Where are we?
Brain-inspired learning
Modeling neurons

Each neuron is a *node*.

Directed *edges* show connections between neurons.

- Neurons get their input from many other neurons.
- Neurons send their output to many other neurons.

Edge *weights* show the strength of connections.
What a neuron computes

1. Multiply each input by the corresponding edge’s weight.
   
   \[ 2 \times 0.8, \ 0 \times -2.2, \ -3 \times 3.5 \]

2. Sum the weighted activations.

   \[ 2 \times 0.8 + 0 \times -2.2 + -3 \times 3.5 = -8.9 \]

3. Pass this weighted sum to the activation function.

   \[ f(x) = \begin{cases} 
   0 & x < -2 \\
   1 & x \geq -2 
   \end{cases} \]
   
   \[ f(-8.9) = 0 \]
Bias input

We’d rather not set the threshold for every node, e.g., $x < -2$ vs $x \geq -2$.

Instead, we give every node an extra input, fixed at 1.

The weight on this node’s edge changes the threshold.
Dot product formulation

We can view weights and inputs as vectors and the weighted sum as a dot product:

\[
\begin{bmatrix}
1 \\
2 \\
0 \\
-3
\end{bmatrix} \cdot \begin{bmatrix}
-2 \\
0.8 \\
-2.2 \\
3.5
\end{bmatrix} = -10.9
\]
A perceptron is a *linear classifier.*

Examples are points
Any weight vector is a hyperplane
On one side of the hyperplane are the positive points; on the other are the negative points.
Linearly separable functions

VS
And

Or

Exclusive Or

Linearly separable
Linearly separable
Not linearly separable
Perceptron learning

**Goal**: Given a set of (input, output) examples, automatically come up with weights so that the perceptron represents the function \( f(\text{input}) = \text{output} \).

**Key idea**: Loop through the training data and update weights when wrong.

The update is almost identical to the approximate q-learning update.
Perceptron learning pseudocode

initialize weights (to 0 or random values)
while any training example is misclassified:
    for each training input i:
        output = compute_perceptron_output(input)
        correction = target[i] - output
        for each input weight w[i]:
            w[i] += learning_rate * correction * input[i]
Perceptron learning example

\[
\begin{bmatrix}
1.0 \\
-0.8 \\
2.4
\end{bmatrix}
\cdot
\begin{bmatrix}
-0.5 \\
0.8 \\
3.0
\end{bmatrix}
= 6.06 \rightarrow 1
\]

target = 1

correction = 1 - 1 = 0

No update!
Perceptron learning example

\[
\begin{bmatrix}
1.0 \\
3.5 \\
-1.1
\end{bmatrix} \cdot \begin{bmatrix}
-0.5 \\
0.8 \\
3.0
\end{bmatrix} = -1.0 \rightarrow 0
\]

target = 1

\text{correction} = 1 - 0 = 1

Increase weights!

\[w_i += \text{learning_rate} \times \text{correction} \times \text{input}[i]\]
We can extend the idea of a perceptron from a binary classifier (1 or 0; true or false) to any number of categories, each of which is a possible output.
Neural networks: Motivation
As powerful as they are, there are simple problems perceptrons can’t solve, e.g.,

- input parity – are there an even or odd number of binary inputs
- connectedness – are the black pixels in a square binary array of black and white pixels connected pixel-to-pixel or not
We know how to construct a perceptron that learns a linear boundary for classification.

But what can we do for data like this?
To fix this problem, we need to add additional, potentially nonlinear, features to construct a decision boundary.

E.g., add $x^2$ as a feature:
Multi-layer perceptrons
Consider a perceptron that takes as input the outputs of another perceptron:

With this additional structure and weights, we can express a much wider set of functions. By increasing the complexity of our model, we in turn greatly increase its expressive power. Multi-layer perceptrons give us a generic way to represent a much wider set of functions. In fact, a multi-layer perceptron is a universal function approximator and can represent any real function, leaving us only with the problem of selecting the best set of weights to parameterize our network. This is formally stated below:

**Theorem. (Universal Function Approximators)**
A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

**Measuring Accuracy**

The accuracy of the binary perceptron after making \( m \) predictions can be expressed as:

\[
\text{acc}(w) = \frac{1}{m} \sum_{i=1}^{m} \text{sgn}(w \cdot f(x(i))) = y(i)
\]

where \( x(i) \) is datapoint \( i \), \( w \) is our weight vector, \( f \) is our function that derives a feature vector from a raw datapoint, and \( y(i) \) is the actual class label of \( x(i) \). In this context, \( \text{sgn}(x) \) represents an indicator function.
We can generalize it to an $n$-layer perceptron:

\[
\text{Theorem. (Universal Function Approximators)}
\]

A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.

