CS 188: Artificial Intelligence

Neural Networks
Recall: Perceptron with Sigmoid Activation

\[ y = \phi(w_1x_1 + w_2x_2 + w_3x_3) \]

\[ = \frac{1}{1 + e^{- (w_1x_1 + w_2x_2 + w_3x_3)}} \]
Recall: 2-Layer, 2-Neuron Neural Network
Recall: 2-Layer, 2-Neuron Neural Network

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

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

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

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

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

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

$$y = \phi(w_1 h_1 + w_2 h_2)$$

$$= \phi(w_1 \phi(w_{11} x_1 + w_{21} x_2 + w_{31} x_3) + w_2 \phi(w_{12} x_1 + w_{22} x_2 + w_{32} x_3))$$
Recall: 2-Layer, 2-Neuron Neural Network

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

\[ \phi(h \times W_{\text{layer } 2}) = y \]
Recall: generalize number of hidden neurons

The hidden layer doesn’t necessarily need to have 3 neurons; it could have any arbitrary number $n$ neurons.
Recall: generalize number of input features

The input feature vector doesn’t necessarily need to have 3 features; it could have some arbitrary number \( \text{dim}(x) \) of features.
Recall: 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.
Recall: generalize number of layers

Note: Sometimes we don’t apply the non-linear function in the last layer.
Deep Neural Network for 3-way classification

\[ h_i^{(\text{layer } l)} = \phi \left( \sum_j w_{ji}^{(\text{layer } l)} \cdot h_j^{(\text{layer } l-1)} \right) \]

\( \phi = \text{nonlinear activation function} \)

- Neural network with \( L \) layers
- \( h^{(l)} \): activations at layer \( l \)
- \( w^{(l)} \): weights taking activations from layer \( l-1 \) to layer \( l \)
Recall: 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)

\[
g(z) = \max(0, z)
\]

\[
g'(z) = \begin{cases} 
1, & z > 0 \\
0, & \text{otherwise}
\end{cases}
\]

Recall: Sizes of neural networks

We have a neural network with the matrices drawn.

1. How many layers are in the network? 2
2. How many input dimensions $\dim(x)$? 3
3. How many hidden neurons $n$? 2
4. How many output dimensions $\dim(y)$? 1
5. What is the batch size? 4
Training Neural Networks
Recall: Deep Neural Network Training

Training the deep neural network is just like logistic regression:

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

just \(w\) tends to be a much, much larger vector

\[\rightarrow\] just run gradient ascent
+ stop when log likelihood of hold-out data starts to decrease
Batch Gradient Ascent on the Log Likelihood Objective

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

\[g(w)\]

**init** \(w\)

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

\[w \leftarrow w + \alpha \sum_i \nabla \log P(y^{(i)}|x^{(i)}; w)\]
How about computing all the derivatives?

Derivatives tables:

<table>
<thead>
<tr>
<th>Derivative</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \frac{d}{dx} (a) )</td>
<td>0</td>
</tr>
<tr>
<td>( \frac{d}{dx} (x) )</td>
<td>1</td>
</tr>
<tr>
<td>( \frac{d}{dx} (ax) )</td>
<td>( a \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx} (u + v - w) )</td>
<td>( \frac{du}{dx} + \frac{dv}{dx} - \frac{dw}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx} (uv) )</td>
<td>( u \frac{dv}{dx} + v \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx} \left( \frac{u}{v} \right) )</td>
<td>( \frac{\frac{dv}{dx} \cdot u - v \cdot \frac{du}{dx}}{v^2} )</td>
</tr>
<tr>
<td>( \frac{d}{dx} (u^n) )</td>
<td>( nu^{n-1} \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx} (\sqrt{u}) )</td>
<td>( \frac{1}{2\sqrt{u}} \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx} \left( \frac{1}{u} \right) )</td>
<td>( -\frac{1}{u^2} \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx} \left( \frac{1}{u^n} \right) )</td>
<td>( -\frac{n}{u^{n+1}} \frac{du}{dx} )</td>
</tr>
<tr>
<td>( \frac{d}{dx} (f(u)) )</td>
<td>( \frac{du}{dx} \left[ f(u) \right] )</td>
</tr>
</tbody>
</table>
How about computing all the derivatives?

- But neural net $f$ is never one of those?
  - No problem: CHAIN RULE:

$$f(x) = g(h(x))$$

Then

$$f'(x) = g'(h(x))h'(x)$$

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

Automatic differentiation software
  e.g. TensorFlow, PyTorch, Jax
  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? Details outside of scope of CS188, but we’ll show a basic example
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

- \( g = b + c \)
  \[ \frac{\partial g}{\partial b} = 1, \quad \frac{\partial g}{\partial c} = 1 \]

- \( b = a \times w_2 \)
  \[ \frac{\partial g}{\partial a} = \frac{\partial g}{\partial b} \frac{\partial b}{\partial a} = 1 \cdot w_2 = 3 \]
  \[ \frac{\partial g}{\partial w_2} = \frac{\partial g}{\partial b} \frac{\partial b}{\partial w_2} = 1 \cdot a = 16 \]

- \( a = w_1^4 \)
  \[ \frac{\partial g}{\partial w_1} = \frac{\partial g}{\partial a} \frac{\partial a}{\partial w_1} = 3 \cdot 4w_1^3 = 96 \]

- \( c = 5w_1 \)
  \[ \frac{\partial g}{\partial w_3} = \frac{\partial g}{\partial c} \frac{\partial c}{\partial w_3} = 1 \cdot 5 = 5 \]
Properties of Neural Networks
Theorem (Universal Function Approximators). A two-layer neural network with a sufficient number of neurons can approximate any continuous function to any desired accuracy.
Universal Function Approximation Theorem*  

- **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*

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”
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!)
Preventing Overfitting in Neural Networks

Early stopping:

Weight regularization
Weight Regularization

What can go wrong when we maximize log-likelihood?

Example: logistic regression with only one datapoint: \( f(x) = 1 \), \( y = +1 \)

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

\[\max \log \left( \frac{1}{1 + e^{-w}} \right)\]

\( w \) can grow very large and lead to overfitting and learning instability
What can go wrong when we maximize log-likelihood?

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

\(w\) can grow very large

Solution: add an objective term to penalize weight magnitude

\[
\max_w \sum_i \log P(y^{(i)} | x^{(i)} ; w) - \frac{\lambda}{2} \sum_j w_j^2
\]

\(\lambda\) is a hyperparameter (typically 0.1 to 0.0001 or smaller)
Preventing Overfitting in Neural Networks

Early stopping:

Weight regularization: \( \max_w \sum_i \log P(y^{(i)}|x^{(i)}; w) - \frac{\lambda}{2} \sum_j w_j^2 \)

Dropout
Consistency vs. Simplicity

- Example: curve fitting (regression, function approximation)

- Consistency vs. simplicity
- Ockham’s razor
Consistency vs. Simplicity

- Usually algorithms prefer consistency by default (why?)

- Several ways to operationalize “simplicity”
  - Reduce the hypothesis/model space
    - Assume more: e.g. independence assumptions, as in naïve Bayes
    - Fewer features or neurons
    - Other limits on model structure
  - Regularization
    - Laplace Smoothing: cautious use of small counts
    - Small weight vectors in neural networks (stay close to zero-mean prior)
    - Hypothesis space stays big, but harder to get to the outskirts
Fun Neural Net Demo Site

Demo-site:
http://playground.tensorflow.org/
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
  - 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? early stopping!

Automatic differentiation gives the derivatives efficiently (how? = outside of scope of 188)
Next: How well does deep learning work?