Plan for today:

- Neural Networks: Learning
- Neural Networks: Feature Learning
- Neural Networks: Deep Learning
Neural Network Learning
Learning Process

Model
= Architecture + Parameters

Things that can change
- Activation function
- Optimizer
- Hyperparameters
- ...

Loss

Gradients

Based on slide by Andrew Ng
Perceptron Learning Rule

\[ \theta \leftarrow \theta + \alpha(y - h(x))x \]

Equivalent to the intuitive rules:
- If output is correct, don’t change the weights
- If output is low \((h(x) = 0, y = 1)\), increment weights for all the inputs which are 1
- If output is high \((h(x) = 1, y = 0)\), decrement weights for all inputs which are 1

Perceptron Convergence Theorem:

• If there is a set of weights that is consistent with the training data (i.e., the data is linearly separable), the perceptron learning algorithm will converge [Minicksy & Papert, 1969]
Batch Perceptron

Given training data \( \{(x^{(i)}, y^{(i)})\}_{i=1}^{n} \)
Let \( \theta \leftarrow [0, 0, \ldots, 0] \)
Repeat:
   Let \( \Delta \leftarrow [0, 0, \ldots, 0] \)
   for \( i = 1 \ldots n \), do
      if \( y^{(i)} x^{(i)} \theta \leq 0 \)  // prediction for \( i^{th} \) instance is incorrect
         \( \Delta \leftarrow \Delta + y^{(i)} x^{(i)} \)
      \( \Delta \leftarrow \Delta / n \)  // compute average update
   \( \theta \leftarrow \theta + \alpha \Delta \)
Until \( ||\Delta||_2 < \epsilon \)

- Simplest case: \( \alpha = 1 \) and don’t normalize, yields the fixed increment perceptron
- Each increment of outer loop is called an epoch
Learning in NN: Backpropagation

• Similar to the perceptron learning algorithm, we cycle through our examples
  – If the output of the network is correct, no changes are made
  – If there is an error, weights are adjusted to reduce the error

• The trick is to assess the blame for the error and divide it among the contributing weights
Cost Function

Logistic Regression:

\[
J(\theta) = -\frac{1}{n} \sum_{i=1}^{n} [y_i \log h_\theta(x_i) + (1 - y_i) \log (1 - h_\theta(x_i))] + \frac{\lambda}{2n} \sum_{j=1}^{d} \theta_j^2
\]

Neural Network:

\[
h_\Theta \in \mathbb{R}^K \\
(h_\Theta(x))_i = i^{th} \text{output}
\]

\[
J(\Theta) = -\frac{1}{n} \left[ \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log (h_\Theta(x_i))_k + (1 - y_{ik}) \log \left(1 - (h_\Theta(x_i))_k\right) \right] + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{s_{l-1}=1}^{s_l} \sum_{s_l} \left(\Theta_{ji}^{(l)}\right)^2
\]

\[k^{th} \text{ class: } \text{true, predicted} \]
\[k^{th} \text{ class: } \text{true, predicted} \]

Based on slide by Andrew Ng
Optimizing the Neural Network

\[ J(\Theta) = -\frac{1}{n} \left[ \sum_{i=1}^{n} \sum_{k=1}^{K} y_{ik} \log(h_{\Theta}(x_i))_k + (1 - y_{ik}) \log\left(1 - (h_{\Theta}(x_i))_k\right) \right] \]

\[ + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} \left(\Theta^{(l)}_{ji}\right)^2 \]

Solve via: \[ \min_{\Theta} J(\Theta) \]

Need code to compute:

• \[ J(\Theta) \]

• \[ \frac{\partial}{\partial \Theta^{(l)}_{ij}} J(\Theta) \]

\( J(\Theta) \) is not convex, so GD on a neural net yields a local optimum
  • But, tends to work well in practice

Based on slide by Andrew Ng
Forward Propagation

- Given one labeled training instance \((x, y)\):

Forward Propagation

- \(a^{(1)} = x\)
- \(z^{(2)} = \Theta^{(1)} a^{(1)}\)
- \(a^{(2)} = g(z^{(2)}) \quad \text{[add } a_0^{(2)}]\)
- \(z^{(3)} = \Theta^{(2)} a^{(2)}\)
- \(a^{(3)} = g(z^{(3)}) \quad \text{[add } a_0^{(3)}]\)
- \(z^{(4)} = \Theta^{(3)} a^{(3)}\)
- \(a^{(4)} = h_\Theta(x) = g(z^{(4)})\)
Backpropagation Intuition

• Each hidden node $j$ is “responsible” for some fraction of the error $\delta_j^{(l)}$ in each of the output nodes to which it connects.

