Introduction to Deep Learning

Alexander Amini

MIT 6.S191

January 27, 2020
Hi everybody, and welcome to MIT 6.S191
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**
Extract patterns from data using neural networks

3 1 3 4 7 2 1 7 4 2 3 5
Lecture Schedule

- Intro to Deep Learning
  - Lecture 1: Slides [Video] coming soon!
- Deep Computer Vision
  - Lecture 3: Slides [Video] coming soon!
- Deep Reinforcement Learning
  - Lecture 5: Slides [Video] coming soon!
- Guest Lecture
  - Lecture 7: Slides [Video] coming soon!
- Neural Rendering
  - Lecture 9: Slides [Video] coming soon!
- Deep Sequence Modeling
  - Lecture 2: Slides [Video] coming soon!
- Deep Generative Modeling
  - Lecture 4: Slides [Video] coming soon!
- Limitations and New Frontiers
  - Lecture 6: Slides [Video] coming soon!
- Robot Learning
  - Lecture 8: Slides [Video] coming soon!
- ML for Scent
  - Lecture 10: Slides [Video] coming soon!
- Intro to Tensorflow; Music Generation
  - Lab Session 1: Code coming soon!
- De-biased Facial Recognition Systems
- Pixels-to-Control Learning
  - Lab Session 3: Code coming soon!
- Final Projects
  - Lab Session 4: Video coming soon!
- Final Projects and Awards Ceremony
  - Lab Session 5: Video coming soon!

- Mon Jan 27 – Fri Jan 31
- 1:00 pm – 4:00pm, 32-123
- Lecture + Lab Breakdown
- Graded P/D/F; 3 Units
- 1 Final Assignment
Final Class Project

**Option 1: Proposal Presentation**
- At least 1+ registered student to be prize eligible
- Present a novel deep learning research idea or application
- 3 minutes (strict)
- Presentations on **Friday, Jan 31**
- Submit groups by **Wednesday 11:59pm** to be eligible
- Submit slide by **Thursday 11:59pm** to be eligible
- Instructions: [shorturl.at/wxBK7](http://shorturl.at/wxBK7)

- Judged by a panel of judges
- Top winners are awarded:
  - 3x NVIDIA 2080 Ti ($4000)
  - 4x Google Home ($400)
  - 3x Display Monitors ($300)
  - 3x SSD 1TB ($200)
Final Class Project

**Option 1: Proposal Presentation**
- At least 1+ registered student to be prize eligible
- Present a novel deep learning research idea or application
- 3 minutes (strict)
- Presentations on **Friday, Jan 31**
- Submit groups by **Wednesday 11:59pm** to be eligible
- Submit slide by **Thursday 11:59pm** to be eligible
- Instructions: shorturl.at/wxBK7

**Option 2:** Write a 1-page review of a deep learning paper
- Grade is based on clarity of writing and technical communication of main ideas
- Due **Friday Jan 31 1:00pm** (before lecture) by email
Labs and Prizes

Lab 1: Music Generation
- Beats Headphones

Lab 2: Computer Vision
- 24" HD Display Monitor

Lab 3: Reinforcement Learning
- Quadcopter Drone
Class Support

- Piazza: http://piazza.com/mit/spring2020/6s191
  - Useful for discussing labs
- Course Website: http://introtodeeplearning.com
  - Lecture schedule
  - Slides and lecture recordings
  - Software labs
  - Grading policy
- Email us: introtodeeplearning-staff@mit.edu
- Office Hours by request
Course Staff

Alexander Amini
Lead Organizer

Ava Soleimany
Lead Organizer

Alana
Andy
Blake
Gilbert

Hunter
Jacob
Julia
Konstantin

Kristian
Monisha
Roshni
Shinjini

introtodeeplearning-staff@mit.edu
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

- 1952: Stochastic Gradient Descent
- 1958: Perceptron
  - Learnable Weights
- 1986: Backpropagation
  - Multi-Layer Perceptron
- 1995: Deep Convolutional NN
  - Digit Recognition

- ImageNet
- Wikipedia
- TensorFlow
The Perceptron
The structural building block of deep learning
The Perceptron: Forward Propagation

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

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

\[
y = g \left( w_0 + \sum_{i=1}^{m} x_i w_i \right)
\]

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

\[ \hat{y} = g \left( w_0 + \sum_{i=1}^{m} x_i w_i \right) \]

\[ \hat{y} = g \left( w_0 + x^T w \right) \]

where: \( X = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \) and \( W = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \)
The Perceptron: Forward Propagation

