# Topics in Natural Language Processing

Shay Cohen

Institute for Language, Cognition and Computation

University of Edinburgh

Lecture 7

# Log-Linear Models

The gradient of the log-likelihood of a log-linear model is the difference between the average value of the features in the data and the expected value of the features according to the model

Question: can we forego a probabilistic interpretation and focus on the linear score?

# Decoding with Log-Linear Models

Given an $x$, want to find the most likely $y$ given an $x$:

$$y^* = \arg\max_y p(y \mid x, w) = \arg\max_y \frac{\exp(w \cdot \phi(x, y))}{Z(x, w)}$$

$$= \arg\max_y \exp(w \cdot \phi(x, y)) = \arg\max_y \log \exp(w \cdot \phi(x, y))$$

$$= \arg\max_y w \cdot \phi(x, y)$$

# Linear Score Models

A non-probabilistic model that only considers the score

$$\text{score}(x, y, w) = w \cdot \phi(x, y)$$

Decoding remains the same:

$$y^* = \arg\max_y w \cdot \phi(x, y)$$

Can we train the model directly with this score?

# Linear Score Models

A non-probabilistic model that only considers the score

$$\text{score}(x, y, w) = w \cdot \phi(x, y)$$

Decoding remains the same:

$$y^* = \arg\max_y w \cdot \phi(x, y)$$

Can we train the model directly with this score?
Yes! For example, with the Perceptron.

# Training Linear Score Models

The Perceptron algorithm:

- Initialise $w$ to $0$.
- For $T$ iterations
  - For each labelled pair $(x, y_{\text{gold}})$ in the data
    - $y_{\text{predict}} = \arg\max_y w \cdot \phi(x, y)$
    - $w \leftarrow w + \phi(x, y_{\text{gold}}) - \phi(x, y_{\text{predict}})$

# Intuition Behind Perceptron

Main update rule in step $t$:

$$w \leftarrow w + \phi(x_t, y_{\text{gold}}) - \phi(x, y_{\text{predict}})$$

- Features that fire in the correct structure positively predict the structure, and if they don't fire in the prediction, we need to increase their weight to make them more "important" (increase the score when they fire)
- If we don't make a mistake, there won't be an update!

# Avoiding Overfitting with the Perceptron

The *averaged* Perceptron: maintain a $w_{\text{average}}$ which is the average of all weight vectors $w$ after each update (whether it happened or not).

Return $w_{\text{average}}$ as the final weight vector

Each $w$ is most adapted to the last example it has seen. Averaging treats each $w$ as a separate classifier, and then takes the average of all scores from all of these classifiers

# Two Interpretations of the Perceptron

- The Mistake Bound Model
- Optimising an objective function

# Perceptron Correctness (Mistake Bound Model)

Correctness guarantee:

# Stochastic Gradient Descent

The Perceptron algorithm can be viewed as a *stochastic subgradient descent* algorithm.

- Stochastic: instead of making an update to the full objective function, summing over all examples, we make an update for an example at a time

$$L(w) = \sum_{i=1}^{n} \ell_i(w)$$

  Update at each step with the gradient $\frac{\partial \ell_i}{\partial w}$.

- How is the Perceptron related to SSGD?

# Perceptron - More Intuition

Consider the minimisation $\min_w \max_{y'} \{0, (\phi(x_i, y') - \phi(x_i, y_i)) \cdot w\}$.
What does it mean?

# Perceptron as Objective Maximisation

We usually think in terms of optimising an objective function (like with log-linear models). Does the Perceptron optimise any function for a training set $(x_1, y_1), \ldots, (x_n, y_n)$?
Consider

$$P(w, x_i, i \in [n]) = \arg\min_w \lambda ||w||_2^2 + \frac{1}{n} \sum_i \max_{y'} \{0, \left(\phi(x_i, y') - \phi(x_i, y_i)\right) \cdot w\}$$

- Maximising $\left(\phi(x_i, y') - \phi(x_i, y_{\text{gold}})\right) \cdot w$ gives the $y'$ that is closest to $y_{\text{gold}}$ in its score.
- Minimising the whole objective function tries to minimise this score difference
- The $||w||_2^2$ term is for regularisation

