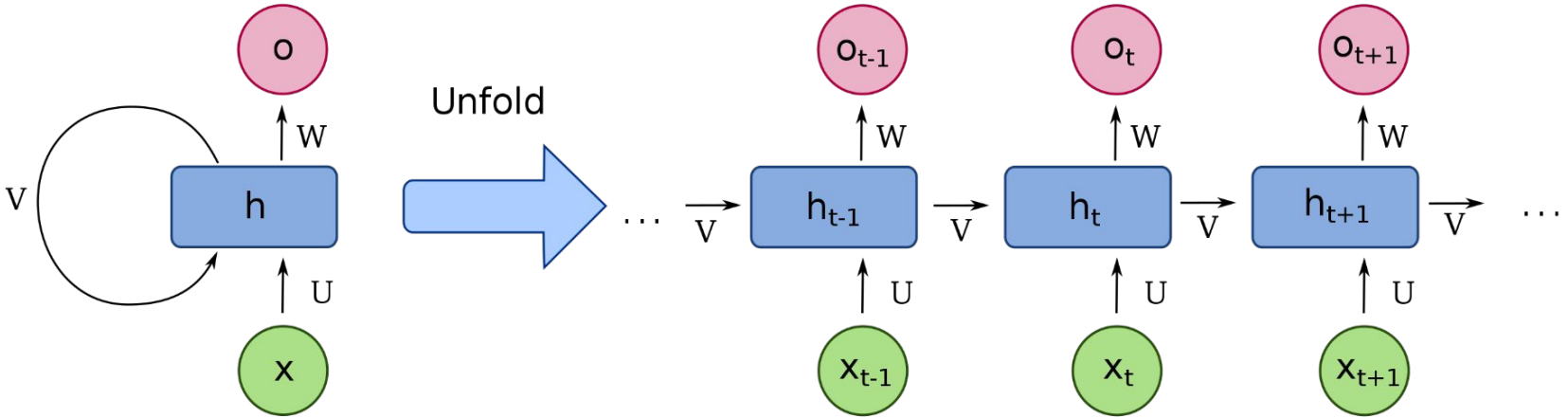
Recurrent Neural Networks (RNN)

- Simple recurrent networks
- Bidirectional and gated recurrent networks
- Recurrent architectures
- Seq2Seq models (next session)
- Attention (next session)