Activation Functions

\[ \hat{y} = g \left( w_0 + X^T W \right) \]

- Example: sigmoid function

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

Graph showing the sigmoid function with input values from -6 to 6.
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)

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

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

TensorFlow code blocks

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: \( w_0 = 1 \) and \( \mathbf{w} = \begin{bmatrix} 3 \\ -2 \end{bmatrix} \)

\[
\hat{y} = g \left( w_0 + \mathbf{x}^T \mathbf{w} \right) \\
= g \left( 1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix} \right) \\
= g \left( 1 + 3x_1 - 2x_2 \right)
\]

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: \( X = \begin{bmatrix} -1 \\ 2 \end{bmatrix} \)

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

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

\[
= g (-6) \approx 0.002
\]
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

\[ \hat{y} = g \left( w_0 + X^T W \right) \]
The Perceptron: Simplified

\[ y = g(z) \]

\[ z = w_0 + \sum_{j=1}^{m} x_j w_j \]
Multi Output Perceptron

Because all inputs are densely connected to all outputs, these layers are called Dense layers

\[ z_i = w_{0,i} + \sum_{j=1}^{m} x_j w_{j,i} \]
Dense layer from scratch

```python
class MyDenseLayer(tf.keras.layers.Layer):
    def __init__(self, input_dim, output_dim):
        super(MyDenseLayer, self).__init__()

        # Initialize weights and bias
        self.W = self.add_weight([input_dim, output_dim])
        self.b = self.add_weight([1, output_dim])

    def call(self, inputs):
        # Forward propagate the inputs
        z = tf.matmul(inputs, self.W) + self.b

        # Feed through a non-linear activation
        output = tf.math.sigmoid(z)

        return output
```
Multi Output Perceptron

Because all inputs are densely connected to all outputs, these layers are called **Dense** layers

\[
\begin{align*}
  y_1 &= g(z_1) \\
  y_2 &= g(z_2)
\end{align*}
\]

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

```python
import tensorflow as tf
layer = tf.keras.layers.Dense( units=2 )
```
Single Layer Neural Network

\[ z_i = w^{(1)}_{0,i} + \sum_{j=1}^{m} x_j w^{(1)}_{j,i} \]

\[ \hat{y}_i = g \left( w^{(2)}_{0,i} + \sum_{j=1}^{d_1} g(z_j) w^{(2)}_{j,i} \right) \]
Single Layer Neural Network

\[ z_2 = w_{0,2}^{(1)} + \sum_{j=1}^{m} x_j w_{j,2}^{(1)} \]

\[ = w_{0,2}^{(1)} + x_1 w_{1,2}^{(1)} + x_2 w_{2,2}^{(1)} + x_m w_{m,2}^{(1)} \]
Multi Output Perceptron

```python
import tensorflow as tf

model = tf.keras.Sequential([
    tf.keras.layers.Dense(n),
    tf.keras.layers.Dense(2)
])
```
Deep Neural Network

\[ z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)} \]
Deep Neural Network

\[ z_{k,i} = w_{0,i}^{(k)} + \sum_{j=1}^{n_{k-1}} g(z_{k-1,j}) w_{j,i}^{(k)} \]

```
import tensorflow as tf
model = tf.keras.Sequential([tf.keras.layers.Dense(n_1),
                            tf.keras.layers.Dense(n_2),
                            ...
                            tf.keras.layers.Dense(2)])
```
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 Circle: Pass
- Red Circle: 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:
- Green Circle: Pass
- Red Circle: Fail
Example Problem: Will I pass this class?

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

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

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

Predicted: 0.1
Actual: 1
Quantifying Loss

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

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

\[ L \left( f (x^{(i)}; W), y^{(i)} \right) \]

Predicted: 0.1
Actual: 1
Empirical Loss

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

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

\[x_1 \rightarrow z_1 \rightarrow \hat{y}_1\]

\[x_2 \rightarrow z_2 \rightarrow \hat{y}_1\]

\[x_2 \rightarrow z_3 \rightarrow \hat{y}_1\]

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

\[
y = \begin{bmatrix}
1 \\
0 \\
1 \\
\vdots
\end{bmatrix}
\]

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

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.

$$f(x) = \begin{bmatrix} 0.1 \\ 0.8 \\ 0.6 \end{bmatrix} \times \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}$$

$$y = \begin{bmatrix} \text{Actual} \\ \text{Predicted} \\ \text{Actual} \\ \text{Predicted} \end{bmatrix}$$