# Subgradient

To optimise this function $P$ we can calculate its "subgradient"

$$P(w, x_i, i \in [n]) = \arg\min_w \lambda ||w||_2^2 + \frac{1}{n} \sum_i \max_{y'} \{0, \left(\phi(x_i, y') - \phi(x_i, y_i)\right) \cdot w\}$$

This is a generalisation of the notion of gradient (because the $\max$ function is not differentiable according to standard gradient calculations), and it gives the update of the Perceptron with $\lambda = 2$:
$w \leftarrow w + \phi(x, y_{\text{gold}}) - \phi(x, y_{\text{predict}})$

Note that we are doing "stochastic optimisation" – at each step updating the weights with a single example

# Support Vector Machines

Similar to the Perceptron, only with a slightly different objective function:

$$\arg\min_{w} \lambda||w||_2^2 + \frac{1}{n}\sum_i \max_{y'}\{0, \Delta(y_i, y') + \big(\phi(x_i, y') - \phi(x_i, y_i)\big) \cdot w\}$$

$\Delta(y_i, y')$ is a loss function that tells how far $y_i$ is from $y$ (for example, accuracy of labels in a sequence)

Idea: We take into account not just the linear score, but also how well $y'$ is according to some evaluation metric

Same update as the Perceptron, only we need to find $y_{\text{predict}}$ such that

$$y_{\text{predict}} = \arg\max_{y'} \Delta(y_i, y') + \phi(x_i, y_i) - \phi(x_i, y')$$

# Summary

- Different types of algorithms for linear model learning
- We learned about: log-linear models, linear models with the Perceptron and linear models with Support Vector Machines
- There is an active area of research that adds nuances to these ideas for better optimisation and other learning objectives

# What is a Neural Network?

# The Three Buckets (Andrew Ng)

- Traditional feed-forward/deep learning (classification)
- Convolutional neural networks (used for vision)
- Sequence learning: LSTM, recurrent NN (used for language)
- Another bucket: reinforcement learning?

# Deep Learning – Why Now?

# Neural Networks

- The area has developed its own terminology
- Logistic regression is a simple neural network with “softmax” activation:

# Training a Neural Network

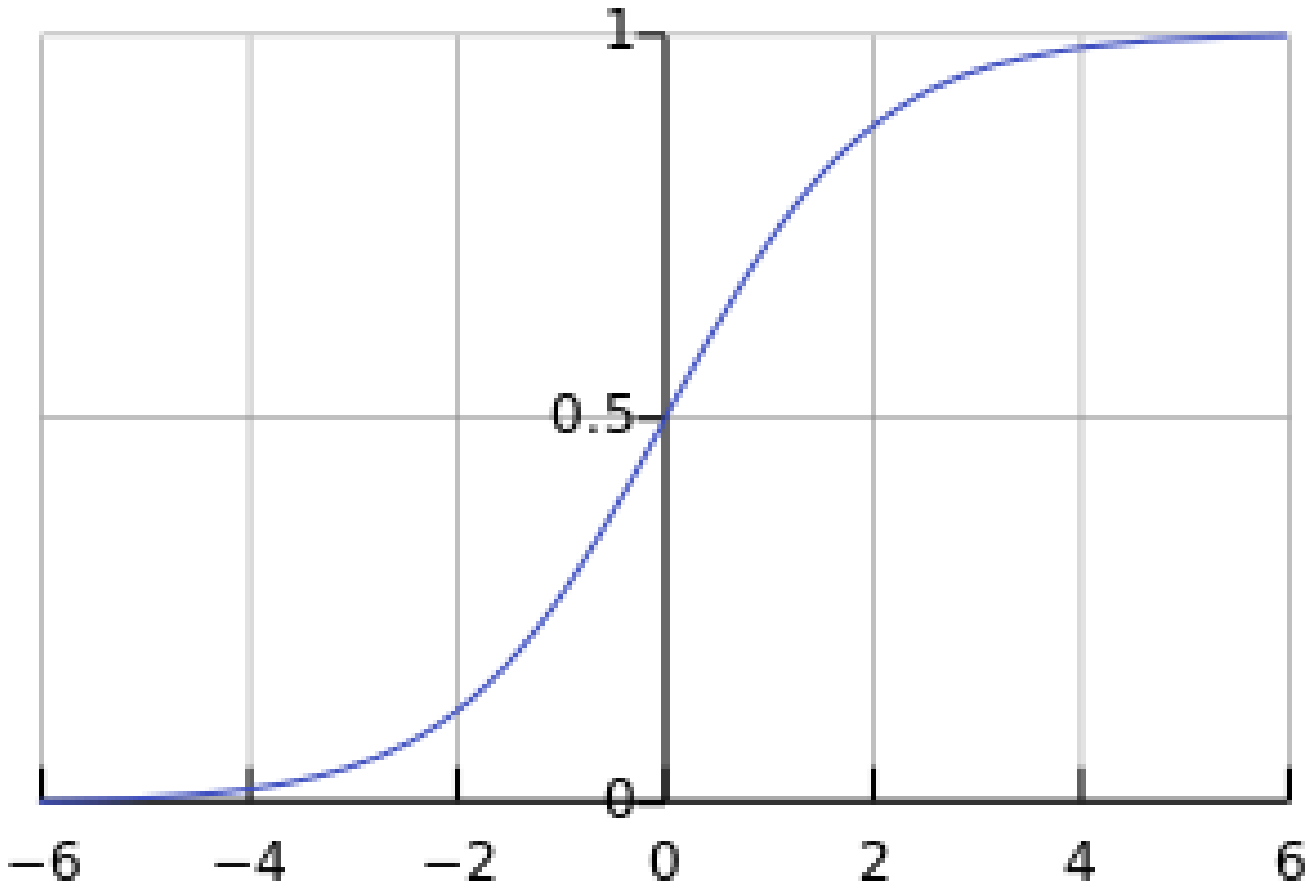
The backprop algorithm:

