Neural Networks

David S. Rosenberg

New York University

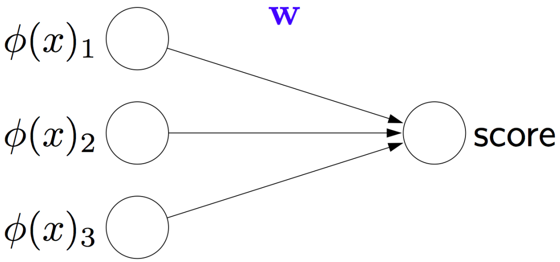
April 17, 2018

Contents

Neural Networks Overview

Linear Prediction Functions

- Linear prediction functions: SVM, ridge regression, Lasso
- Generate the feature vector $\phi(x)$ by hand.
- Learn parameter vector w from data.



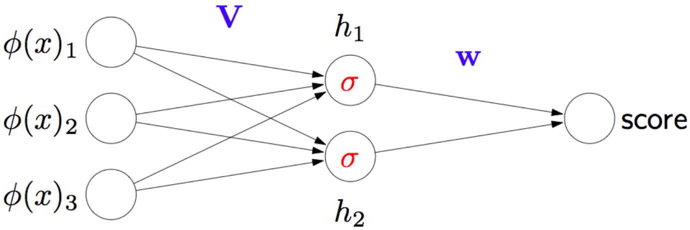
- So for $w \in \mathbf{R}^3$,

$$\text{score} = w^T \phi(x)$$

From Percy Liang's "Lecture 3" slides from Stanford's CS221, Autumn 2014.

Basic Neural Network (Multilayer Perceptron)

- Add an extra layer with **hidden nodes** h_1 and h_2:



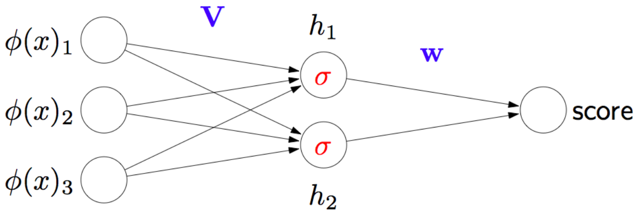
- For parameter vector $v_i \in \mathbf{R}^3$, define

$$h_i = \sigma\left(v_i^T \phi(x)\right),$$

where σ is a nonlinear **activation function**. (We'll come back to this.)

From Percy Liang's "Lecture 3" slides from Stanford's CS221, Autumn 2014.

Basic Neural Network (Multilayer Perceptron)



- For parameters $w_1, w_2 \in \mathbf{R}$, score is just

$$
\begin{aligned}
\text{score} &= w_1 h_1 + w_2 h_2 \\
&= w_1 \sigma(v_1^T \phi(x)) + w_2 \sigma\left(v_2^T \phi(x)\right)
\end{aligned}
$$

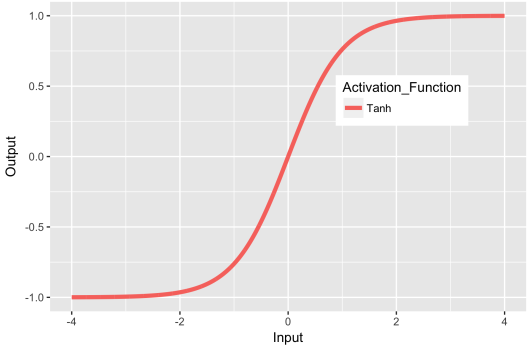
- This is the basic recipe.
    - We can add more hidden nodes.
    - We can add more hidden layers. (>1 hidden layer is a "deep network".)

From Percy Liang's "Lecture 3" slides from Stanford's CS221, Autumn 2014.

Activation Functions

- The **hyperbolic tangent** is a common activation function these days:

$$\sigma(x) = \tanh(x).$$

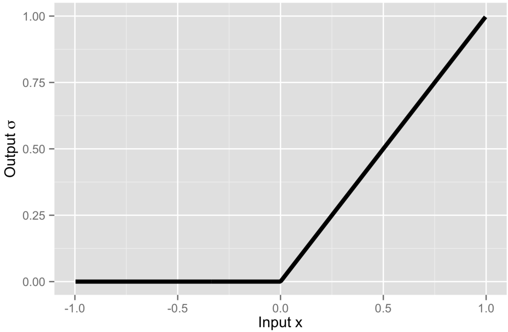


Activation Functions

- More recently, the **rectified linear** function has been very popular:

$$\sigma(x) = \max(0, x).$$

- "**ReLU**" is much faster to calculate, and to calculate its derivatives.
- Also often seems to work better.



