Review: Perceptron

\[ y = \begin{cases} 
1 & w_1 x_1 + w_2 x_2 + w_3 x_3 > 0 \\
0 & \text{otherwise}
\end{cases} \]
Properties of Perceptrons

- **Separability**: true if some parameters get the training set perfectly correct.

- **Convergence**: if the training is separable, perceptron will eventually converge (binary case).

- **Mistake Bound**: the maximum number of mistakes (binary case) related to the *margin* or degree of separability.

\[
\text{mistakes} < \frac{k}{\delta^2}
\]
Problems with the Perceptron

- **Noise:** if the data isn’t separable, weights might thrash
  - Averaging weight vectors over time can help (averaged perceptron)

- **Mediocre generalization:** finds a “barely” separating solution

- **Overtraining:** test / held-out accuracy usually rises, then falls
  - Overtraining is a kind of overfitting
Improving the Perceptron
Non-Separable Case: Deterministic Decision

Even the best linear boundary makes at least one mistake
Non-Separable Case: Probabilistic Decision
How to get deterministic decisions?

- Perceptron scoring: $z = w \cdot f(x)$

- If $z = w \cdot f(x)$ positive $\rightarrow$ classifier says: 1.0 probability this is class +1

- If $z = w \cdot f(x)$ negative $\rightarrow$ classifier says: 0.0 probability this is class +1

- Step function

  $H(z) = \begin{cases} 
  1 & z > 0 \\
  0 & z \leq 0
  \end{cases}$

- $z =$ output of perceptron
  $H(z) =$ probability the class is +1, according to the classifier
How to get probabilistic decisions?

- Perceptron scoring: \( z = w \cdot f(x) \)
- If \( z = w \cdot f(x) \) very positive \( \rightarrow \) probability of class +1 should approach 1.0
- If \( z = w \cdot f(x) \) very negative \( \rightarrow \) probability of class +1 should approach 0.0

- Sigmoid function

\[ \phi(z) = \frac{1}{1 + e^{-z}} \]

- \( z \) = output of perceptron
  \( \phi(z) = \) probability the class is +1, according to the classifier
Probabilistic Decisions: Example

Suppose $w = [-3, 4, 2]$ and $x = [1, 2, 0]$

What label will be selected if we classify deterministically?
- $wx = -3 + 8 + 0 = 5$
- 5 is positive, so the classifier guesses the positive label

What are the probabilities of each label if we classify probabilistically?
- $\frac{1}{1 + e^{-wx}} = \frac{1}{1 + e^{-5}} = 0.9933$ probability of positive label
- $1 - 0.9933 = 0.0067$ probability of negative label
A 1D Example

\[ P(\text{red}|x) = \frac{1}{1 + e^{-wx}} \]

where \( w \) is some weight constant (1D vector) we have to learn (assume \( w \) is positive in this example)
Best w?

- Recall maximum likelihood estimation: Choose the w value that maximizes the probability of the observed (training) data

\[
\text{Likelihood} = P(\text{training data}|w) = \prod_i P(\text{training datapoint } i \mid w) = \prod_i P(\text{point } x^{(i)} \text{ has label } y^{(i)}|w) = \prod_i P(y^{(i)}|x^{(i)}; w)
\]

\[
\text{Log Likelihood} = \sum_i \log P(y^{(i)}|x^{(i)}; w)
\]
Recall maximum likelihood estimation: Choose the $w$ value that maximizes the probability of the observed (training) data

\[
P(\text{point } x^{(i)} \text{ has label } y^{(i)} = +1 \mid w) = \frac{1}{1 + e^{-w \cdot x^{(i)}}}
\]

\[
P(\text{point } x^{(i)} \text{ has label } y^{(i)} = -1 \mid w) = 1 - \frac{1}{1 + e^{-w \cdot x^{(i)}}}
\]
Best w?

- Maximum likelihood estimation:

\[
\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)};w)
\]

with:

\[
P(y^{(i)} = +1|x^{(i)};w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}
\]

\[
P(y^{(i)} = -1|x^{(i)};w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}
\]

= Logistic Regression
Logistic Regression Example

- What function are we trying to maximize for this training data?
  - Data point [2, 1] is class +1
  - Data point [0, –2] is class +1
  - Data point [–1, –1] is class –1

\[
\max_w ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)} ; w)
\]

\[
P(y^{(i)} = +1 | x^{(i)} ; w) = \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}
\]

\[
P(y^{(i)} = -1 | x^{(i)} ; w) = 1 - \frac{1}{1 + e^{-w \cdot f(x^{(i)})}}
\]
Logistic Regression Example

- What function are we trying to maximize for this training data?
  - Data point \([2, 1]\) is class +1
  - Data point \([0, -2]\) is class +1
  - Data point \([-1, -1]\) is class -1

\[
\arg\max_w \left[ \log \left( \frac{1}{1 + e^{-(2w_1 + w_2)}} \right) + \log \left( \frac{1}{1 + e^{-(-2w_2)}} \right) + \log \left( \frac{1}{1 + e^{-(w_1 - w_2)}} \right) \right]
\]
Separable Case: Deterministic Decision – Many Options
Separable Case: Probabilistic Decision – Clear Preference
Multiclass Logistic Regression