- An application of the chain rule: the rate of change with respect to a variable $x$ is the sum of rate of changes with respect to other variables $z_i$ multiplied by the rate of change of $z_i$ with respect to $x$

$$\frac{\partial f}{\partial x} = \sum_{i=1}^{d} \frac{\partial f}{\partial z_i} \frac{\partial z_i}{\partial x}$$

- The "extra" variables we use are the activations in different parts of the network: the derivative of the output with respect to a parameter is the derivative of the output with respect to its activation times the derivative of the activation with respect to a parameter... and apply it recursively

# What Happens When Deep is Really Deep?




The sigmoid function “squashes” values into the $[0, 1]$ range

# Vanishing Gradient Problem

Gradient descent works by checking how changes in the weights make changes to the final output of the neural network

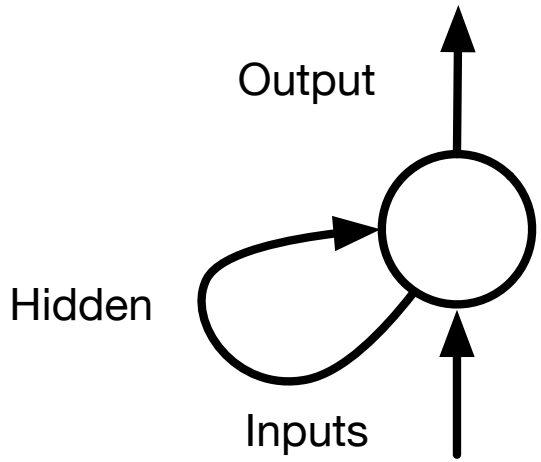
With sigmoid or tanh, the values at each layer are “squashed” into a small range

This means that even large changes in the weights, especially in the early layers, make small changes in the final output

$\Rightarrow$ slow convergence or even worse than that

Solution: ReLU activation (instead of sigmoid/tanh), Long Short Term Memory networks (see next)

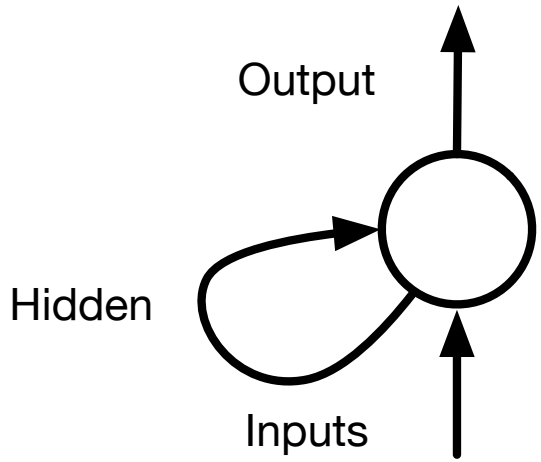
# Recurrent Architectures




Example applications: (1) neural MT; (2) character recognition; (3) grammar learning; (4) language modelling.