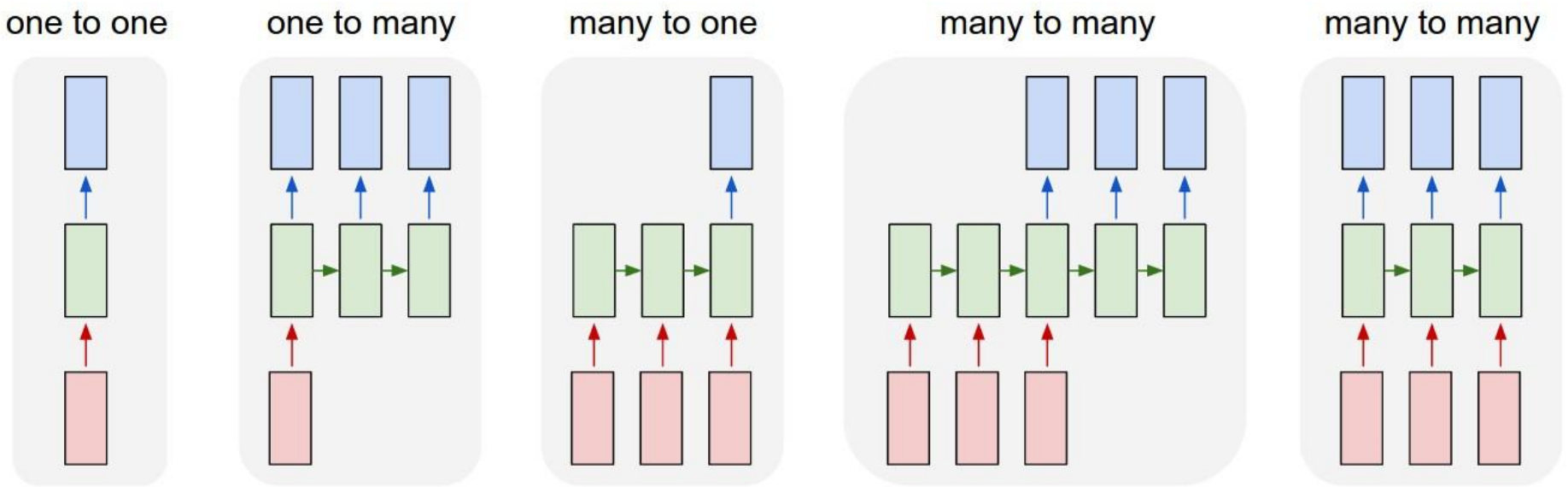
Short history of RNN

- 1986: RNNs are Introduced by David Rumelhart
- 1995: LSTMs are introduced by Sepp Hochreiter and Jürgen Schmidhuber based on Hochreiter's 1991 research on vanishing gradient problem
- 2001: Gers and Schmidhuber trained LSTMs to learn language models (unlearnable by HMMs)
- 2009: Graves et al. won ICDAR handwriting recognition competition using LSTMs
- 2013: Hinton and his team destroyed previous record for speech recognition using LSTM
- 2014: GRU is introduced by Cho et al.
- 2015: Widespread use in both academia and industry due to Google's adaptation of LSTM in their Google Voice speech recognition system

Structure of an RNN

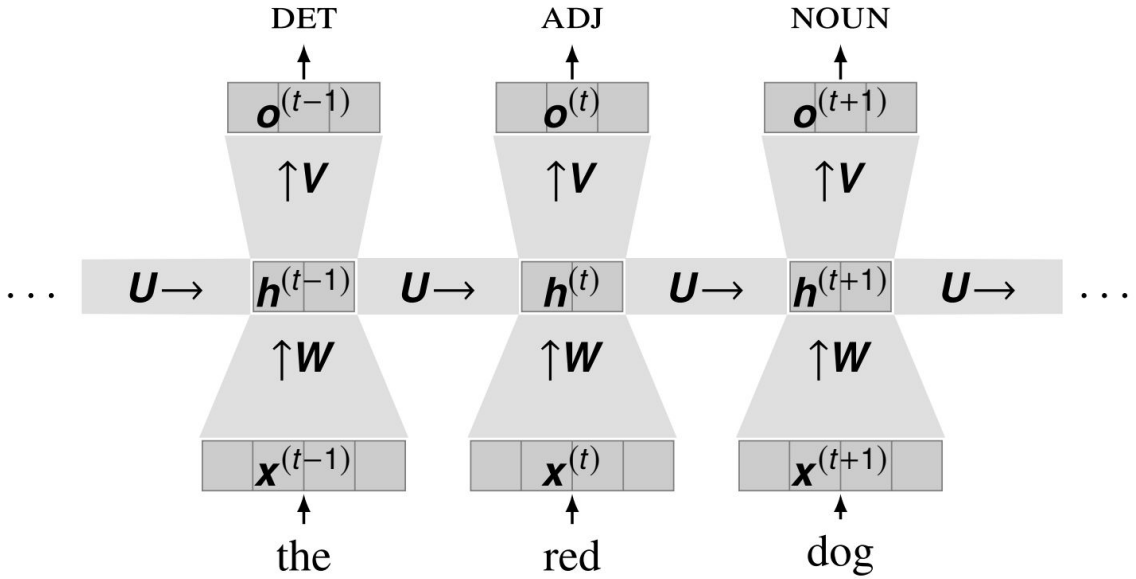


Types of RNNs



https://karpathy.github.io/2015/05/21/rnn-effectiveness/

A completely unrolled Simple RNN



$$o^{(t)} = \text{softmax}(Vh^{(t)})$$

$$h^{(t)} = \tanh(Uh^{(t-1)} + Wx^{(t)})$$

Simple RNN

Intuitions:

- Each step combines the current input with the history
- Each prediction is made based on this combination

Observations:

- The input and hidden state change at each time step
- The parameters W, U, V are the same at each step

