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

---

6.8191 Introduction to Deep Learning
introtodeeplearning.com

1/29/18
Deep Learning Success: Vision

Image Recognition

<table>
<thead>
<tr>
<th>mite</th>
<th>container ship</th>
<th>motor scooter</th>
<th>leopard</th>
</tr>
</thead>
<tbody>
<tr>
<td>mite</td>
<td>lifeboat</td>
<td>motor scooter</td>
<td>leopard</td>
</tr>
<tr>
<td>black widow</td>
<td>amphibiann</td>
<td>go-kart</td>
<td>jaguar</td>
</tr>
<tr>
<td>cockroach</td>
<td>fireboat</td>
<td>moped</td>
<td>cheetah</td>
</tr>
<tr>
<td>tick</td>
<td>drilling platform</td>
<td>bumper car</td>
<td>snow leopard</td>
</tr>
<tr>
<td>starfish</td>
<td></td>
<td>golfcart</td>
<td>Egyptian cat</td>
</tr>
</tbody>
</table>
Deep Learning Success

And so many more…
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
Mid Level Features
High Level Features

Lines & Edges
Eyes & Nose & Ears
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

Why Now?
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**: \(x_1, x_2, \ldots, x_m\)
- **Weights**: \(\theta_1, \theta_2, \ldots, \theta_m\)
- **Sum**: \(\sum\)
- **Non-Linearity**: Output
- **Output**: \(\hat{y}\)
The Perceptron: Forward Propagation

The diagram illustrates the Perceptron model with inputs $x_1, x_2, \ldots, x_m$, weights $\theta_0, \theta_1, \theta_2, \ldots, \theta_m$, and a non-linear activation function $g$. The output $\hat{y}$ is calculated as

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

The linear combination of inputs is $\theta_0 + \sum_{i=1}^{m} x_i \theta_i$, and the output is passed through the non-linear activation function $g$. The diagram also shows the inputs and weights, and the sum and non-linearity leading to the output.
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

Activation Functions

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

- Example: sigmoid function

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

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

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 $\mathbf{\theta} = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

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

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

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

- If \( z < 0 \) then \( y < 0.5 \)
- If \( z > 0 \) then \( y > 0.5 \)
Building Neural Networks with Perceptrons
The Perceptron: Simplified

Inputs  Weights  Sum  Non-Linearity  Output
The Perceptron: Simplified

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

\[ y = g(z) \]
Multi Output Perceptron

\[
\begin{align*}
    z_1 &= \theta_{0,i} + \sum_{j=1}^{m} x_j \theta_{j,i} \\
    y_1 &= g(z_1) \\
    z_2 &= \theta_{0,i} + \sum_{j=1}^{m} x_j \theta_{j,i} \\
    y_2 &= g(z_2)
\end{align*}
\]
Single Layer Neural Network

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

\[ \hat{y}_i = \theta_{0,i}^{(2)} + \sum_{j=1}^{d_1} z_j \theta_{j,i}^{(2)} \]
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

\[ x_1, x_2, x_m \quad \rightarrow \quad z_1, z_2, z_3, z_{d1} \quad \rightarrow \quad \hat{y}_1, \hat{y}_2 \]

Inputs \quad Hidden \quad Output
Deep Neural Network

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

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

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

- $x_2 =$ Hours spent on the final project
- $x_1 =$ 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

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

\[ \mathcal{L}(f(x^{(i)}; \theta), y^{(i)}) \]

Predicted: 0.1
Actual: 1
Empirical Loss

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

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

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

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

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

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.

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

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

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

\[
\hat{y}_1
\]

\[
x_1
\]

\[
x_2
\]

\[
z_1
\]

\[
z_2
\]

\[
z_3
\]

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

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

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

Mean squared error loss is given by:

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

Where:
- \( y^{(i)} \) is the actual value.
- \( f(x^{(i)}; \theta) \) is the predicted value.
- \( n \) is the number of samples.

Example:

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

\[
f(x) = \begin{bmatrix}
30 & \times & 90 \\
80 & \times & 20 \\
85 & \checkmark & 95 \\
\vdots & & \vdots \\
\end{bmatrix}
\]

Actual Grades

Predicted Grades

Final Grades (percentage)

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

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

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

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

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

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

\[
\theta^* = \arg\min J(\theta)
\]

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

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

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

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

\[ J(\theta_0, \theta_1) \]
Loss Optimization

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

Take small step in opposite direction of gradient
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
Computing Gradients: Backpropagation

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

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

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

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

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

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

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

\[ J(\theta) \]

\[ \theta \]

Initial guess
Setting the Learning Rate

*Large learning rates* overshoot, become unstable and diverge

\[ J(\theta) \]
Setting the Learning Rate

*Stable learning rates* converge smoothly and avoid local minima

![Graph showing the relationship between J(\(\theta\)) and \(\theta\)](image-url)

- Initial guess
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
  - `tf.train.MomentumOptimizer`
- Adagrad
  - `tf.train.AdagradOptimizer`
- Adadelta
  - `tf.train.AdadeltaOptimizer`
- Adam
  - `tf.train.AdamOptimizer`
- RMSProp
  - `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, $\frac{\partial J(\theta)}{\partial \theta}$
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 \( \sim N(0, \sigma^2) \)
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
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

\[
\sum_{i=1}^{n} x_i W_i + b = y
\]
Questions?