There is an internal state that memorises context up to that point. Similarity to HMMs?

# Recurrent Architectures

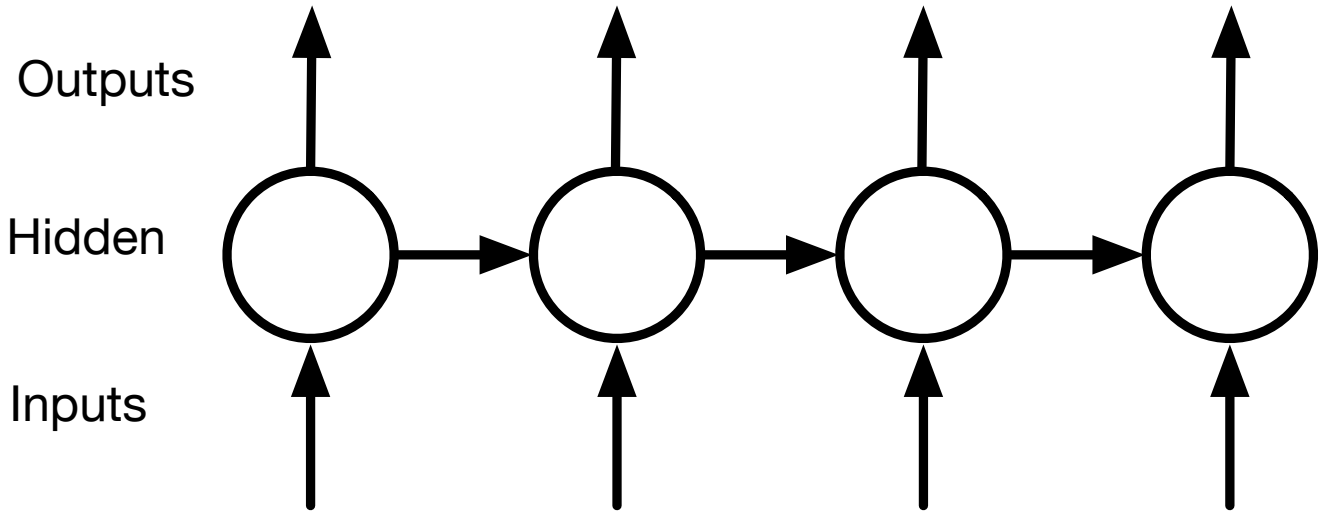



Example applications: (1) neural MT; (2) character recognition; (3) grammar learning; (4) language modelling.

There is an internal state that memorises context up to that point. Similarity to HMMs?

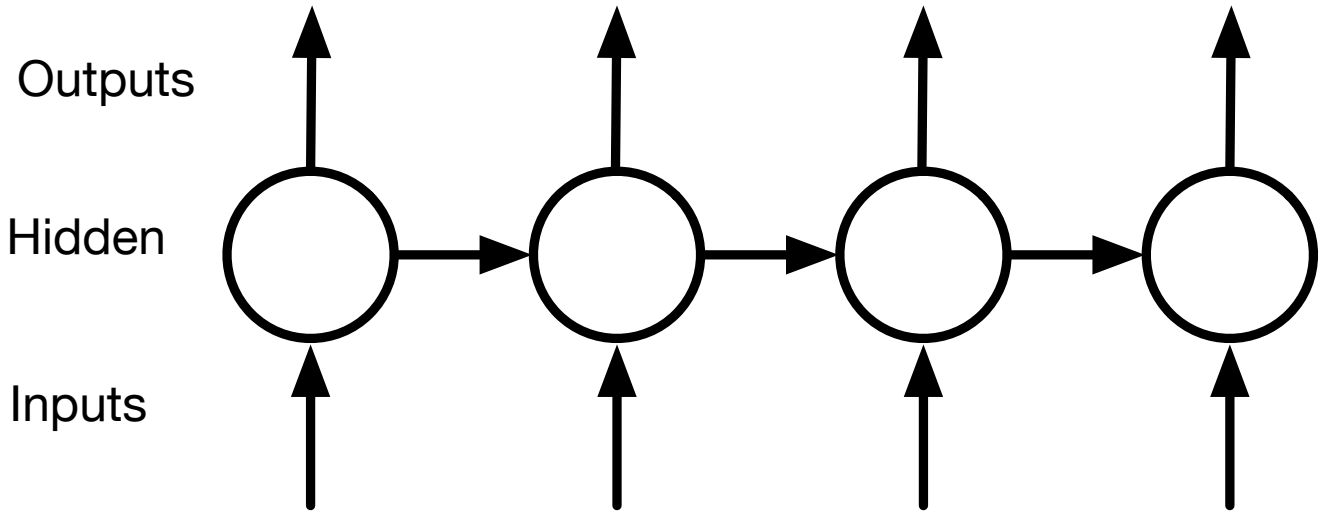
How do we train a recurrent network?

