Introduction to Deep Learning
MIT 6.S191

Alexander Amini
What is Deep Learning?

**Artificial Intelligence**
Any technique that enables computers to mimic human behavior

**Machine Learning**
Ability to learn without explicitly being programmed

**Deep Learning**
Learn underlying features in data using neural networks

What is Deep Learning?
A RTIFICIAL I NTELLIGENCE

Any technique that enables computers to mimic human behavior

MACHINE LEARNING

Ability to learn without explicitly being programmed

DEEP LEARNING

Learn underlying features in data using neural networks
Deep Learning Success: Vision

Image Recognition
Deep Learning Success: Vision

Detect pneumothorax in real X-Ray scans
Deep Learning Success: Audio

Music Generation

Deep Learning Success: Audio

Music Generation
Deep Learning Success

And so many more…
6.S191 Goals

- Fundamentals
- Practical Skills
- Advancements
- Community @ MIT

Knowledge, intuition, know-how, and community to do deep learning research and development.
Administrative Information

- Meets Mon Jan 29 – Fri Feb 2
- 10:30 am – 1:30pm
- Lecture + Lab Breakdown
- Graded P/D/F; 3 Units
- 1 Final Assignment
<table>
<thead>
<tr>
<th>Session</th>
<th>Part 1</th>
<th>Part 2</th>
<th>Lab</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction to Deep Learning [Slides]</td>
<td>Deep Sequence Modeling [Slides]</td>
<td>Intro to TensorFlow, Music Generation with RNNs [Code]</td>
</tr>
<tr>
<td></td>
<td>[Video]</td>
<td>[Video]</td>
<td>coming soon!</td>
</tr>
<tr>
<td></td>
<td>[Video]</td>
<td>[Video]</td>
<td>coming soon!</td>
</tr>
<tr>
<td></td>
<td>[Video]</td>
<td>[Video]</td>
<td>coming soon!</td>
</tr>
<tr>
<td></td>
<td>[Slides] [Video]</td>
<td>[Slides] [Video]</td>
<td>coming soon!</td>
</tr>
<tr>
<td></td>
<td>[Slides] [Video]</td>
<td>[Slides] [Video]</td>
<td>coming soon!</td>
</tr>
</tbody>
</table>
Final Class Project

Option 1: Proposal Presentation
- Groups of 3 or 4
- Present a novel deep learning research idea or application
- 1 slide, 1 minute (strict)
- List of example proposals on website: introtodeeplearning.com
- Presentations on Friday, Feb 2
- Submit groups by Wednesday 5pm to be eligible
- Submit slide by Thursday 9pm to be eligible

- Judged by a panel of industry judges
- Top winners are awarded:
  - 1x NVIDIA Titan Xp
    MSRP: $1200
  - 2x NVIDIA Titan TX2
    MSRP: $600
  - 3x Google Home
    MSRP: $300
Final Class Project

**Option 1: Proposal Presentation**
- Groups of 3 or 4
- Present a novel deep learning research idea or application
- 1 slide, 1 minute (strict)
- List of example proposals on website: introtodeeplearning.com
- Presentations on Friday, Feb 2
- Submit groups by Wednesday 5pm to be eligible
- Submit slide by Thursday 9pm to be eligible

**Option 2: Write a 1-page review of a deep learning paper**
- Suggested papers listed on website: introtodeeplearning.com
- Grade is based on clarity of writing and technical communication of main ideas
- Due Friday 10:30am (before lecture)
Class Support

• Piazza: https://piazza.com/class/iwmlwep2fnd5uu
• Course Website: http://introtodeeplearning.com
• Lecture slides: http://introtodeeplearning.com/#schedule
• Email us: introtodeeplearning-staff@mit.edu
• Office Hours by request
Course Staff

Alexander Amini
Lead Organizer

Ava Soleimany
Lead Organizer

Harini Suresh
Co-Chair

Lex Fridman
Co-Chair

Shraman

Yilun

Abhimanyu

Jose Javier

Alex

Nikhil

Faraaz

Tim

Bhaktipriya

introtodeeplearning-staff@mit.edu
Thanks to Sponsors!
Why Deep Learning and Why Now?
Why Deep Learning?

Hand engineered features are time consuming, brittle and not scalable in practice.

Can we learn the *underlying features* directly from data?