• $\delta_j^{(l)}$ is divided according to the strength of the connection between hidden node and the output node.

• Then, the “blame” is propagated back to provide the error values for the hidden layer.
Backpropagation Intuition

\[ \delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l \]

Formally,
\[ \delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(x_i) \]

where \( \text{cost}(x_i) = y_i \log h_\Theta(x_i) + (1 - y_i) \log(1 - h_\Theta(x_i)) \)
Backpropagation Intuition

$\delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l$

Formally, $\delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(x_i)$

where $\text{cost}(x_i) = y_i \log h_\Theta(x_i) + (1 - y_i) \log(1 - h_\Theta(x_i))$

Based on slide by Andrew Ng
Backpropagation Intuition

\[ \delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l \]

Formally, \( \delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(\mathbf{x}_i) \)

where \( \text{cost}(\mathbf{x}_i) = y_i \log h_\Theta(\mathbf{x}_i) + (1 - y_i) \log(1 - h_\Theta(\mathbf{x}_i)) \)
Backpropagation Intuition

\[ \delta_j^{(l)} = \text{“error” of node } j \text{ in layer } l \]

Formally, \[ \delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(x_i) \]

where \[ \text{cost}(x_i) = y_i \log h_{\Theta}(x_i) + (1 - y_i) \log(1 - h_{\Theta}(x_i)) \]
Backpropagation Intuition

\[ \delta_j^{(l)} = \text{"error" of node } j \text{ in layer } l \]

Formally, \[ \delta_j^{(l)} = \frac{\partial}{\partial z_j^{(l)}} \text{cost}(x_i) \]

where \[ \text{cost}(x_i) = y_i \log h_\Theta(x_i) + (1 - y_i) \log(1 - h_\Theta(x_i)) \]
Backpropagation: Gradient Computation

Let \( \delta_j^{(l)} = \) “error” of node \( j \) in layer \( l \)

(\#layers \( L = 4 \))

Backpropagation

- \( \delta^{(4)} = a^{(4)} - y \)
- \( \delta^{(3)} = (\Theta^{(3)})^T \delta^{(4)} \).
- \( \delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} \).
- (No \( \delta^{(1)} \))

\[
\frac{\partial}{\partial \Theta^{(l)}_{ij}} J(\Theta) = a_j^{(l)} \delta_i^{(l+1)} \]  

(ignoring \( \lambda \); if \( \lambda = 0 \))

Based on slide by Andrew Ng
**Backpropagation**

Set $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$

For each training instance $(x_i, y_i)$:

- Set $a^{(1)} = x_i$
- Compute $\{a^{(2)}, \ldots, a^{(L)}\}$ via forward propagation
- Compute $\delta^{(L)} = a^{(L)} - y_i$
- Compute errors $\{\delta^{(L-1)}, \ldots, \delta^{(2)}\}$
- Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient $D_{ij}^{(l)} = \begin{cases} 
\frac{1}{n} \Delta_{ij}^{(l)} + \lambda \Theta_{ij}^{(l)} & \text{if } j \neq 0 \\
\frac{1}{n} \Delta_{ij}^{(l)} & \text{otherwise}
\end{cases}$

$D^{(l)}$ is the matrix of partial derivatives of $J(\Theta)$

Note: Can vectorize $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$ as $\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} a^{(l)^T}$

Based on slide by Andrew Ng
Training a Neural Network via Gradient Descent with Backprop

Given: training set \{ (x_1, y_1), \ldots, (x_n, y_n) \}
Initialize all \( \Theta^{(l)} \) randomly (NOT to 0!)
Loop // each iteration is called an epoch

Set \( \Delta^{(l)}_{ij} = 0 \) \( \forall l, i, j \)