# Training Recurrent Architectures




- “Unroll” the inputs and the outputs of the network into a long sequence (or larger structure)
- Then one can use backpropgation!

# Training Recurrent Architectures




- “Unroll” the inputs and the outputs of the network into a long sequence (or larger structure)
- Then one can use backpropgation!
- Problem: vanishing gradient again...

# Long Short Term Memory (LSTM)

Replace each hidden unit with an LSTM unit at each time step $t$
An LSTM unit maintains a state $C_t$, just like a recurrent neural network. It depends on the the previous state, the previous output ($o_{t-1}$) and the current input ($x_t$)

- The current state is affected by a “forget” component that can reduce the influence of previous activation

# Long Short Term Memory (LSTM)

Replace each hidden unit with an LSTM unit at each time step $t$
An LSTM unit maintains a state $C_t$, just like a recurrent neural network. It depends on the the previous state, the previous output ($o_{t-1}$) and the current input ($x_t$)

- The current state is affected by a "forget" component that can reduce the influence of previous activation

- The current state also takes into account current input and previous output with an "input update" degree component

$$C_t = f_t C_{t-1} + i_t \hat{h}_t \qquad \hat{h}_t = \tanh(W_C[o_{t-1}, x_t] + b_C)$$

$f_t$ and $i_t$ are values between $0$ and $1$ based on a sigmoid over $o_{t-1}$ and $x_t$

# Long Short Term Memory (LSTM)

Replace each hidden unit with an LSTM unit at each time step $t$
An LSTM unit maintains a state $C_t$, just like a recurrent neural network. It depends on the the previous state, the previous output ($o_{t-1}$) and the current input ($x_t$)

- The current state is affected by a "forget" component that can reduce the influence of previous activation

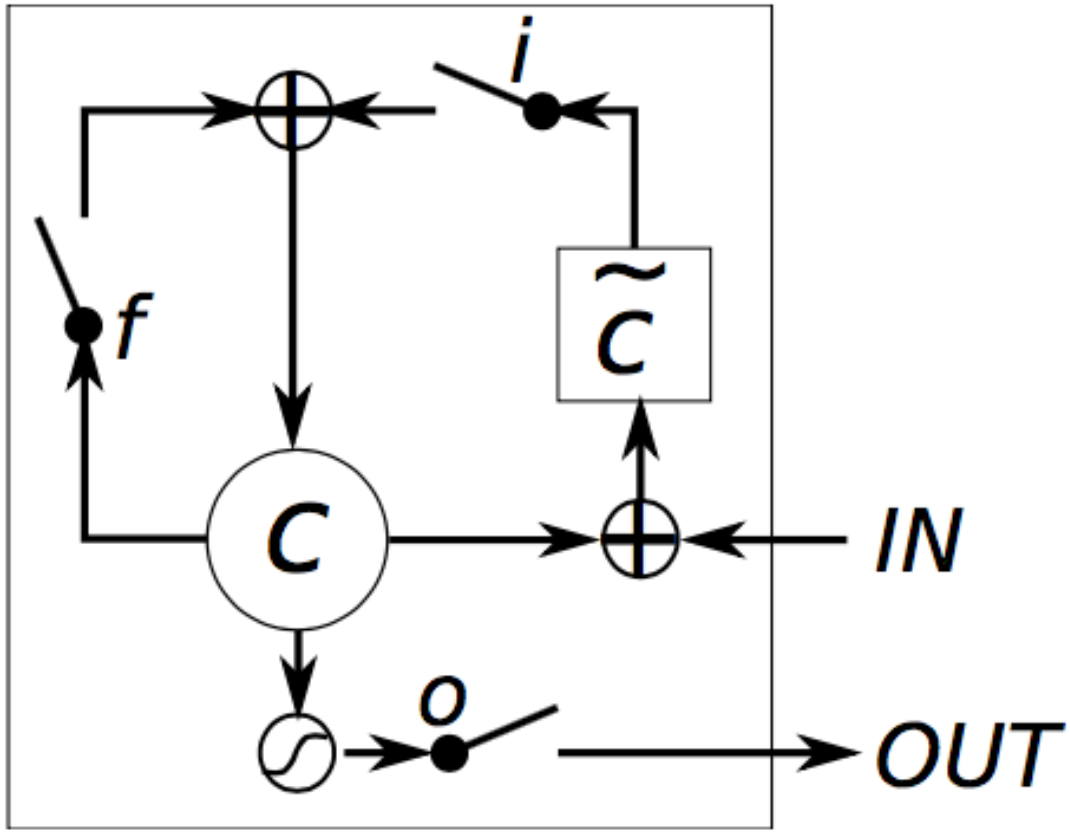
- The current state also takes into account current input and previous output with an "input update" degree component