- **Low Level Features**
  - Lines & Edges

- **Mid Level Features**
  - Eyes & Nose & Ears

- **High Level Features**
  - Facial Structure
Why Now?

Neural Networks date back decades, so why the resurgence?

1. Big Data
   - Larger Datasets
   - Easier Collection & Storage

2. Hardware
   - Graphics Processing Units (GPUs)
   - Massively Parallelizable

3. Software
   - Improved Techniques
   - New Models
   - Toolboxes
The Perceptron
The structural building block of deep learning
The Perceptron: Forward Propagation

\[ \hat{y} = g \left( \sum_{i=1}^{m} x_i \theta_i \right) \]

- Inputs
- Weights
- Sum
- Non-Linearity
- Output
The Perceptron: Forward Propagation

\[ \hat{y} = g \left( \theta_0 + \sum_{i=1}^{m} x_i \theta_i \right) \]

Inputs: \( x_1, x_2, \ldots, x_m \)
Weights: \( \theta_0, \theta_1, \theta_2, \ldots, \theta_m \)
Sum: \( \Sigma \)
Non-Linearity: \( g \)
Output: \( \hat{y} \)
The Perceptron: Forward Propagation

\[
\hat{y} = g \left( \theta_0 + \sum_{i=1}^{m} x_i \theta_i \right)
\]

\[
\hat{y} = g ( \theta_0 + X^T \theta )
\]

where: \( X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \) and \( \theta = \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_m \end{bmatrix} \)
The Perceptron: Forward Propagation

\[ \hat{y} = g \left( \theta_0 + X^T \theta \right) \]

- Example: sigmoid function

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

Inputs  Weights  Sum  Non-Linearity  Output
Common Activation Functions

**Sigmoid Function**

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

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

- `tf.nn.sigmoid(z)`

**Hyperbolic Tangent**

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

\[ g'(z) = 1 - g(z)^2 \]

- `tf.nn.tanh(z)`

**Rectified Linear Unit (ReLU)**

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

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

- `tf.nn.relu(z)`

**NOTE:** All activation functions are non-linear
Importance of Activation Functions

The purpose of activation functions is to introduce non-linearities into the network.

What if we wanted to build a Neural Network to distinguish green vs red points?
Importance of Activation Functions

The purpose of activation functions is to introduce non-linearities into the network.

Linear Activation functions produce linear decisions no matter the network size.
Importance of Activation Functions

The purpose of activation functions is to introduce non-linearities into the network.

Linear Activation functions produce linear decisions no matter the network size.

Non-linearities allow us to approximate arbitrarily complex functions.
The Perceptron: Example

We have: $\theta_0 = 1$ and $\theta = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g \left( \theta_0 + X^T \theta \right)$$
$$\hat{y} = g \left( 1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix} \right)$$
$$\hat{y} = g \left( 1 + 3x_1 - 2x_2 \right)$$

This is just a line in 2D!
The Perceptron: Example

\[ \hat{y} = g(1 + 3x_1 - 2x_2) \]
The Perceptron: Example

Assume we have input: \( X = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \)

\[
\hat{y} = g \left( 1 + 3x_1 - 2x_2 \right)
\]

\[
= g(-6) \approx 0.002
\]
The Perceptron: Example

\[ \hat{y} = g(1 + 3x_1 - 2x_2) \]

\[ z < 0 \]
\[ y < 0.5 \]

\[ z > 0 \]
\[ y > 0.5 \]
Building Neural Networks with Perceptrons
The Perceptron: Simplified

Inputs  Weights  Sum  Non-Linearity  Output
The Perceptron: Simplified

\[ y = g(z) \]

\[ z = \theta_0 + \sum_{j=1}^{m} x_j \theta_j \]
Multi Output Perceptron

\[ z_i = \theta_{0,i} + \sum_{j=1}^{m} x_j \theta_{j,i} \]

\[ y_1 = g(z_1) \]

\[ y_2 = g(z_2) \]
Single Layer Neural Network

\[ z_i = \theta^{(1)}_{0_i} + \sum_{j=1}^{m} x_j \theta^{(1)}_{j_i} \]

\[ \hat{y}_i = \theta^{(2)}_{0_i} + \sum_{j=1}^{d_1} z_j \theta^{(2)}_{j_i} \]
Single Layer Neural Network

\[
z_2 = \theta_{0,2}^{(1)} + \sum_{j=1}^{m} x_j \theta_{j,2}^{(1)} \\
= \theta_{0,2}^{(1)} + x_1 \theta_{1,2}^{(1)} + x_2 \theta_{2,2}^{(1)} + x_m \theta_{m,2}^{(1)}
\]
Multi Output Perceptron