Equations:

$$\boldsymbol{h}^{(t)} = \tanh(\boldsymbol{U}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}\boldsymbol{x}^{(t)})$$

$$\boldsymbol{o}^{(t)} = \mathrm{softmax}(\boldsymbol{V}\boldsymbol{h}^{(t)})$$

Classwork

Consider an RNN that predicts as:

$$\boldsymbol{o}^{(t)} = \text{softmax}(\boldsymbol{V}\boldsymbol{h}^{(t)})$$

$$\boldsymbol{h}^{(t)} = \boldsymbol{U}\boldsymbol{h}^{(t-1)} + \boldsymbol{W}\boldsymbol{x}^{(t)}$$

whose parameters have been set to:

$$\boldsymbol{U} = \begin{bmatrix} 2 & 0 \\ 1 & 2 \end{bmatrix} \quad \boldsymbol{W} = \begin{bmatrix} 2 & 0 & -1 \\ 1 & -1 & 1 \end{bmatrix} \quad \boldsymbol{V} = \begin{bmatrix} 1 & 0 \\ -1 & 1 \\ 0 & 3 \end{bmatrix}$$

If you are given the following input:

$$h^{(0)} = \begin{bmatrix} 1 \\ -1 \end{bmatrix} \quad x^{(1)} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad x^{(2)} = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Which label will be predicted for each word if in the final softmax, index 0=ADJ, index 1=DET, and index 2=NOUN?

Drawbacks of simple RNN

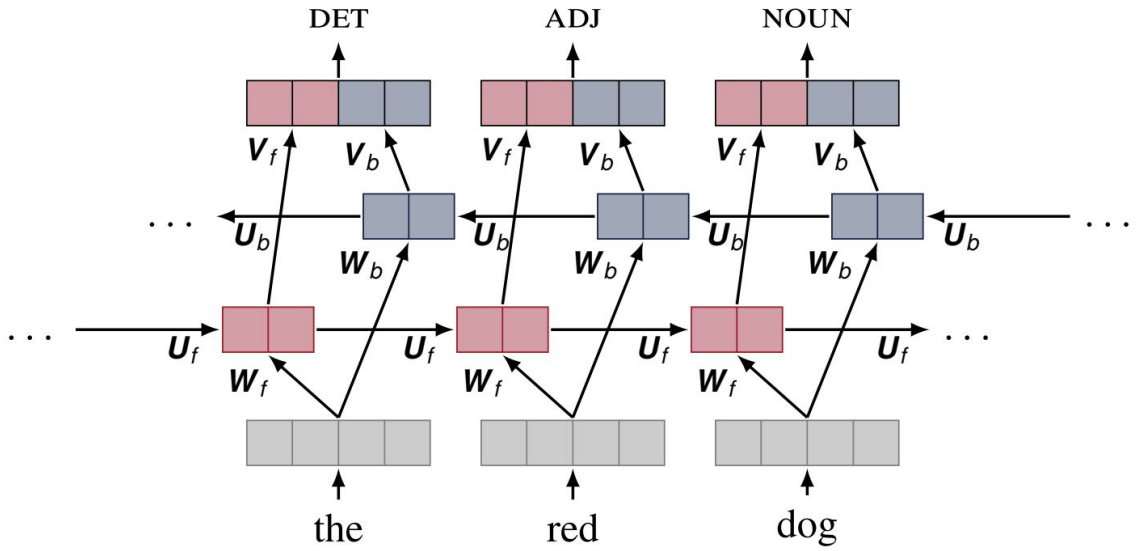
- They can only see the past, not the future
- They must forget the same amount of history at each time step
  - Theoretically, they don't have to
  - Maintaining long term memory is difficult

Bidirectional RNNs

Intuition:

- Run one forward RNN
- Run one backward RNN
- Combine their outputs

Bidirectional RNNs



$$\boldsymbol{o}^{(t)} = \text{softmax}(\boldsymbol{V}_f \boldsymbol{h}_f^{(t)} + \boldsymbol{V}_b \boldsymbol{h}_b^{(t)})$$

$$\boldsymbol{h}_b^{(t)} = \tanh(\boldsymbol{U}_b \boldsymbol{h}^{(t+1)} + \boldsymbol{W}_b \boldsymbol{x}^{(t)})$$