Measuring Accuracy

The accuracy of the binary perceptron after making $m$ predictions can be expressed as:

\[
l_{\text{acc}}(w) = \frac{1}{m} \sum_{i=1}^{m} \text{sgn}(w \cdot f(x_i)) = y_i
\]

where $x_i$ is datapoint $i$, $w$ is our weight vector, $f$ is our function that derives a feature vector from a raw datapoint, and $y_i$ is the actual class label of $x_i$. In this context, $\text{sgn}(x)$ represents an indicator function.
Multilayer perceptrons perform computations layer by layer, left to right.

Each hidden unit computes its activation value, and these activation values become the inputs for the output units, which then compute their own activations.
Because we can represent AND, OR, and NOT with perceptrons, a neural network can represent any Boolean function.
By increasing the complexity of our model, we greatly increase its expressive power.

A multi-layer perceptron is a *universal function approximator*, which can represent any real function.

But this leaves us with the problem of selecting the right set of weights to parameterize the network!
(Multilayer) neural networks
When we put perceptrons together, the result is a graph with many neurons – a *neural network*.
A neural network is a directed acyclic graph, where

Nodes are organized into *layers*,
Outputs of one layer are inputs to the next,
Consecutive layers are densely connected
  (Though other topologies are possible!),
Edges have *weights*, and
Nodes have *activation functions*. 
For a multi-layer perceptron, we used a step function:

\[ f(x) = \begin{cases} 
1 & \text{if } x \geq 0 \\
-1 & \text{otherwise}
\end{cases} \]
A better choice is a continuous function, e.g., the \textit{sigmoid function}:

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]
However, the current favorite is the **ReLU** (rectified linear unit) function:

\[
f(x) = \begin{cases} 
0 & \text{if } x < 0 \\
x & \text{if } x \geq 0 
\end{cases}
\]
Activation functions

Threshold
\[ f(x) = \begin{cases} 
0 & x < 0 \\
1 & x \geq 0 
\end{cases} \]

Sigmoid
\[ f(x) = \frac{1}{1 + e^{-x}} \] Smooth approximation of a threshold

ReLU
\[ f(x) = \begin{cases} 
0 & x < 0 \\
x & x \geq 0 
\end{cases} \] Rectified linear unit

Others are possible!
Neural network terminology
Calculating the output of a multi-layer neural network is done as before, with the difference that at the output of each layer, we apply one of the new non-linearities instead of the initial indicator function.
The activation level for each output can be considered the confidence that that output is right.

We classify the input as the output we’re most confident in.
What are these hidden layers for?

In principle, they can learn *abstract features*, e.g.,

the first hidden layer might recognize bits of straight or curved lines,
the second hidden layer might recognize longer lines or shapes like the circles in an “8”
Deep neural networks

Image from https://blog.eduardovalle.com/tag/deep-learning
We’ve looked at networks where the signal propagation is unidirectional from layer to layer. These are called *feed-forward neural networks*. Networks that allow cyclic signal pathways are called *recurrent neural networks* (RNNs).
Interlude: An AI koan

In the days when Sussman was a novice, Minsky once came to him as he sat hacking at the PDP-6.

“What are you doing?”, asked Minsky.

“I am training a randomly wired neural net to play Tic-Tac-Toe” Sussman replied.

“So why is the net wired randomly?”, asked Minsky.

“I do not want it to have any preconceptions of how to play”, Sussman said.

Minsky then shut his eyes.

“So why do you close your eyes?”, Sussman asked his teacher.

“So that the room will be empty.”

At that moment, Sussman was enlightened.

http://catb.org/jargon/html/koans.html
Training neural networks
How can we train a neural network, iteratively improving its accuracy?