- **Recall Perceptron:**
  - A weight vector for each class: \( w_y \)
  - Score (activation) of a class \( y \): \( w_y \cdot f(x) \)
  - Prediction highest score wins: \( y = \arg \max_y w_y \cdot f(x) \)

- **How to make the scores into probabilities?**

\[
\begin{align*}
 z_1, z_2, z_3 \rightarrow & \quad \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}, \quad \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}, \quad \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}} \\
\end{align*}
\]

original activations \quad softmax activations
Multi-Class Probabilistic Decisions: Example

\[ z_1, z_2, z_3 \rightarrow \frac{e^{z_1}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_2}}{e^{z_1} + e^{z_2} + e^{z_3}}, \frac{e^{z_3}}{e^{z_1} + e^{z_2} + e^{z_3}} \]

- Suppose \( w_1 = [-3, 4, 2], w_2 = [2, 2, 7], w_3 = [0, -1, 0], \) and \( x = [1, 2, 0] \)
- What label will be selected if we classify deterministically?
  - \( w_1 \cdot x = 5, \) and \( w_2 \cdot x = 6, \) and \( w_3 \cdot x = -2 \)
  - \( w_2 \cdot x \) has the highest score, so the classifier guesses class 2
- What are the probabilities of each label if we classify probabilistically?
  - Probability of class 1: \( \frac{e^5}{e^5 + e^6 + e^{-2}} = 0.2689 \)
  - Probability of class 2: \( \frac{e^6}{e^5 + e^6 + e^{-2}} = 0.7310 \)
  - Probability of class 3: \( \frac{e^{-2}}{e^5 + e^6 + e^{-2}} = 0.0002 \)
Recall maximum likelihood estimation: Choose the $w$ value that maximizes the probability of the observed (training) data

\[
\text{Likelihood} = P(\text{training data}|w)
= \prod_i P(\text{training datapoint } i \mid w)
= \prod_i P(\text{point } x^{(i)} \text{ has label } y^{(i)}|w)
= \prod_i P(y^{(i)}|x^{(i)}; w)
\]

Log Likelihood $= \sum_i \log P(y^{(i)}|x^{(i)}; w)$
Best w?

- Maximum likelihood estimation:

\[
\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)
\]

with:

\[
P(y^{(i)}|x^{(i)}; w) = \frac{e^{w_{y(i)} \cdot f(x^{(i)})}}{\sum_y e^{w_y \cdot f(x^{(i)})}}
\]

= Multi-Class Logistic Regression
Multi-Class Logistic Regression Example

- What function are we trying to maximize for this training data?
  - Data point [2, 1] is class Red
  - Data point [0, –2] is class Green
  - Data point [–1, –1] is class Blue

\[
\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)} | x^{(i)}; w)
\]

\[
P(y^{(i)} | x^{(i)}; w) = \frac{e^{w_y(i) \cdot f(x^{(i)})}}{\sum_y e^{w_y \cdot f(x^{(i)})}}
\]
What function are we trying to maximize for this training data?

- Data point [2, 1] is class Red
- Data point [0, –2] is class Green
- Data point [–1, –1] is class Blue

\[
\text{argmax}_w \left[ \log \left( \frac{e^{2w_1+w_2}}{e^{2w_1+w_2}+e^{2w_1+w_2}+e^{2w_1+w_2}} \right) + \log \left( \frac{e^{-2w_2}}{e^{-2w_2}+e^{-2w_2}+e^{-2w_2}} \right) + \log \left( \frac{e^{-w_1-w_2}}{e^{-w_1-w_2}+e^{-w_1-w_2}+e^{-w_1-w_2}} \right) \right]
\]

Log probability of [2, 1] being red
Log probability of [0,–2] being green
Log probability of [–1, –1] being blue
Softmax with Different Bases

\[ P(\text{red}|x) = \frac{e^{w_{\text{red}} \cdot x}}{e^{w_{\text{red}} \cdot x} + e^{w_{\text{blue}} \cdot x}} \]

The expression looks like \( \max_y w_y \cdot x \)
Recall: Binary perceptron is a special case of multi-class perceptron

- Multi-class: Compute $w_y \cdot f(x)$ for each class $y$, pick class with the highest activation
- Binary case:
  - Let the weight vector of $+1$ be $w$ (which we learn).
  - Let the weight vector of $-1$ always be 0 (constant).
- Binary classification as a multi-class problem:
  - Activation of negative class is always 0.
  - If $w \cdot f$ is positive, then activation of $+1$ ($w \cdot f$) is higher than $-1$ (0).
  - If $w \cdot f$ is negative, then activation of $-1$ (0) is higher than $+1$ ($w \cdot f$).

$$P(\text{red}|x) = \frac{e^{w_{\text{red}} \cdot x}}{e^{w_{\text{red}} \cdot x} + e^{w_{\text{blue}} \cdot x}}$$

with $w_{\text{red}} = 0$ becomes:

$$P(\text{red}|x) = \frac{1}{1 + e^{-wx}}$$
Optimization

i.e., how do we solve:

\[
\max_{w} \quad ll(w) = \max_{w} \quad \sum_{i} \log P(y^{(i)} | x^{(i)}; w)
\]
Review: Derivatives and Gradients

- What is the derivative of the function \( g(x) = x^2 + 3 \)?

\[
\frac{dg}{dx} = 2x
\]

- What is the derivative of \( g(x) \) at \( x=5 \)?

\[
\frac{dg}{dx} \bigg|_{x=5} = 10
\]
What is the gradient of the function $g(x, y) = x^2y$?