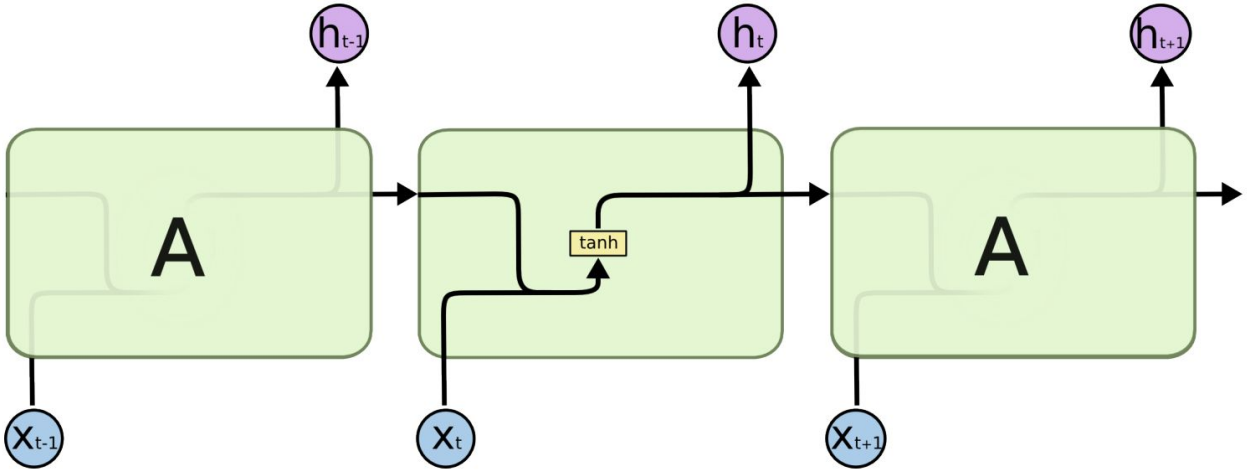
$$\boldsymbol{h}_f^{(t)} = \tanh(\boldsymbol{U}_f \boldsymbol{h}^{(t-1)} + \boldsymbol{W}_f \boldsymbol{x}^{(t)})$$

Gated RNNs

Intuition:

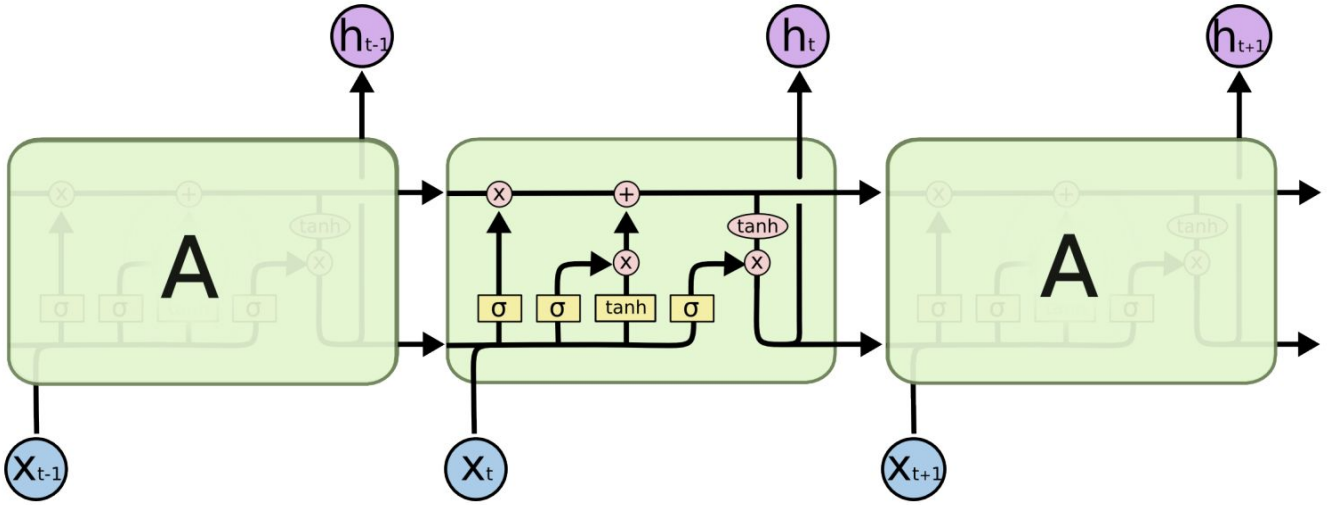
- Simple RNNs forget the same amount at each time step
- Look at the previous hidden state and the current input
- Decide how much to forget based on those
- Two gated RNNs
  - Long Short-Term Memory (LSTM)
  - Gated Recurrent Units (GRU)