$$C_t = f_t C_{t-1} + i_t \hat{h}_t \qquad \hat{h}_t = \tanh(W_C[o_{t-1}, x_t] + b_C)$$

$f_t$ and $i_t$ are values between $0$ and $1$ based on a sigmoid over $o_{t-1}$ and $x_t$

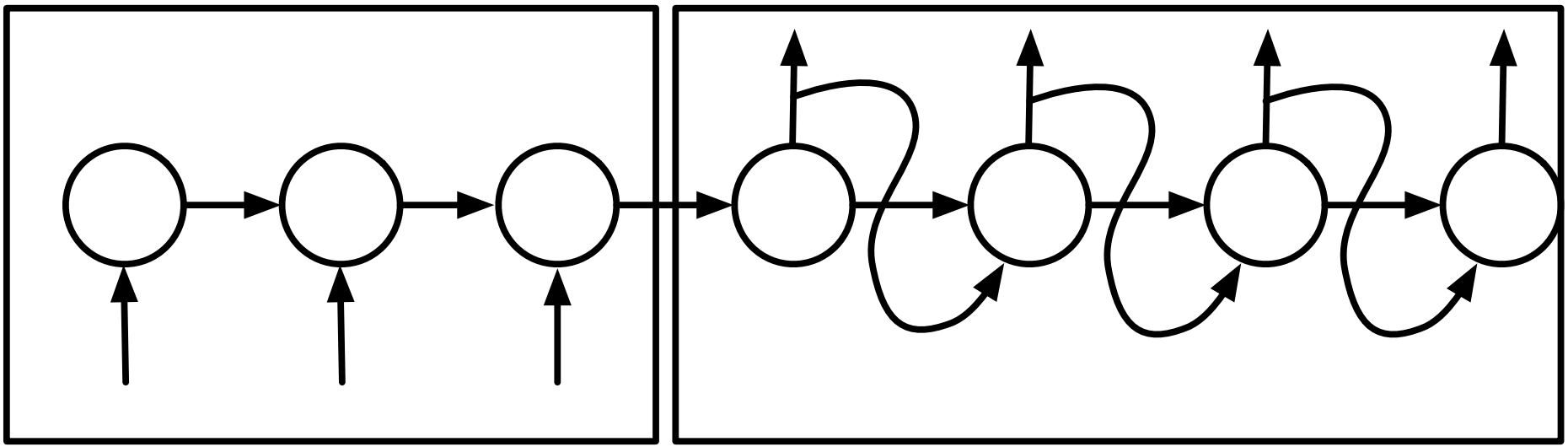
In addition, the output $o_t$ depends on the current state ($C_t$), the previous output ($o_{t-1}$) and the current input ($x_t$)