- Recall: Gradient is a vector of partial derivatives with respect to each variable

\[
\nabla g = \begin{bmatrix}
\frac{\partial f}{\partial x} \\
\frac{\partial f}{\partial y}
\end{bmatrix} = \begin{bmatrix}
2xy \\
x^2
\end{bmatrix}
\]

What is the derivative of $g(x, y)$ at $x=0.5$, $y=0.5$?

\[
\nabla g\bigg|_{x=0.5, y=0.5} = \begin{bmatrix}
2(0.5)(0.5) \\
(0.5^2)
\end{bmatrix} = \begin{bmatrix}
0.5 \\
0.25
\end{bmatrix}
\]
Hill Climbing

- Recall from CSPs lecture: simple, general idea
  - Start wherever
  - Repeat: move to the best neighboring state
  - If no neighbors better than current, quit

- What’s particularly tricky when hill-climbing for multiclass logistic regression?
  - Optimization over a continuous space
    - Infinitely many neighbors!
  - How to do this efficiently?
1-D Optimization

- Could evaluate $g(w_0 + h)$ and $g(w_0 - h)$
  - Then step in best direction

- Or, evaluate derivative:
  \[
  \frac{\partial g(w_0)}{\partial w} = \lim_{h \to 0} \frac{g(w_0 + h) - g(w_0 - h)}{2h}
  \]
  - Tells which direction to step into
2-D Optimization

Source: offconvex.org
Gradient Ascent

- Perform update in uphill direction for each coordinate
- The steeper the slope (i.e. the higher the derivative) the bigger the step for that coordinate
- E.g., consider: \( g(w_1, w_2) \)

**Updates:**

\[
\begin{align*}
  w_1 & \leftarrow w_1 + \alpha \times \frac{\partial g}{\partial w_1}(w_1, w_2) \\
  w_2 & \leftarrow w_2 + \alpha \times \frac{\partial g}{\partial w_2}(w_1, w_2)
\end{align*}
\]

**Updates in vector notation:**

\[
 w \leftarrow w + \alpha \times \nabla_w g(w)
\]

with: \( \nabla_w g(w) = \begin{bmatrix} \frac{\partial g}{\partial w_1}(w) \\
\frac{\partial g}{\partial w_2}(w) \end{bmatrix} = \text{gradient} \)
Gradient Ascent

- Idea:
  - Start somewhere
  - Repeat: Take a step in the gradient direction
What is the Steepest Direction?*

\[
\max_{\Delta: \Delta_1^2 + \Delta_2^2 \leq \varepsilon} g(w + \Delta)
\]

- First-Order Taylor Expansion:
  \[
g(w + \Delta) \approx g(w) + \frac{\partial g}{\partial w_1} \Delta_1 + \frac{\partial g}{\partial w_2} \Delta_2
\]

- Steepest Descent Direction:
  \[
  \max_{\Delta: \Delta_1^2 + \Delta_2^2 \leq \varepsilon} g(w) + \frac{\partial g}{\partial w_1} \Delta_1 + \frac{\partial g}{\partial w_2} \Delta_2
  \]

- Recall:
  \[
  \max_{\Delta: \|\Delta\| \leq \varepsilon} \Delta^T a \rightarrow \Delta = \varepsilon \frac{a}{\|a\|}
  \]

- Hence, solution:
  \[
  \Delta = \varepsilon \frac{\nabla g}{\|\nabla g\|}
  \]

  **Gradient direction = steepest direction!**

  \[
  \nabla g = \begin{bmatrix}
  \frac{\partial g}{\partial w_1} \\
  \frac{\partial g}{\partial w_2}
  \end{bmatrix}
  \]
Gradient in n dimensions

\[ \nabla g = \begin{bmatrix} \frac{\partial g}{\partial w_1} \\
\frac{\partial g}{\partial w_2} \\
\vdots \\
\frac{\partial g}{\partial w_n} \end{bmatrix} \]
Optimization Procedure: Gradient Ascent

- init $w$
- for iter = 1, 2, ...

\[
w \leftarrow w + \alpha \star \nabla g(w)\]

- $\alpha$: learning rate --- tweaking parameter that needs to be chosen carefully
- How? Try multiple choices
  - Crude rule of thumb: update changes $w$ about 0.1 – 1%
Batch Gradient Ascent on the Log Likelihood Objective

$$\max_w ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

**init** $w$

**for** iter = 1, 2, ...

$$w \leftarrow w + \alpha \sum_i \nabla \log P(y^{(i)}|x^{(i)}; w)$$
Stochastic Gradient Ascent on the Log Likelihood Objective

$$\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)$$

**Observation:** once gradient on one training example has been computed, might as well incorporate before computing next one

- init $w$
- for iter = 1, 2, ...
  - pick random $j$

$$w \leftarrow w + \alpha \nabla \log P(y^{(j)}|x^{(j)}; w)$$
Mini-Batch Gradient Ascent on the Log Likelihood Objective

\[
\max_w \ ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w)
\]

**Observation:** gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

- **init** \(w\)
- **for** iter = 1, 2, ...
  - **pick** random subset of training examples \(J\)
  \[
  w \leftarrow w + \alpha \sum_{j \in J} \nabla \log P(y^{(j)}|x^{(j)}; w)
  \]
How about computing all the derivatives?