Simple RNN

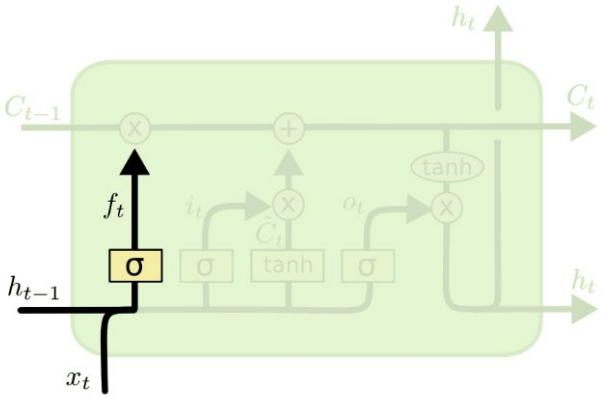


→ Not support Parallelism

LSTM



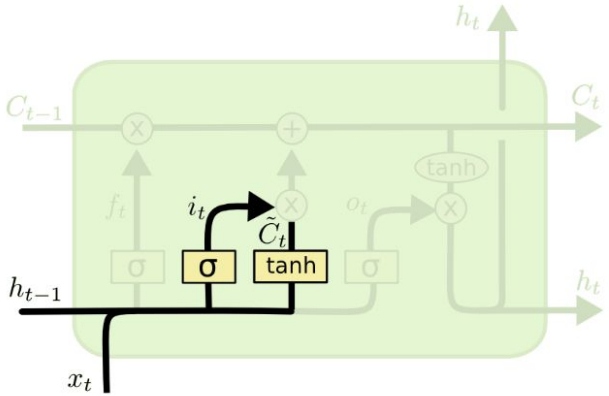
Forget gate



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

→ Exploding gradient is impossible in LSTM

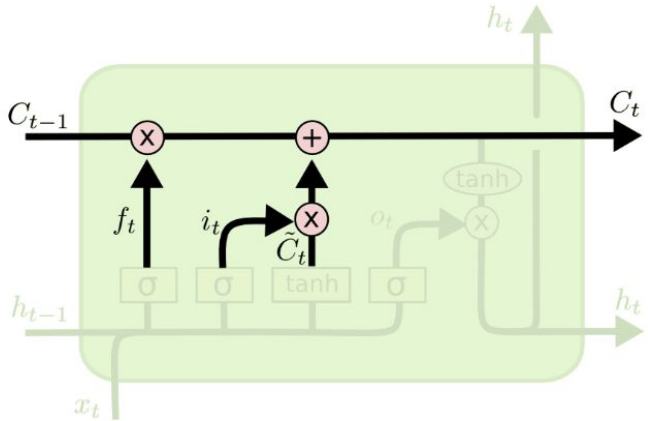
Input gate



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

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

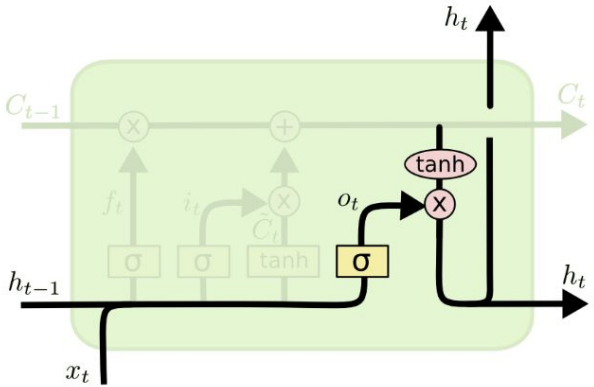
Cell update



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

↳ Cell state is persistant memory