Inputs

$\mathbf{x}_1$
$\mathbf{x}_2$
$\mathbf{x}_m$

Hidden

$\mathbf{z}_1$
$\mathbf{z}_2$
$\mathbf{z}_3$
$\mathbf{z}_{d_1}$

Output

$\hat{\mathbf{y}}_1$
$\hat{\mathbf{y}}_2$
Deep Neural Network

Inputs

\[ x_1 \]
\[ x_2 \]
\[ x_m \]

Hidden

\[ z_{k,1} \]
\[ z_{k,2} \]
\[ z_{k,3} \]
\[ z_{k,d_k} \]

Output

\[ \hat{y}_1 \]
\[ \hat{y}_2 \]

\[ z_{k,i} = \theta_{0,i}^{(k)} + \sum_{j=1}^{d_{k-1}} g(z_{k-1,j}) \theta_{j,i}^{(k)} \]
Applying Neural Networks
Example Problem

Will I pass this class?

Let's start with a simple two feature model

\[ x_1 = \text{Number of lectures you attend} \]
\[ x_2 = \text{Hours spent on the final project} \]
Example Problem: Will I pass this class?

- $x_2 = \text{Hours spent on the final project}$
- $x_1 = \text{Number of lectures you attend}$

Legend:
- Green: Pass
- Red: Fail
Example Problem: Will I pass this class?

\[ x_2 = \text{Hours spent on the final project} \]

\[ x_1 = \text{Number of lectures you attend} \]

Legend:
- \( \text{Pass} \)
- \( \text{Fail} \)

\[ \begin{bmatrix} 4 \\ 5 \end{bmatrix} \]
Example Problem: Will I pass this class?

$x^{(1)} = [4, 5]$

Predicted: 0.1
Example Problem: Will I pass this class?

\[ x^{(1)} = [4, 5] \]

\[
\begin{align*}
    x_1 &
    \quad \rightarrow \quad z_1 \\
    x_2 &
    \quad \rightarrow \quad z_2 \\
    &
    \quad \rightarrow \quad z_3 \\
    &
    \quad \rightarrow \quad \hat{y}_1
\end{align*}
\]

Predicted: 0.1
Actual: 1
Quantifying Loss

The loss of our network measures the cost incurred from incorrect predictions.

The input vector $x^{(1)} = [4, 5]$.

$$L(f(x^{(i)}; \theta), y^{(i)})$$

Predicted: 0.1
Actual: 1
Empirical Loss

The empirical loss measures the total loss over our entire dataset.

\[
x = \begin{bmatrix} 4, & 5 \\ 2, & 1 \\ 5, & 8 \\ \vdots & \vdots \end{bmatrix}
\]

\[
f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix}, \quad y = \begin{bmatrix} 1 \\ \times \\ \times \\ \vdots \end{bmatrix}
\]

\[
J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; \theta), y^{(i)})
\]

Also known as:
- Objective function
- Cost function
- Empirical Risk
Binary Cross Entropy Loss

*Cross entropy loss* can be used with models that output a probability between 0 and 1.

$$ J(\theta) = \frac{1}{n} \sum_{i=1}^{n} y^{(i)} \log \left( f(x^{(i)}; \theta) \right) + (1 - y^{(i)}) \log \left( 1 - f(x^{(i)}; \theta) \right) $$

Sample calculation:

$$ x = \begin{bmatrix} 4, 5 \\ 2, 1 \\ 5, 8 \\ \vdots \end{bmatrix} $$

$$ x_1 \rightarrow z_1 $$

$$ x_2 \rightarrow z_2 $$

$$ z_1 \rightarrow \hat{y}_1 $$

$$ f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \\ \vdots \end{bmatrix} $$

$$ y = \begin{bmatrix} 1 \\ 0 \\ 1 \\ \vdots \end{bmatrix} $$

**Actual** Predicted **Actual** Predicted

$$ \text{loss} = \text{tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )} $$
Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers.