The perceptron algorithm won’t work:

- We have non-threshold activation functions.
- We have multiple layers. (What’s the correction for hidden nodes?)
Problem: SGD on threshold functions

The derivative of this function is always 0.

We can't “move in the direction of the gradient”. 
Better activation functions

**sigmoid**

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

**tanh**

$$\tanh(x) = \frac{1 + e^{-2x}}{1 - e^{-2x}}$$

**RELU**

$$\text{RELU}(x) = \begin{cases} 
0 & x < 0 \\
\ x & x \geq 0 
\end{cases}$$
Derivatives of activation functions

**sigmoid**

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

\[ \frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x)) \]

**tanh**

\[ \tanh(x) = \frac{1 + e^{-2x}}{1 - e^{-2x}} \]

\[ \frac{d\tanh(x)}{dx} = 1 - \tanh^2(x) \]

**RELU**

\[ \text{RELU}(x) = \begin{cases} 
0 & x < 0 \\
x & x \geq 0 
\end{cases} \]

\[ \frac{d\text{RELU}(x)}{dx} = \begin{cases} 
0 & x < 0 \\
1 & x > 0 
\end{cases} \]
We want to choose weights that optimize the likelihood of explaining the observed labels for the datapoints $x^{(1)}, \ldots, x^{(m)}$ in our training set:

$$\ell(w) = \prod_{i=1}^{m} P(y^{(i)} | x^{(i)}; w)$$

or, equivalently, the log-likelihood:

$$\ell\ell(w) = \sum_{i=1}^{m} \log P(y^{(i)} | x^{(i)}; w)$$
To maximize our log-likelihood function, we differentiate it to obtain a gradient vector, consisting of its partial derivatives for each parameter:

$$\nabla_w \ell(w) = \left[ \frac{\partial \ell(w)}{\partial w_1}, \ldots, \frac{\partial \ell(w)}{\partial w_n} \right]$$

What is this? It’s the local direction of the steepest rise in the likelihoods.
Gradient ascent is a greedy algorithm that calculates this gradient for the current weight parameters and updates them along the direction of the gradient, scaled by a step size, $\alpha$.

initialize weights $w$
for $i = 0, 1, 2, ...$
    $w += \alpha \nabla_w \text{ll}(w)$
If we’re minimizing instead of maximizing, we subtract the scaled gradient instead of adding it to \( w \).

This is called the \textit{gradient descent} algorithm.
Training neural networks: Optimization
Let $\mathbf{w}$ be all the parameters of our network. We want to find the values for $\mathbf{w}$ that maximize the correct classes for our data, so we run gradient ascent on the log likelihood

$$\ell \ell(\mathbf{w}) = \sum_{i=1}^{m} \log P(y^{(i)} \mid x^{(i)}; \mathbf{w})$$

where $x^{(1)}, \ldots, x^{(m)}$ are the $m$ datapoints in our training set.
One way to try minimizing the log-likelihood function is, at each iteration of gradient descent:

- use all the data points $x^{(1)}, \ldots, x^{(m)}$ to compute gradients for the parameters $\mathbf{w}$,
- update the parameters,

and repeat until the parameters converge.

(This means we’re at a local minimum of the function.)

This is called *batch gradient descent*. 
In practice, batch gradient descent is very rare because datasets are so large that computing the gradients for this full likelihood function will be very slow.

Instead, we use **mini-batching**.

This rotates through randomly sampled batches of $k$ data points at a time, taking one batch for each step of gradient descent and computing gradients of the loss function using only that batch.

Taking the sum in the log likelihood over $k$ instead of the full $m$ data points makes each gradient update much quicker, and it often still makes fast progress toward the minimum.

Where $k = 1$ (i.e., we use just a single example at each step), this is known as **stochastic gradient descent**.
**Stochastic gradient descent** (SGD)

- Compute the error on a random training example.
- Compute the derivative of the error with respect to each weight.
- Update weights in the direction that reduces error.
Neural networks are powerful universal function approximators, but they can be difficult to design and train.