Example: Regression with Multilayer Perceptrons (MLPs)

MLP Regression

- **Input space**: $\mathcal{X} = \mathbf{R}$
- **Action Space / Output space**: $\mathcal{A} = \mathcal{Y} = \mathbf{R}$
- **Hypothesis space**: MLPs with a single 3-node hidden layer:

$$f(x) = w_0 + w_1 h_1(x) + w_2 h_2(x) + w_3 h_3(x),$$

  where

$$h_i(x) = \sigma(v_i x + b_i) \text{ for } i = 1, 2, 3,$$

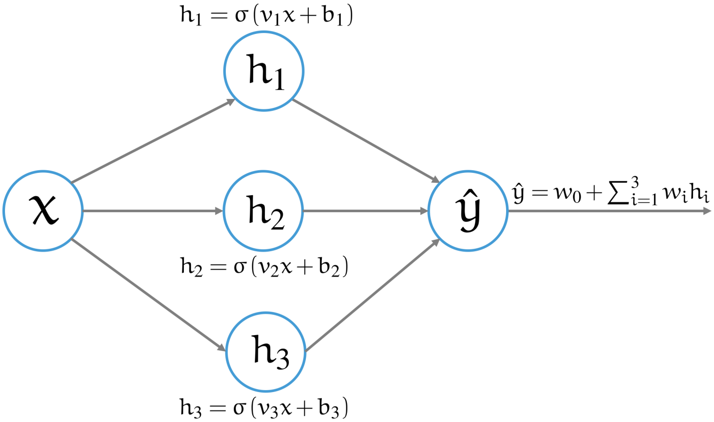
  for some fixed nonlinear "activation function" $\sigma : \mathbf{R} \to \mathbf{R}$.

- What are the parameters we need to fit?

$$b_1, b_2, b_3, v_1, v_2, v_3, w_0, w_1, w_2, w_3 \in \mathbf{R}$$

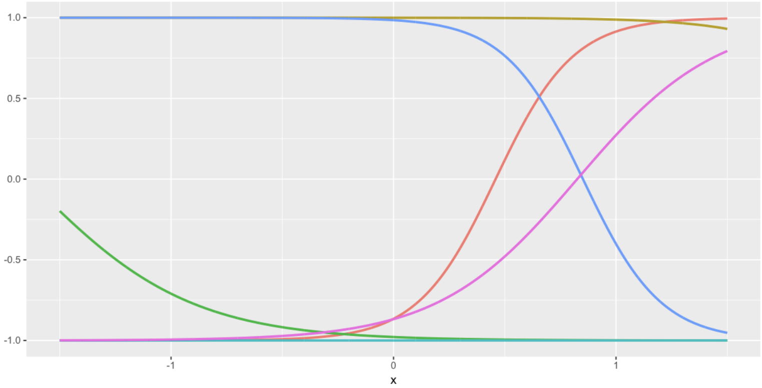
Multilayer Perceptron for $f : \mathbf{R} \to \mathbf{R}$

- MLP with one hidden layer; σ typically tanh or RELU; $x, h_1, h_2, h_3, \hat{y} \in \mathbf{R}$.



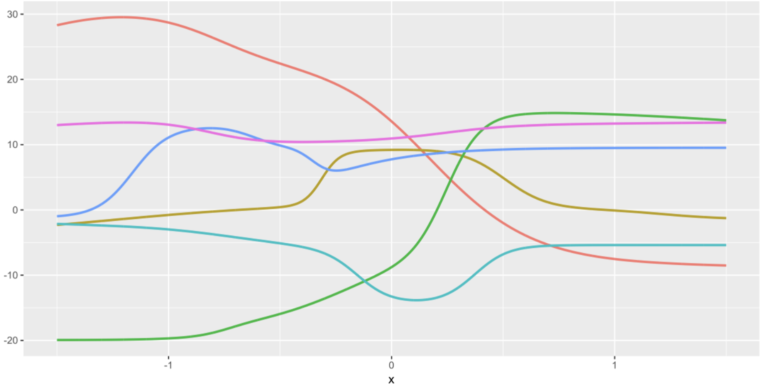
Hidden Layer as Feature/Basis Functions

- Can think of $h_i = h_i(x) = \sigma(v_i x + b_i)$ as a feature of x.
  - Learned by fitting the parameters v_i and b_i.
- Here are some $h_i(x)$'s for $\sigma = \tanh$ and randomly chosen v_i and b_i:



Samples from the Hypothesis Space

- Choosing 6 sets of random settings for $b_1, b_2, b_3, v_1, v_2, v_3, w_0, w_1, w_2, w_3 \in \mathbf{R}$, we get the following score functions:



How to choose the best hypothesis?