For each training instance \((x_i, y_i)\):
  - Set \( a^{(1)} = x_i \)
  - Compute \( \{a^{(2)}, \ldots, a^{(L)}\} \) via forward propagation
  - Compute \( \delta^{(L)} = a^{(L)} - y_i \)
  - Compute errors \( \{\delta^{(L-1)}, \ldots, \delta^{(2)}\} \)
  - Compute gradients \( \Delta^{(l)}_{ij} = \Delta^{(l)}_{ij} + a^{(l)}_{j} \delta^{(l+1)}_{i} \)

Compute avg regularized gradient \( D^{(l)}_{ij} = \begin{cases} \frac{1}{n} \Delta^{(l)}_{ij} + \lambda \Theta^{(l)}_{ij} & \text{if } j \neq 0 \\ \frac{1}{n} \Delta^{(l)}_{ij} & \text{otherwise} \end{cases} \)

Update weights via gradient step \( \Theta^{(l)}_{ij} = \Theta^{(l)}_{ij} - \alpha D^{(l)}_{ij} \)

Until weights converge or max #epochs is reached

Based on slide by Andrew Ng
Backprop Issues

“Backprop is the cockroach of machine learning. It’s ugly, and annoying, but you just can’t get rid of it.”

–Geoff Hinton

Problems:

• black box

• local minima
Implementation Details
Random Initialization

• Important to randomize initial weight matrices
• Can’t have uniform initial weights, as in logistic regression
  – Otherwise, all updates will be identical & the net won’t learn
Implementation Details

• For convenience, compress all parameters into $\Theta$
  – “unroll” $\Theta^{(1)}$, $\Theta^{(2)}$, ..., $\Theta^{(L-1)}$ into one long vector $\theta$
    • E.g., if $\Theta^{(1)}$ is 10 x 10, then the first 100 entries of $\theta$ contain the value in $\Theta^{(1)}$
  – Use the reshape command to recover the original matrices
    • E.g., if $\Theta^{(1)}$ is 10 x 10, then
      $\text{thetal1 }= \text{ reshape(theta[0:100], (10, 10))}$

• Each step, check to make sure that $J(\theta)$ decreases

• Implement a gradient-checking procedure to ensure that the gradient is correct...
Gradient Checking

**Idea:** estimate gradient numerically to verify implementation, then turn off gradient checking

\[
\frac{\partial}{\partial \theta_i} J(\theta) \approx \frac{J(\theta_{i+c}) - J(\theta_{i-c})}{2c}
\]

\[
\theta_{i+c} = [\theta_1, \theta_2, \ldots, \theta_{i-1}, \theta_i + c, \theta_{i+1}, \ldots]
\]

\[c \approx 1E-4\]

Change ONLY the \(i^{th}\) entry in \(\theta\), increasing (or decreasing) it by \(c\)

Based on slide by Andrew Ng
Gradient Checking

\( \theta \in \mathbb{R}^m \quad \theta \) is an “unrolled” version of \( \Theta^{(1)}, \Theta^{(2)}, \ldots \)

\( \theta = [\theta_1, \theta_2, \theta_3, \ldots, \theta_m] \)

Put in vector called \textit{gradApprox}

\[
\frac{\partial}{\partial \theta_1} J(\theta) \approx \frac{J([\theta_1 + c, \theta_2, \theta_3, \ldots, \theta_m]) - J([\theta_1 - c, \theta_2, \theta_3, \ldots, \theta_m])}{2c}
\]

\[
\frac{\partial}{\partial \theta_2} J(\theta) \approx \frac{J([\theta_1, \theta_2 + c, \theta_3, \ldots, \theta_m]) - J([\theta_1, \theta_2 - c, \theta_3, \ldots, \theta_m])}{2c}
\]

\vdots

\[
\frac{\partial}{\partial \theta_m} J(\theta) \approx \frac{J([\theta_1, \theta_2, \theta_3, \ldots, \theta_m + c]) - J([\theta_1, \theta_2, \theta_3, \ldots, \theta_m - c])}{2c}
\]

Check that the approximate numerical gradient matches the entries in the \( D \) matrices

Based on slide by Andrew Ng
Implementation Steps

- Implement backprop to compute $DVec$
  - $DVec$ is the unrolled $\{D^{(1)}, D^{(2)}, \ldots \}$ matrices

- Implement numerical gradient checking to compute $gradApprox$

- Make sure $DVec$ has similar values to $gradApprox$

- Turn off gradient checking. Using backprop code for learning.