There’s a lot of research looking at aspects like

*Network architectures*: How to choose activation functions, number of layers, etc. to fit a particular problem

*Learning algorithms*: How to find parameters that achieve a low value of the loss function, given that neural networks can have many local optima

*Generalization and transfer learning*: How do you guarantee low loss on testing data rather than overfit?
Training neural networks: Backpropagation
Backpropagation represents the neural network as a dependency graph of operators and operands, called a computational graph, e.g.,

\[ 2 \times 3 + 4 \times 4 + 2 \]

The Chain Rule

The chain rule is the fundamental rule from calculus which both motivates the usage of computation graphs and allows for a computationally feasible backpropagation algorithm. Mathematically, it states that for a variable \( z \) which is a function of \( n \) variables \( x_1, \ldots, x_n \) and each \( x_i \) is a function of \( m \) variables \( t_1, \ldots, t_m \), then we can compute the derivative of \( z \) with respect to any \( t_i \) as follows:

\[
\frac{\partial f}{\partial t_i} = \frac{\partial f}{\partial x_1} \cdot \frac{\partial x_1}{\partial t_i} + \frac{\partial f}{\partial x_2} \cdot \frac{\partial x_2}{\partial t_i} + \ldots + \frac{\partial f}{\partial x_n} \cdot \frac{\partial x_n}{\partial t_i}
\]

In the context of computation graphs, this means that to compute the gradient of a given node \( t_i \) with respect to the output \( z \), we take a sum of children \( t_i \) terms.

The Backpropagation Algorithm

Figure 1: A computation graph for computing \((x + y) \times z\) with the values \( x = 2, y = 3, z = 4 \).
From calculus, we have the \textit{chain rule}, which will make the backpropagation algorithm over a computational graph feasible:

For a variable \( z \) that's a function of variables \( x_1, \ldots, x_n \), where each \( x_i \) is a function of variables \( t_1, \ldots, t_m \), we can compute the derivative of \( z \) with respect to any \( t_i \) as:

\[
\frac{\partial f}{\partial t_i} = \frac{\partial f}{\partial x_1} \cdot \frac{\partial x_1}{\partial t_i} + \frac{\partial f}{\partial x_2} \cdot \frac{\partial x_2}{\partial t_i} + \cdots + \frac{\partial f}{\partial x_n} \cdot \frac{\partial x_n}{\partial t_i}
\]

For computational graphs, this means that to compute the gradient of a node \( t_i \) with respect to output \( z \), we take a sum of of the children of \( t_i \).
Suppose we’re computing \((x + y) \cdot z\) with values \(x = 2, y = 3, z = 4\).

Let \(g\) be the \(x + y\) part and \(f\) be \(g \cdot z\):

![Computation graph](image-url)
Suppose we’re computing \((x + y) \cdot z\) with values \(x = 2, y = 3, z = 4\).

Let \(g\) be the \(x + y\) part and \(f\) be \(g \cdot z\):
Suppose we’re computing \((x + y) \cdot z\) with values \(x = 2, y = 3, z = 4\).

Let \(g\) be the \(x + y\) part and \(f\) be \(g \cdot z\):

![Computation Graph](image)

**Figure 1:** A computation graph for computing \((x + y) \cdot z\) with the values \(x = 2, y = 3, z = 4\).
Intuitively, each node’s gradient measures how much a change in that node’s value contributes to a change in the final node’s value.

This will be the product of how much the node contributes to a change in its child node, with how much the child node contributes to a change in the final node.

Each node receives and combines gradients from its children, updates this combined gradient based on the node’s inputs and the node’s operation, and then passes the updated gradient backward to its parents.
Taken together, the *forward pass* – applying each node’s operation to its input values – and the *backward pass* – computing the gradients of the function – make up the *backpropagation algorithm*. 
Backpropagation algorithm

for each training run:
    for example in shuffled training data:
        run example through network
        compute error for each output node
    for each layer (starting from output):
        for each node in layer:
            gradient descent update on incoming weights
Acknowledgments

The lecture incorporates material from:

Dan Klein and Pieter Abbeel, University of California, Berkeley; ai.berkeley.edu
George Konidaris, Brown
Lisa Meeden, Swarthmore
Melanie Mitchell, Santa Fe Institute / Portland State
Peter Norvig and Stuart Russell, *Artificial Intelligence: A Modern Approach*
Ketrina Yim (illustrations)