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

```python
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(y, predicted) )
```
Mean Squared Error Loss

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

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

Final Grades (percentage)
Training Neural Networks
Loss Optimization

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

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

\[ \mathbf{W}^* = \arg \min_{\mathbf{W}} J(\mathbf{W}) \]
Loss Optimization

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

\[
\mathbf{W}^* = \text{argmin}_W \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}(f(x^{(i)}; \mathbf{W}), y^{(i)})
\]

\[
\mathbf{W}^* = \text{argmin}_W J(\mathbf{W})
\]

Remember:
\[
\mathbf{W} = \{\mathbf{W}^{(0)}, \mathbf{W}^{(1)}, \ldots \}
\]
Loss Optimization

\[ W^* = \underset{W}{\operatorname{argmin}} J(W) \]

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

Randomly pick an initial $(w_0, w_1)$
Loss Optimization

Compute gradient, \( \frac{\partial J(W)}{\partial W} \)

\[ J(w_0, w_1) \]

\[ w_0 \quad w_1 \]
Loss Optimization

Take small step in opposite direction of gradient

$J(w_0, w_1)$
Gradient Descent

Repeat until convergence

$J(w_0, w_1)$

$w_0$  $w_1$
Gradient Descent

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

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

```python
import tensorflow as tf

weights = tf.Variable([tf.random.normal()])

while True:  # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```
Gradient Descent

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

```python
import tensorflow as tf

weights = tf.Variable([[tf.random.normal()]]

while True:  # loop forever
    with tf.GradientTape() as g:
        loss = compute_loss(weights)
        gradient = g.gradient(loss, weights)

    weights = weights - lr * gradient
```
Computing Gradients: Backpropagation

How does a small change in one weight (ex. $w_2$) affect the final loss $J(W)$?
Computing Gradients: Backpropagation

\[
\frac{\partial J(W)}{\partial w_2} = \text{Let's use the chain rule!}
\]
Computing Gradients: Backpropagation

\[
\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial w_2}
\]
Computing Gradients: Backpropagation

\[
\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \ast \frac{\partial \hat{y}}{\partial w_1}
\]

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

\[
\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_1} \times \frac{\partial z_1}{\partial w_1}
\]
Computing Gradients: Backpropagation

\[
\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} \times \frac{\partial \hat{y}}{\partial z_1} \times \frac{\partial z_1}{\partial w_1}
\]

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

"Visualizing the loss landscape of neural nets". Dec 2017.
Loss Functions Can Be Difficult to Optimize

Remember:
Optimization through gradient descent

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

Remember:
Optimization through gradient descent

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

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

Small learning rate converges slowly and gets stuck in false local minima

$J(W)$

Initial guess
Setting the Learning Rate

Large learning rates overshoot, become unstable and diverge

$J(W)$
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...
# Gradient Descent Algorithms

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>TF Implementation</th>
<th>Reference</th>
</tr>
</thead>
<tbody>
<tr>
<td>RMSProp</td>
<td><code>tf.keras.optimizers.RMSProp</code></td>
<td></td>
</tr>
</tbody>
</table>

Putting it all together

```python
import tensorflow as tf
model = tf.keras.Sequential([...])

# pick your favorite optimizer
optimizer = tf.keras.optimizer.SGD()

while True:  # loop forever
    # forward pass through the network
    prediction = model(x)

    with tf.GradientTape() as tape:
        # compute the loss
        loss = compute_loss(y, prediction)

    # update the weights using the gradient
    grads = tape.gradient(loss, model.trainable_variables)
    optimizer.apply_gradients(zip(grads, model.trainable_variables))
```
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, $\frac{\partial J(W)}{\partial W}$
4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
5. Return weights
Gradient Descent

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

Can be very computationally intensive 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, $\frac{\partial j_i(W)}{\partial w}$
5. Update weights, $W \leftarrow W - \eta \frac{\partial j(W)}{\partial w}$
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, $\frac{\partial J_i(W)}{\partial W}$
5. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
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, $\frac{\partial j(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}$
5. Update weights, $W \leftarrow W - \eta \frac{\partial j(W)}{\partial W}$
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, \[
   \frac{\partial j(W)}{\partial W} = \frac{1}{B} \sum_{k=1}^{B} \frac{\partial J_k(W)}{\partial W}
   \]
   5. Update weights, \(W \leftarrow W - \eta \frac{\partial j(W)}{\partial W}\)
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

\[ \text{tf.keras.layers.Dropout}(p=0.5) \]
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
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
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
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