**Important:** Be sure to disable your gradient checking code before training your classifier.

- If you run the numerical gradient computation on every iteration of gradient descent, your code will be very slow
Putting It All Together
Training a Neural Network

Pick a network architecture (connectivity pattern between nodes)

- # input units = # of features in dataset
- # output units = # classes

Reasonable default: 1 hidden layer
- or if >1 hidden layer, have same # hidden units in every layer (usually the more the better)
1. Randomly initialize weights

2. Implement forward propagation to get $h_\Theta(x_i)$ for any instance $x_i$

3. Implement code to compute cost function $J(\Theta)$

4. Implement backprop to compute partial derivatives

$$\frac{\partial}{\partial \Theta^{(l)}_{j,k}} J(\Theta)$$

5. Use gradient checking to compare $\frac{\partial}{\partial \Theta^{(l)}_{j,k}} J(\Theta)$ computed using backpropagation vs. the numerical gradient estimate.

   – Then, disable gradient checking code

6. Use gradient descent with backprop to fit the network
Activation Functions
Activation Functions

**Sigmoid**

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

\[ g'(z) = g(z)(1 - g(z)) \]

**Tanh**

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

\[ g'(z) = g(z)(1 - g(z)) \]

**Rectified Linear Unit (ReLU)**

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

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

**Leaky ReLU**

\[ g(z) = max(0.1z, z) \]

\[ g'(z) = \begin{cases} 
0.1 & \text{if } z \leq 0 \\ 
1 & \text{if } z > 0 
\end{cases} \]
Exploding and Vanishing Gradients

- **Vanishing:** gradient go to zero - parameters not updated (e.g., CNN)
  - **Solution:** Use Relu or leaky Relu, Batch Normalization, ...
- **Exploding:** gradient get very large - cause saturation (e.g., RNN)
  - **Solution:** Use Gradient Clipping, ...

![Diagram of a neural network showing vanishing and exploding gradients.](image-url)
Feature Learning
Unsupervised Learning - Autoencoders

- An **autoencoder** is an unsupervised learning algorithm that set the target values equal to the inputs. i.e., it uses $y_i = x_i$

$$J(\Theta) = -\frac{1}{n} \sum_{i=1}^{n} \| h_\Theta(x_i) - x_i \|^2 + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} \theta_{ji}^{(l)}$$
Sparse Autoencoders

- Autoencoders require number of hidden units ($S_2$) to be small
- If we impose a **sparsity** on the hidden units, then autoencoder can discover structure in data even if $S_2$ is large
- Let $\hat{\rho}$ be the average activation of hidden unit $j$:

$$\hat{\rho} = \frac{1}{n} \sum_{i=1}^{n} a_j^{(2)}(x_i)$$

- We want to enforce that

$$\hat{\rho} = \rho$$

Where $\rho$ is a sparsity parameter, typically a small value (0.05)

$$J(\Theta) = -\frac{1}{n} \sum_{i=1}^{n} \| h_\Theta(x_i) - x_i \|^2 + \frac{\lambda}{2n} \sum_{l=1}^{L-1} \sum_{i=1}^{s_{l-1}} \sum_{j=1}^{s_l} \theta_{ji}^{(l)} + \beta \sum_{j=1}^{s_2} KL(\rho || \hat{\rho})$$
Sparse Autoencoders - Visualization

• What input $x$ would cause $a_i^{(1)}$ to be maximally activated?

• Pixel $x_j$ which maximally activates hidden unit $i$ is given by

$$x_j = \frac{\theta_{ij}^{(1)}}{\sqrt{\sum_{j=1}^{p} \left(\theta_{ij}^{(1)}\right)^2}}$$
Unsupervised pre-training + Fine-tuning
Deep Learning
Unsupervised pre-training + Fine-tuning
Deep Learning Architectures
(Neural Network Zoo)
More Deep Learning Architectures

Deep Convolutional Network

Deconvolutional Network

Deep Convolutional Inverse Graphics Network

Generative Adversarial Network

Liquid State Machine

Echo State Network

Kohonen Network

Deep Residual Network

Support Vector Machine

Neural Turing Machine
Today

- Backpropagation
- Autoencoders
- Sparse Autoencoders