- We’ll talk about that once we covered neural networks, which are a generalization of logistic regression.
Neural Network Architectures

Brand new untested slides ahead – apologies for any typos/mistakes!
Manual Feature Design vs. Deep Learning

- Manual feature design requires:
  - Domain-specific expertise
  - Domain-specific effort

- What if we could learn the features, too?
  - Deep Learning
Review: Perceptron

\[ y = \begin{cases} 
1 & w_1 x_1 + w_2 x_2 + w_3 x_3 > 0 \\
0 & \text{otherwise} 
\end{cases} \]
Review: Perceptron with Sigmoid Activation

\[ y = \phi(w_1 x_1 + w_2 x_2 + w_3 x_3) \]

\[ = \frac{1}{1 + e^{-(w_1 x_1 + w_2 x_2 + w_3 x_3)}} \]
2-Layer, 2-Neuron Neural Network
2-Layer, 2-Neuron Neural Network

Intermediate output \( h_1 = \phi(w_{11}x_1 + w_{21}x_2 + w_{31}x_3) \)

\[
\frac{1}{1 + e^{-(w_{11}x_1 + w_{21}x_2 + w_{31}x_3)}}
\]
2-Layer, 2-Neuron Neural Network

intermediate output $h_2 = \phi(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)$

$$= \frac{1}{1 + e^{-(w_{12}x_1+w_{22}x_2+w_{32}x_3)}}$$
2-Layer, 2-Neuron Neural Network

\[ y = \phi(w_1 h_1 + w_2 h_2) \]

\[ = \frac{1}{1 + e^{-(w_1 h_1 + w_2 h_2)}} \]
2-Layer, 2-Neuron Neural Network

\[ y = \phi(w_1 h_1 + w_2 h_2) \]
\[ = \phi(w_1 \phi(w_{11} x_1 + w_{21} x_2 + w_{31} x_3) + w_2 \phi(w_{12} x_1 + w_{22} x_2 + w_{32} x_3)) \]
2-Layer, 2-Neuron Neural Network

\[ y = \phi(w_1h_1 + w_2h_2) \]
\[ = \phi(w_1\phi(w_{11}x_1 + w_{21}x_2 + w_{31}x_3) + w_2\phi(w_{12}x_1 + w_{22}x_2 + w_{32}x_3)) \]

The same equation, formatted with matrices:

\[
\phi \left( \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \right) 
= \phi \left( \begin{bmatrix} w_{11}x_1 + w_{21}x_2 + w_{31}x_3 \\ w_{12}x_1 + w_{22}x_2 + w_{32}x_3 \end{bmatrix} \right) 
= \begin{bmatrix} h_1 & h_2 \end{bmatrix}
\]

\[
\phi \left( \begin{bmatrix} h_1 & h_2 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \right) = \phi(w_1h_1 + w_2h_2) = y
\]

The same equation, formatted more compactly by introducing variables representing each matrix:

\[
\phi(x \times W_{\text{layer 1}}) = h \\
\phi(h \times W_{\text{layer 2}}) = y
\]
2-Layer, 2-Neuron Neural Network

\[ \phi(x \times W_{\text{layer 1}}) = h \]

- Shape (1, 3).
- Input feature vector.
- Shape (3, 2).
- Weights to be learned.
- Shape (1, 2).
- Outputs of layer 1, inputs to layer 2.

\[ \phi(h \times W_{\text{layer 2}}) = y \]

- Shape (1, 2).
- Outputs of layer 1, inputs to layer 2.
- Shape (2, 1).
- Weights to be learned.
- Shape (1, 1).
- Output of network.
2-Layer, 3-Neuron Neural Network

\[
\begin{align*}
    s &= w_{11}x_1 + w_{12}x_2 + w_{13}x_3, \\
    s &= w_{21}x_1 + w_{22}x_2 + w_{23}x_3, \\
    s &= w_{31}x_1 + w_{32}x_2 + w_{33}x_3, \\
    y &= w_1s_1 + w_2s_2 + w_3s_3.
\end{align*}
\]
2-Layer, 3-Neuron Neural Network

\[
\phi \left( \begin{bmatrix} x_1 & x_2 & x_3 \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \\ w_{31} & w_{32} & w_{33} \end{bmatrix} \right)
= \phi \left( \begin{bmatrix} w_{11}x_1 + w_{21}x_2 + w_{31}x_3 & w_{12}x_1 + w_{22}x_2 + w_{32}x_3 & w_{13}x_1 + w_{23}x_2 + w_{33}x_3 \end{bmatrix} \right)
= \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix}
\]

\[
\phi \left( \begin{bmatrix} h_1 & h_2 & h_3 \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \\ w_3 \end{bmatrix} \right) = \phi (w_1h_1 + w_2h_2 + w_3h_3) = y
\]
2-Layer, 3-Neuron Neural Network

\[ \phi(x \times W_{\text{layer 1}}) = h \]

- Shape (1, 3).
  - Input feature vector.

- Shape (3, 3).
  - Weights to be learned.

- Shape (1, 3).
  - Outputs of layer 1, inputs to layer 2.

\[ \phi(h \times W_{\text{layer 2}}) = y \]

- Shape (1, 3).
  - Outputs of layer 1, inputs to layer 2.