\[
x = \begin{bmatrix} 4, 5 \\ 2, 1 \\ 5, 8 \\ \vdots \end{bmatrix}
\]

\[
f(x) = \begin{bmatrix} 30 \\ 80 \\ 85 \\ \vdots \end{bmatrix}, \quad y = \begin{bmatrix} 90 \\ 20 \\ 95 \\ \vdots \end{bmatrix}
\]

\[
J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \left( y^{(i)} - f(x^{(i)}; \theta) \right)^2
\]

\[
\text{Actual} \quad \text{Predicted}
\]

\[
\text{loss} = \text{tf.reduce_mean}(\text{tf.square(tf.subtract(model.y, model.pred)))}
\]
Training Neural Networks
Loss Optimization

We want to find the network weights that achieve the lowest loss

\[ \boldsymbol{\theta}^* = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; \theta), y^{(i)}) \]

\[ \boldsymbol{\theta}^* = \arg\min_{\theta} J(\theta) \]
Loss Optimization

We want to find the network weights that achieve the lowest loss

\[ \theta^* = \arg\min_{\theta} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; \theta), y^{(i)}) \]

\[ \theta^* = \arg\min_{\theta} J(\theta) \]

Remember:

\[ \theta = \{\theta^{(0)}, \theta^{(1)}, \ldots\} \]
Loss Optimization

\[ \theta^* = \underset{\theta}{\text{argmin}} J(\theta) \]

Remember:

Our loss is a function of the network weights!
Loss Optimization

Randomly pick an initial \((\theta_0, \theta_1)\)
Loss Optimization

Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
Loss Optimization

Take small step in opposite direction of gradient

\[ J(\theta_0, \theta_1) \]
Gradient Descent

Repeat until convergence

\[ J(\theta_0, \theta_1) \]
Gradient Descent

Algorithm
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

weights = tf.random_normal(shape, stddev=sigma)

grads = tf.gradients(ys=loss, xs=weights)

weights_new = weights.assign(weights - lr * grads)
Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:

3. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta}$

4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$

5. Return weights

$\text{weights} = \text{tf.random_normal}(\text{shape}, \text{stddev} = \text{sigma})$

$\text{grads} = \text{tf.gradients}(\text{ys} = \text{loss}, \text{xs} = \text{weights})$

$\text{weights\_new} = \text{weights\_assign}(\text{weights} - \text{lr} \times \text{grads})$
Computing Gradients: Backpropagation

How does a small change in one weight (ex. $\theta_2$) affect the final loss $J(\theta)$?
Computing Gradients: Backpropagation

\[ \frac{\partial J(\theta)}{\partial \theta_2} = \]

Let's use the chain rule!
Computing Gradients: Backpropagation

\[
\frac{\partial J(\theta)}{\partial \theta_2} = \frac{\partial J(\theta)}{\partial \hat{y}} \ast \frac{\partial \hat{y}}{\partial \theta_2}
\]
Computing Gradients: Backpropagation

\[
\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial J(\theta)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial \theta_1}
\]

Apply chain rule!  
Apply chain rule!
Computing Gradients: Backpropagation

\[
\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial J(\theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial \theta_1}
\]
Computing Gradients: Backpropagation

\[
\frac{\partial J(\theta)}{\partial \theta_1} = \frac{\partial J(\theta)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial z_1} \cdot \frac{\partial z_1}{\partial \theta_1}
\]

*Repeat this for every weight in the network using gradients from later layers*
Neural Networks in Practice: Optimization
Training Neural Networks is Difficult

Loss Functions Can Be Difficult to Optimize

Remember:
Optimization through gradient descent

\[ \theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta} \]
Loss Functions Can Be Difficult to Optimize

Remember:
Optimization through gradient descent

\[ \theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta} \]

How can we set the learning rate?
Setting the Learning Rate

Small learning rate converges slowly and gets stuck in false local minima
Setting the Learning Rate

Large learning rates overshoot, become unstable and diverge

\[ J(\theta) \]

Initial guess
Setting the Learning Rate

*Stable learning rates* converge smoothly and avoid local minima.
How to deal with this?

Idea 1:

Try lots of different learning rates and see what works “just right”
How to deal with this?