# LSTM Graphical Depiction



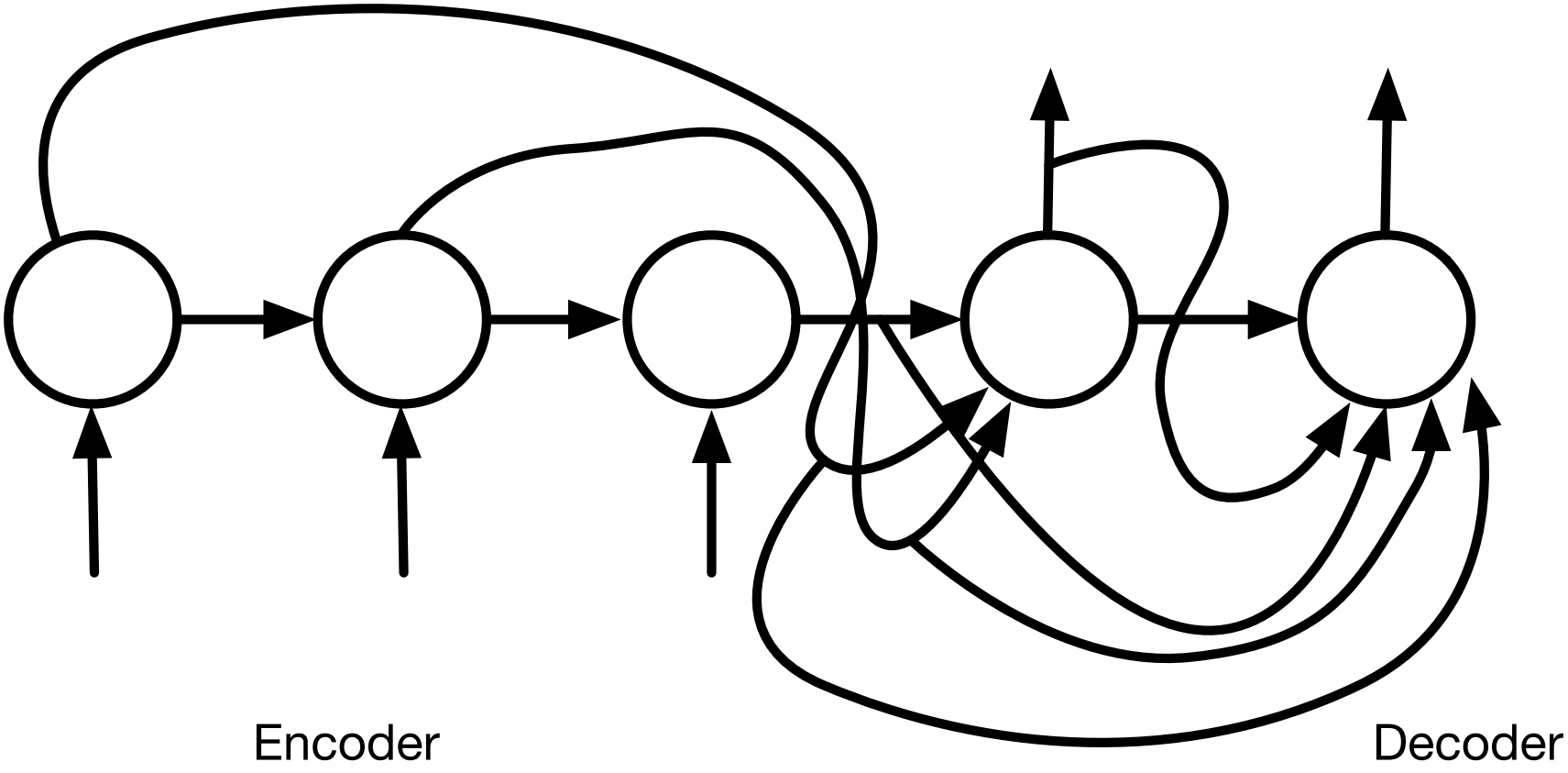

taken from Chung et al. (2014)

# Sequence to Sequence Models




- Each cell is an LSTM cell
- The encoder “encodes” the input into a vector, starting state
- The decoder “decodes” it to the output

# Sequence to Sequence Models




- Each encoder output is connected to each of the decoder cells as input

# Summary

- Large and deep neural networks have the “vanishing gradient” problem
- Also true for recurrent architectures
- LSTMs are one way to fix that issue
- They implement sequence-to-sequence models (with and without attention)
- Many off-the-shelf packages for implementing neural networks and sequence-to-sequence models