- Shape (3, 1).
  - Weights to be learned.

- Shape (1, 1).
  - Output of network.
Generalize: Number of hidden neurons

The hidden layer doesn’t necessarily need to have 3 neurons; it could have any arbitrary number $n$ neurons.
Generalize: Number of hidden neurons

The hidden layer doesn’t necessarily need to have 3 neurons; it could have any arbitrary number $n$ neurons.
The input feature vector doesn’t necessarily need to have 3 features; it could have some arbitrary number \( \dim(x) \) of features.
Generalize: Number of input features

The input feature vector doesn’t necessarily need to have 3 features; it could have some arbitrary number $dim(x)$ of features.
Generalize: Number of outputs

The output doesn’t necessarily need to be just one number; it could be some arbitrary $\text{dim}(y)$ length vector.
Generalize: Number of input features

The output doesn’t necessarily need to be just one number; it could be some arbitrary $\text{dim}(y)$ length vector.
Generalized 2-Layer Neural Network

\[ \phi(x \times W_{\text{layer 1}}) = h \]

- **Input feature vector.** Shape (1, \( dim(x) \)).
- **Weights to be learned.** Shape (\( dim(x) \), \( n \)).
- **Outputs of layer 1, inputs to layer 2.** Shape (1, \( n \)).

Layer 1 has weight matrix with shape (\( dim(x) \), \( n \)). These are the weights for \( n \) neurons, each taking \( dim(x) \) features as input.

This transforms a \( dim(x) \)-dimensional input vector into an \( n \)-dimensional output vector.

\[ \phi(h \times W_{\text{layer 2}}) = y \]

- **Outputs of layer 1, inputs to layer 2.** Shape (1, \( n \)).
- **Weights to be learned.** Shape (\( n \), \( dim(y) \)).
- **Output of network.** Shape (1, \( dim(y) \)).

Layer 2 has weight matrix with shape (\( n \), \( dim(y) \)). These are the weights for \( dim(y) \) neurons, each taking \( n \) features as input.

This transforms an \( n \)-dimensional input vector into a \( dim(y) \)-dimensional output vector.

**Big idea:** The shape of a weight matrix is determined by the dimensions of the input and output of that layer.
3-Layer, 3-Neuron Neural Network
3-Layer, 3-Neuron Neural Network

- **Layer 1:**
  - $x$ has shape $(1, 3)$. Input vector, 3-dimensional.
  - $W_{\text{layer 1}}$ has shape $(3, 3)$. Weights for 3 neurons, each taking in a 3-dimensional input vector.
  - $h_{\text{layer 1}}$ has shape $(1, 3)$. Outputs of the 3 neurons at this layer.

- **Layer 2:**
  - $h_{\text{layer 1}}$ has shape $(1, 3)$. Outputs of the 3 neurons from the previous layer.
  - $W_{\text{layer 2}}$ has shape $(3, 3)$. Weights for 3 new neurons, each taking in the 3 previous perceptron outputs.
  - $h_{\text{layer 2}}$ has shape $(1, 3)$. Outputs of the 3 new neurons at this layer.

- **Layer 3:**
  - $h_{\text{layer 2}}$ has shape $(1, 3)$. Outputs from the previous layer.
  - $W_{\text{layer 3}}$ has shape $(3, 1)$. Weights for 1 final neuron, taking in the 3 previous perceptron outputs.
  - $y$ has shape $(1, 1)$. Output of the final neuron.

\[
\phi(x \times W_{\text{layer 1}}) = h_{\text{layer 1}} \\
\phi(h_{\text{layer 1}} \times W_{\text{layer 2}}) = h_{\text{layer 2}} \\
\phi(h_{\text{layer 2}} \times W_{\text{layer 3}}) = y
\]
Generalized 3-Layer Neural Network

Layer 1:
- $x$ has shape $(1, \text{dim}(x))$
- $W_{\text{layer 1}}$ has shape $(\text{dim}(x), \text{dim}(L1))$
- $h_{\text{layer 1}}$ has shape $(1, \text{dim}(L1))$

Layer 2:
- $h_{\text{layer 1}}$ has shape $(1, \text{dim}(L1))$
- $W_{\text{layer 2}}$ has shape $(\text{dim}(L1), \text{dim}(L2))$
- $h_{\text{layer 2}}$ has shape $(1, \text{dim}(L2))$

Layer 3:
- $h_{\text{layer 2}}$ has shape $(1, \text{dim}(L2))$
- $W_{\text{layer 3}}$ has shape $(\text{dim}(L2), \text{dim}(y))$
- $y$ has shape $(1, \text{dim}(y))$

\[
\phi(x \times W_{\text{layer 1}}) = h_{\text{layer 1}} \\
\phi(h_{\text{layer 1}} \times W_{\text{layer 2}}) = h_{\text{layer 2}} \\
\phi(h_{\text{layer 2}} \times W_{\text{layer 3}}) = y
\]
Multi-Layer Neural Network

Note: Sometimes we don’t apply the non-linear function in the last layer.
Multi-Layer Neural Network