Output gate



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

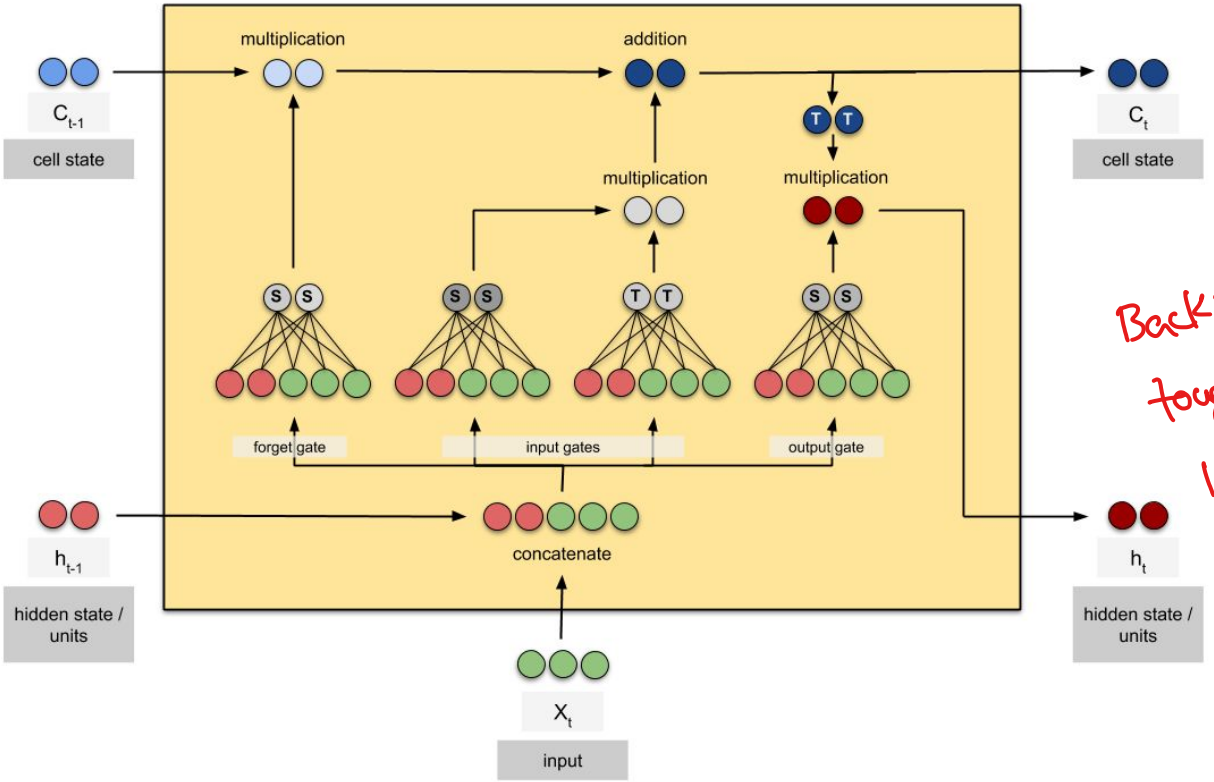
$$h_t = o_t * \tanh\left(C_t\right)$$

→ Have memory redundancy problem

Useful blog from Colah

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

One LSTM cell

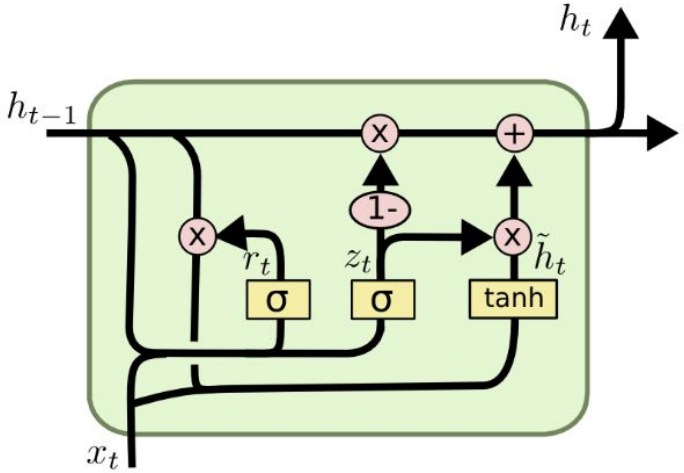


Backpropagation is tough because of losing weight in multi-layer case

LSTM

- Step 1: Calculate forget (f_t), input (i_t) and output gates (o_t)
- Step 2: Calculate cell state update
- Step 3: Update cell state (C_t)
- Step 4: Calculate h_t