Idea 1:
Try lots of different learning rates and see what works “just right”

Idea 2:
Do something smarter!
Design an adaptive learning rate that “adapts” to the landscape
Adaptive Learning Rates

- Learning rates are no longer fixed
- Can be made larger or smaller depending on:
  - how large gradient is
  - how fast learning is happening
  - size of particular weights
  - etc...
Adaptive Learning Rate Algorithms

- Momentum
- Adagrad
- Adadelta
- Adam
- RMSProp


```python
tf.train.MomentumOptimizer
tf.train.AdagradOptimizer
tf.train.AdadeltaOptimizer
tf.train.AdamOptimizer
tf.train.RMSPropOptimizer
```
Neural Networks in Practice: Mini-batches
Gradient Descent

Algorithm
1. Initialize weights randomly \( \sim \mathcal{N}(0, \sigma^2) \)
2. Loop until convergence:
   3. Compute gradient, \( \frac{\partial J(\theta)}{\partial \theta} \)
   4. Update weights, \( \theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta} \)
5. Return weights
Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
3. Compute gradient, $rac{\partial J(\theta)}{\partial \theta}$
4. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
5. Return weights

Can be very computational to compute!
Stochastic Gradient Descent

Algorithm
1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$
2. Loop until convergence:
   3. Pick single data point $i$
   4. Compute gradient, $\frac{\partial J_i(\theta)}{\partial \theta}$
   5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$
6. Return weights
Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:

3. Pick single data point $i$

4. Compute gradient, $\frac{\partial J_i(\theta)}{\partial \theta}$

5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$

6. Return weights

Easy to compute but very noisy (stochastic)!
Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:

3. Pick batch of $B$ data points

4. Compute gradient, $\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\theta)}{\partial \theta}$

5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$

6. Return weights
Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

2. Loop until convergence:

3. Pick batch of $B$ data points

4. Compute gradient,

$$\frac{\partial J(\theta)}{\partial \theta} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(\theta)}{\partial \theta}$$

5. Update weights, $\theta \leftarrow \theta - \eta \frac{\partial J(\theta)}{\partial \theta}$

6. Return weights

Fast to compute and a much better estimate of the true gradient!
Mini-batches while training

More accurate estimation of gradient
    Smoother convergence
    Allows for larger learning rates
Mini-batches while training

More accurate estimation of gradient
Smoother convergence
Allows for larger learning rates

Mini-batches lead to fast training!
Can parallelize computation + achieve significant speed increases on GPU’s
Neural Networks in Practice: Overfitting
The Problem of Overfitting

**Underfitting**
Model does not have capacity to fully learn the data

**Ideal fit**

**Overfitting**
Too complex, extra parameters, does not generalize well
Regularization

What is it?

*Technique that constrains our optimization problem to discourage complex models*
Regularization

What is it?
Technique that constrains our optimization problem to discourage complex models

Why do we need it?
Improve generalization of our model on unseen data
Regularization 1: Dropout

- During training, randomly set some activations to 0
Regularization 1: Dropout

- During training, randomly set some activations to 0
  - Typically ‘drop’ 50% of activations in layer
  - Forces network to not rely on any 1 node

```
tf.nn.dropout(hiddenLayer, p=0.5)
```
Regularization 1: Dropout

- During training, randomly set some activations to 0
  - Typically ‘drop’ 50% of activations in layer
  - Forces network to not rely on any 1 node

\[
\begin{align*}
&x_1 
&x_2 
&x_3 
&z_{1,1} 
&z_{1,2} 
&z_{1,3} 
&z_{1,4} 
&z_{2,1} 
&z_{2,2} 
&z_{2,3} 
&z_{2,4} 
&\hat{y}_1 
&\hat{y}_2
\end{align*}
\]

\[
tf.nn.dropout(hiddenLayer, p=0.5)
\]
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit
Regularization 2: Early Stopping

• Stop training before we have a chance to overfit
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit
Regularization 2: Early Stopping

• Stop training before we have a chance to overfit

![Diagram showing loss vs. training iterations with early stopping graph]
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit
Regularization 2: Early Stopping

- Stop training before we have a chance to overfit

Legend:
- Testing
- Training
Core Foundation Review

The Perceptron
- Structural building blocks
- Nonlinear activation functions

Neural Networks
- Stacking Perceptrons to form neural networks
- Optimization through backpropagation

Training in Practice
- Adaptive learning
- Batching
- Regularization
Questions?