- Input to a layer: some $dim(x)$-dimensional input vector
- Output of a layer: some $dim(y)$-dimensional output vector
  - $dim(y)$ is the number of neurons in the layer (1 output per neuron)
- Process of converting input to output:
  - Multiply the (1, $dim(x)$) input vector with a ($dim(x)$, $dim(y)$) weight vector. The result has shape (1, $dim(y)$).
  - Apply some non-linear function (e.g. sigmoid) to the result. The result still has shape (1, $dim(y)$).
- Big idea: Chain layers together
  - The input could come from a previous layer’s output
  - The output could be used as the input to the next layer
Deep Neural Network

\[ z_i^{(k)} = g \left( \sum_j W_{i,j}^{(k-1,k)} z_j^{(k-1)} \right) \]

\( g = \text{nonlinear activation function} \)
Common Activation Functions

Sigmoid Function

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

Hyperbolic Tangent

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)

$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$
Batch Sizes

\[ \text{intermediate output } h_{11} \]
\[ \text{intermediate output } h_{21} \]
\[ \text{intermediate output } h_{12} \]
\[ \text{intermediate output } h_{22} \]
Batch Sizes

We’re not changing the architecture; we’re just running the 2-neuron, 2-layer network twice to classify 2 inputs.

\[ y_1 = \phi(w_1 h_{11} + w_2 h_{12}) \]
\[ = \phi(w_1 \phi(w_{11} x_{11} + w_{21} x_{12} + w_{31} x_{13}) + w_2 \phi(w_{12} x_{11} + w_{22} x_{12} + w_{32} x_{13})) \]
\[ y_2 = \phi(w_1 h_{21} + w_2 h_{22}) \]
\[ = \phi(w_1 \phi(w_{11} x_{21} + w_{21} x_{22} + w_{31} x_{23}) + w_2 \phi(w_{12} x_{21} + w_{22} x_{22} + w_{32} x_{23})) \]
Batch Sizes

\[ y_1 = \phi(w_1 h_{11} + w_2 h_{12}) \]
\[ = \phi(w_1 \phi(w_{11} x_{11} + w_{21} x_{12} + w_{31} x_{13}) + w_2 \phi(w_{12} x_{11} + w_{22} x_{12} + w_{32} x_{13})) \]
\[ y_2 = \phi(w_1 h_{21} + w_2 h_{22}) \]
\[ = \phi(w_1 \phi(w_{11} x_{21} + w_{21} x_{22} + w_{31} x_{23}) + w_2 \phi(w_{12} x_{21} + w_{22} x_{22} + w_{32} x_{23})) \]

Rewriting in matrix form:

\[
\phi \left( \begin{bmatrix} x_{11} & x_{21} & x_{31} \\ x_{12} & x_{22} & x_{32} \end{bmatrix} \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \\ w_{31} & w_{32} \end{bmatrix} \right) 
= \phi \left( \begin{bmatrix} w_{11} x_{11} + w_{21} x_{21} + w_{31} x_{31} & w_{12} x_{11} + w_{22} x_{21} + w_{32} x_{31} \\ w_{11} x_{12} + w_{21} x_{22} + w_{31} x_{32} & w_{12} x_{12} + w_{22} x_{22} + w_{32} x_{32} \end{bmatrix} \right) 
= \begin{bmatrix} h_{11} & h_{21} \\ h_{12} & h_{22} \end{bmatrix} 
\]

\[
\phi \left( \begin{bmatrix} h_{11} & h_{21} \\ h_{12} & h_{22} \end{bmatrix} \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \right) = \phi \left( \begin{bmatrix} w_1 h_{11} + w_2 h_{21} \\ w_1 h_{12} + w_2 h_{22} \end{bmatrix} \right) = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}
\]
Batch Sizes

$$\phi(x \times W_{\text{layer 1}}) = h$$

\begin{itemize}
  \item Shape \((batch, dim(x))\). Input feature vector.
  \item Shape \((dim(x), n)\). Weights to be learned.
  \item Shape \((batch, n)\). Outputs of layer 1, inputs to layer 2.
\end{itemize}

$$\phi(h \times W_{\text{layer 2}}) = y$$

\begin{itemize}
  \item Shape \((batch, n)\). Outputs of layer 1, inputs to layer 2.
  \item Shape \((n, dim(y))\). Weights to be learned.
  \item Shape \((batch, dim(y))\). Output of network.
\end{itemize}

Big idea: We can “stack” inputs together to classify multiple inputs at once. The result is multiple outputs “stacked” together.
Multi-Layer Network, with Batches

- Input to a layer: *batch* different $\text{dim}(x)$-dimensional input vectors
- Output of a layer: *batch* different $\text{dim}(y)$-dimensional output vectors
  - $\text{dim}(y)$ is the number of neurons in the layer (1 output per neuron)
- Process of converting input to output:
  - Multiply the ($\text{batch}$, $\text{dim}(x)$) input matrix with a ($\text{dim}(x)$, $\text{dim}(y)$) weight vector. The result has shape ($\text{batch}$, $\text{dim}(y)$).
  - Apply some non-linear function (e.g. sigmoid) to the result. The result still has shape ($\text{batch}$, $\text{dim}(y)$).
- Big idea: Stack inputs/outputs to batch them
  - The multiplication by weights and non-linear function will be applied to each row (data point in the batch) separately.
Training Neural Networks
Training Neural Networks

- **Step 1**: For each input in the training (sub)set $x$, predict a classification $y$ using the current weights

  $\phi(x \times W_{\text{layer 1}}) = h \quad \phi(h \times W_{\text{layer 2}}) = y$