GRU



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

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

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

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

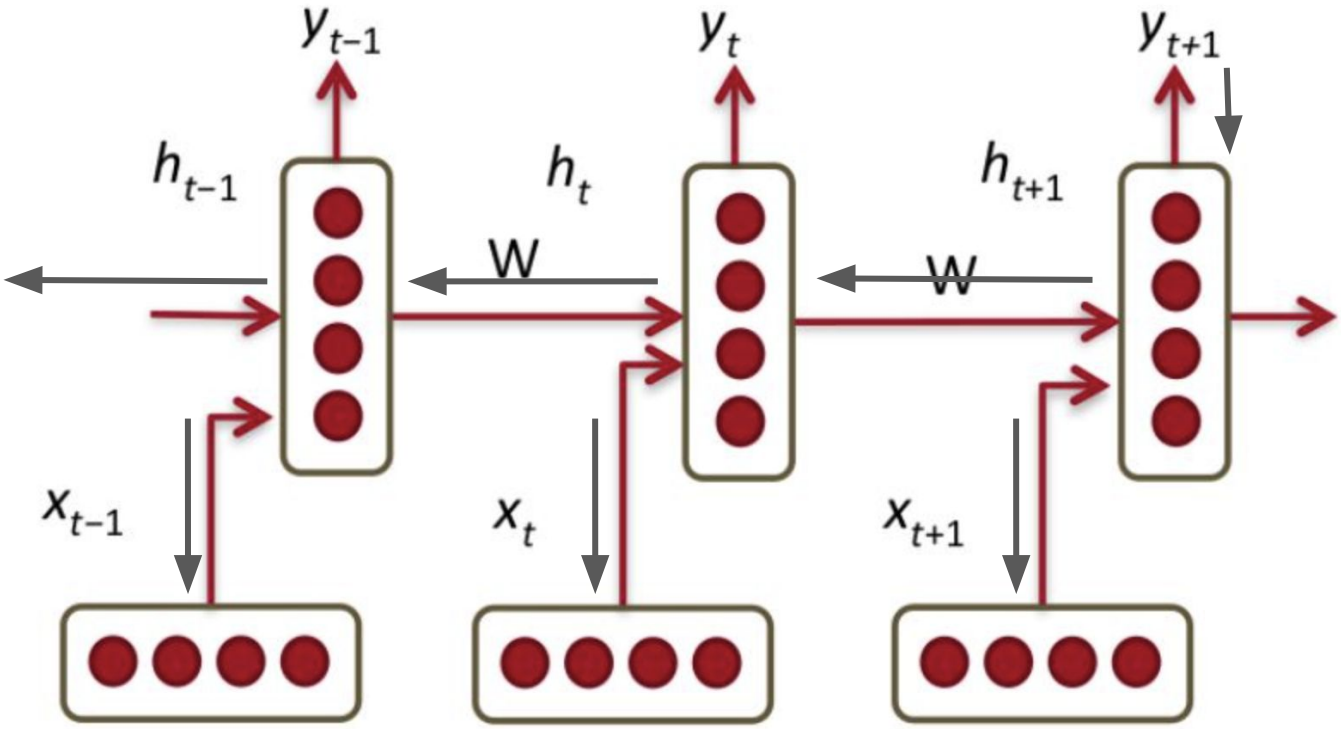
Gated RNNs

Properties:

- Can forget different amounts at each time step
- Much better at using long distance information