- **Step 2**: Compare predictions with the true $y$ values, using a **loss function**
  - Higher value of loss function = bad model
  - Lower value of loss function = good model
  - Example: **zero-one loss**: count the number of misclassified inputs
  - Example: **log loss** (derived from maximum likelihood; more on this soon)
  - Example: **sum of squared errors** (if you’re solving a regression problem)

- **Step 3**: Use numerical method (e.g. gradient descent) to minimize loss
  - Loss is a function of the weights. Optimization goal: find weights that minimize loss
Log Loss Function

- Recall: loss function is a measure of how far off our model is
  - Higher value of loss function = bad model
  - Lower value of loss function = good model

- Log loss function for binary classification:

\[ \text{Log Loss} = - \sum_{i} y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \]

  - \( y_i \) = The true class of the \( i \)th data point in the training dataset
  - \( p_i \) = The probability of positive class, predicted by our classifier
  - Each data point contributes some loss. The total loss is the sum over all data points.

- Note: Log loss is a function of the weights
  - Changing the weights changes the predictions \( p_i \). The \( y_i \) do not change.
Log Loss Function

- Log loss function for a single data point: $-y \log(p) - (1 - y) \log(1 - p)$

- **Case I: True class is $y = 1$**
  - Log loss function becomes just $-\log(p)$
  - If classifier gives $p$ near 0:
    The model was confident in guessing $y=0$
    Bad model $\rightarrow$ high loss value
  - If classifier gives $p$ near 1:
    The model was confident in guessing $y=1$
    Good model $\rightarrow$ low loss value
Log Loss Function

- Log loss function for a single data point: \[-y \log(p) - (1 - y) \log(1 - p)\]

- Case II: True class is \( y = 0 \)
  - Log loss function becomes just \(-\log(1-p)\)
  - If classifier gives \( p \) near 0: The model was confident in guessing \( y=0 \) Good model \( \rightarrow \) low loss value
  - If classifier gives \( p \) near 1: The model was confident in guessing \( y=1 \) Bad model \( \rightarrow \) high loss value
  - Note: This equation uses \( y=0 \), not \( y=-1 \), for the negative class
Log Loss Example

\[ \text{Log Loss} = - \sum_i y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \]

- **Data point 1:** True class \( y=1 \). We predicted \( p=0.1 \) probability it’s the positive class.
  - Intuitively: Our guess is bad. We should have guessed closer to \( p=1 \).
  - This contributes \(-\log(0.1) = 1\) to the loss function.

- **Data point 2:** True class \( y=0 \). We predicted \( p=0 \) probability it’s the positive class.
  - Intuitively: Our guess was perfect. We were certain it was \( y=0 \).
  - This contributes \(-\log(1.0) = 0\) to our loss function.

- **Data point 3:** True class \( y=1 \). We predicted 0.8 probability it’s the positive class.
  - Intuitively: Our guess was pretty good, but not perfect.
  - This contributes \(-\log(0.8) = 0.1\) to the loss function.

- **Total loss:** \( 1 + 0 + 0.1 = 1.1 \)
  - Goal: Find the weights that lead to the probabilities that minimize this loss function.
Optimization Procedure: Gradient Descent

- init \( w \)
- for \( \text{iter} = 1, 2, \ldots \)
  \[
  w \leftarrow w - \alpha \nabla \text{Log Loss}(w)
  \]
  where
  \[
  \text{Log Loss} = - \sum_i y_i \log(p_i) + (1 - y_i) \log(1 - p_i)
  \]
  and \( p_i \) is computed by running the network on input \( x_i \) and weights \( w \)

- \( \alpha \): learning rate --- tweaking parameter that needs to be chosen carefully
How do we compute gradients of these loss functions?

Repeated application of the chain rule:

If \( f(x) = g(h(x)) \)

Then \( f'(x) = g'(h(x))h'(x) \)

→ Derivatives can be computed by following well-defined procedures.
Automatic Differentiation

- Automatic differentiation software
  - e.g. Theano, TensorFlow, PyTorch, Chainer
  - Only need to program the function \( g(x,y,w) \)
  - Can automatically compute all derivatives w.r.t. all entries in \( w \)
  - This is typically done by caching info during forward computation pass of \( f \), and then doing a backward pass = “backpropagation”
  - Autodiff / Backpropagation can often be done at computational cost comparable to the forward pass

- Need to know this exists
- How this is done? -- outside of scope of CS188
Backpropagation*

- Gradient of \( g(w_1, w_2, w_3) = w_1^4w_2 + 5w_3 \) at \( w_1 = 2, w_2 = 3, w_3 = 2 \)
- Think of \( g \) as a composition of many functions
  - Then, we can use the chain rule to compute the gradient

\[
\begin{align*}
\frac{\partial g}{\partial b} &= 1, \quad \frac{\partial g}{\partial c} = 1 \\
\frac{\partial g}{\partial a} &= \frac{\partial g}{\partial b} \frac{\partial b}{\partial a} = 1 \cdot w_2 = 3 \\
\frac{\partial g}{\partial w_2} &= \frac{\partial g}{\partial b} \frac{\partial b}{\partial w_2} = 1 \cdot a = 16 \\
\frac{\partial g}{\partial w_1} &= \frac{\partial g}{\partial a} \frac{\partial a}{\partial w_1} = 3 \cdot 4w_1^3 = 96 \\
\frac{\partial g}{\partial w_3} &= \frac{\partial g}{\partial c} \frac{\partial c}{\partial w_3} = 1 \cdot 5 = 5
\end{align*}
\]
Properties of Neural Networks
Neural Networks Properties

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

- Practical considerations
  - Can be seen as learning the features
  - Large number of neurons
    - Danger for overfitting
    - (hence early stopping!)
Universal Function Approximation Theorem*

**Hornik theorem 1:** Whenever the activation function is bounded and nonconstant, then, for any finite measure $\mu$, standard multilayer feedforward networks can approximate any function in $L^p(\mu)$ (the space of all functions on $\mathbb{R}^k$ such that $\int_{\mathbb{R}^k} |f(x)|^p d\mu(x) < \infty$) arbitrarily well, provided that sufficiently many hidden units are available.

**Hornik theorem 2:** Whenever the activation function is continuous, bounded and nonconstant, then, for arbitrary compact subsets $X \subseteq \mathbb{R}^k$, standard multilayer feedforward networks can approximate any continuous function on $X$ arbitrarily well with respect to uniform distance, provided that sufficiently many hidden units are available.

- **In words:** Given any continuous function $f(x)$, if a 2-layer neural network has enough hidden units, then there is a choice of weights that allow it to closely approximate $f(x)$.

Cybenko (1989) “Approximations by superpositions of sigmoidal functions”
Hornik (1991) “Approximation Capabilities of Multilayer Feedforward Networks”
Leshno and Schocken (1991) “Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function”
Universal Function Approximation Theorem*

G. Cybenko

Abstract. In this paper we demonstrate that finite linear combinations of compositions of a fixed, nonlinear function and a set of affine functions can uniformly approximate any continuous function of n variables with support in the unit n-cube, provided only mild conditions are imposed on the nonlinear function. Our results raise as open questions about representability in class of single hidden-layer neural networks. In particular, we show that arbitrary decision regions can be arbitrarily well approximated by continuous feedforward neural networks with only a single hidden-layer layer and any nonlinear sigmoidal activation function.

Hornik (1991) "Approximation Capabilities of Multilayer Feedforward Networks"

Kurt Hornik

Technische Universität Wien, Austria

1. INTRODUCTION

The approximation capabilities of neural network ar- chitectures have recently been investigated by many authors, including Geman and Hinton (1986), Cybenko (1989), Hornik (1989), Lin (1989), Leshno and Sacken (1991), and Leshno and Shoham (1992), among others. In this paper we study the approximation capabilities of a very specific class of neural networks, namely, feedforward networks with a single hidden layer.

The main result of this paper is a demonstration of the fact that some of the results of (3) are false when the model is extended to functions of more than one variable. In this case, however, the approximation capabilities of the network are not affected by the addition of extra variables.

Cybenko (1989) “Approximations by superpositions of sigmoidal functions”

Hornik (1991) “Approximation Capabilities of Multilayer Feedforward Networks”

Leshno and Shoham (1992) “Multilayer Feedforward Networks with Non-Polynomial Activation Functions Can Approximate Any Function”

* This work was supported in part by NSF Grant DCR-8901295, ONR Contract N00014-89-K-0312, and DOE Grant DE-FG-02-80ER1009, Centers for Supercomputing Research and Development and Department of Electrical and Computer Engineering, University of Illinois, Urbana, Illinois 61801, USA.

Cybenko (1989) “Approximations by superpositions of sigmoidal functions”
Summary of Key Ideas

- Optimize probability of label given input
  \[ \max_w ll(w) = \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w) \]

- Continuous optimization
  - Gradient ascent:
    - Compute steepest uphill direction = gradient (= just vector of partial derivatives)
    - Take step in the gradient direction
    - Repeat (until held-out data accuracy starts to drop = “early stopping”)

- Deep neural nets
  - Last layer = still logistic regression
  - Now also many more layers before this last layer
    - = computing the features
    - \( \rightarrow \) the features are learned rather than hand-designed
  - Universal function approximation theorem
    - If neural net is large enough
    - Then neural net can represent any continuous mapping from input to output with arbitrary accuracy
    - But remember: need to avoid overfitting / memorizing the training data \( \rightarrow \) early stopping!
  - Automatic differentiation gives the derivatives efficiently (how? = outside of scope of 188)
Application: Computer Vision
Performance

ImageNet Error Rate 2010-2014

Error Rate

2010 2011 2012 2013 2014

79% 60% 40% 20% 7%

Traditional CV

graph credit Matt Zeiler, Clarifai
Performance

ImageNet Error Rate 2010-2014

Error Rate

- 79%
- 60%
- 40%
- 20%
- 7%

2010 2011 2012 2013 2014

Traditional CV

graph credit Matt Zeiler, Clarifai
ImageNet Error Rate 2010-2014

Performance

AlexNet

graph credit Matt Zeiler, Clarifai
Performance

ImageNet Error Rate 2010-2014

Error Rate

- Traditional CV
- Deep Learning

AlexNet

Graph credit Matt Zeiler, Clarifai
Performance

ImageNet Error Rate 2010-2014

Graph credit: Matt Zeiler, Clarifai

AlexNet
Fun Neural Net Demo Site

- Demo-site:
  - http://playground.tensorflow.org/
Next: More Neural Net Applications!