A bidirectional GRU is a good starting point for many sequence tagging tasks

Backpropagation through time



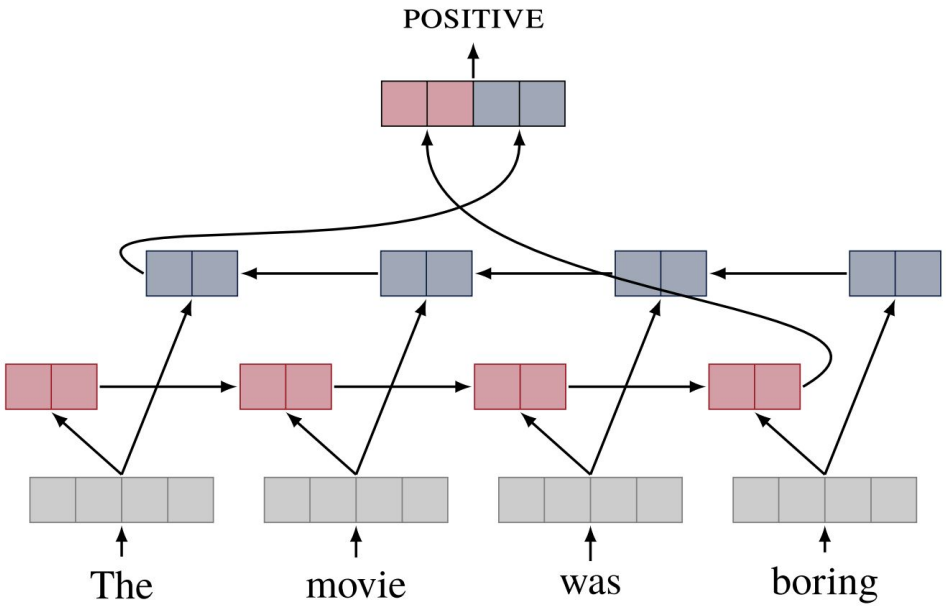
The vanishing gradient problem

- BPTT calculates gradients backward through time, it involves taking derivatives of the loss with respect to the model's parameters at each time step
- These derivatives are multiplied together as they are propagated backward
- Since gradients are multiplied together, if the gradients at each time step are less than 1, this multiplication leads to a compounding effect
  - As you go further back in time, the gradients become increasingly smaller
- The compounding effect causes the gradients for early time steps to become vanishingly small, approaching zero
  - When the gradients are too close to zero, they don't provide meaningful information for parameter updates
  - This makes it challenging for the RNN to learn long-term dependencies in the data

Recurrent architectures for related tasks

RNNs for text classification

- Last hidden state of the RNN represents the entire sentence.



Recurrent architectures for related tasks

What other tasks can RNNs handle?

Next to come

- Seq2seq models
  - Encoders and decoders
- Attention

Practice

Equations for LSTM

$f_t = \text{sigmoid}(W_f x_t + U_f h_{t-1} + b_f)$

$i_t = \text{sigmoid}(W_i x_t + U_i h_{t-1} + b_i)$

$o_t = \text{sigmoid}(W_o x_t + U_o h_{t-1} + b_o)$

- All the weights (Ws and Us and bs) will be given. You need to calculate f, i and o.
- How to calculate sigmoid/tanh for a vector (because the argument for sigmoid in this case will be a vector)
  - sigmoid([1, 2, 0]) = [sigmoid(1), sigmoid(2), sigmoid(0)]
  - tanh([1, 2, 0]) = [tanh(1), tanh(2), tanh(0)]

Practice

How to calculate f_t for input $x_t = [1\ 1]^T$

- Given: $W_f = [1\ 1, 0\ 1]$, $U_f = [0\ 0\ , 2\ 3]$, $h_{t-1} = [4\ 5]^T$, $b_f = [0\ 0]^T$
- Using this equation: $f_t = \text{sigmoid}(W_f x_t + U_f h_{t-1} + b_f)$
  - Multiply W_f with x_t (W_f has a shape of 2x2, xt has a shape of 2x1), output will be a 2x1 vector
  - Multiply U_f with h_{t-1} (U_f has a shape of 2x2, ht-1 has a shape of 2x1), output will be a 2x1 vector
  - b_f already is a 2x1 vector
  - So, f_t will also be a 2x1 vector

Then calculate i_t and o_t and use these values to calculate the C_t and h_t

$h_t = o_t * \tanh(C_t)$

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

$\hat{C}_t = \tanh(W_c x_t + U_c h_{t-1} + b_c)$

P.S. * means element-wise multiplication

Practice

$W_f X_t = [2\ 1]^T$, $U_f h_{t-1} = [0\ 23]^T$ $bf = [0\ 0]^T$, so $W_f x_t + U_f h_{t-1} + b = [2\ 24]^T$

$f_t = \text{sigmoid}([2\ 24]^T) = [\text{sigmoid}(2)\ \text{sigmoid}(24)]^T$

$Ct = f_t * C_{t-1} + i_t * \hat{C}_t$

If $C_{t-1} = [1\ 5]^T$

$f_t * C_{t-1} = [\text{sigmoid}(2)\ \text{sigmoid}(6)]^T * [1\ 5]^T$

$= [\text{sigmoid}(2)*1\ \text{sigmoid}(24)*5]^T$ which is another 2x1 matrix