- As usual, choose our prediction function using empirical risk minimization.
- Our hypothesis space is parameterized by $\theta = (b_1, b_2, b_3, v_1, v_2, v_3, w_0, w_1, w_2, w_3) \in \Theta = \mathbf{R}^{10}$.
- For a training set $(x_1, y_1), \ldots, (x_n, y_n)$, find

$$\hat{\theta} = \operatorname*{arg\,min}_{\theta \in \mathbf{R}^{10}} \frac{1}{n} \sum_{i=1}^{n} (f_\theta(x_i) - y_i)^2 .$$

- Do we have the tools to find $\hat{\theta}$?
- Not quite, but close enough...

Gradient Methods for MLPs

- Note that

$$
\begin{aligned}
f(x) &= w_0 + \sum_{i=1}^{3} w_i h_i(x) \\
&= w_0 + \sum_{i=1}^{3} w_i \tanh(v_i x + b_i)
\end{aligned}
$$

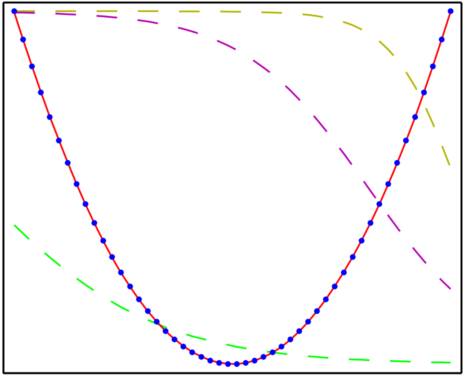
  is differentiable w.r.t. all parameters.

- We can use gradient-based methods as usual.
- However, the objective function is not convex w.r.t. parameters.
- So we can only hope to converge to a local minimum.
- In practice, this seems to be good enough.

Approximation Properties of Multilayer Perceptrons

Approximation Ability: $f(x) = x^2$

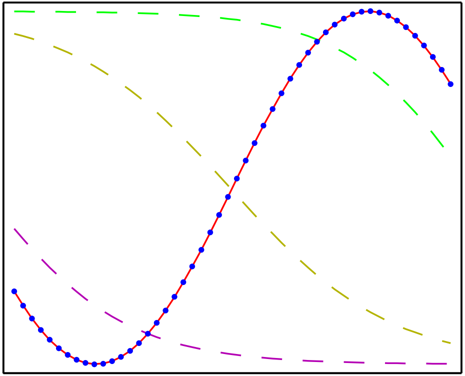
- 3 hidden units; tanh activation functions
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



From Bishop's *Pattern Recognition and Machine Learning*, Fig 5.3

Approximation Ability: $f(x) = \sin(x)$

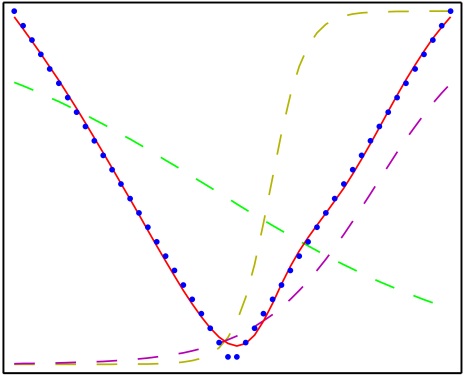
- 3 hidden units; logistic activation function
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



From Bishop's *Pattern Recognition and Machine Learning*, Fig 5.3

Approximation Ability: $f(x) = |x|$

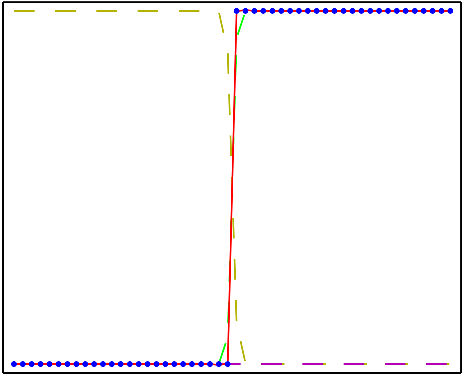
- 3 hidden units; logistic activation functions
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



From Bishop's *Pattern Recognition and Machine Learning*, Fig 5.3

Approximation Ability: $f(x) = 1(x > 0)$

- 3 hidden units; logistic activation function
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



From Bishop's *Pattern Recognition and Machine Learning*, Fig 5.3

Universal Approximation Theorems

- Leshno and Schocken (1991) showed:
  - A neural network with one [possibly huge] hidden layer can uniformly approximate any continuous function on a compact set iff the activation function is not a polynomial (i.e. tanh, logistic, and ReLU all work, as do sin,cos, exp, etc.).
- In more words:
  - Let $\varphi(\cdot)$ be any non-polynomial function (an activation function).
  - Let $f : K \to \mathbf{R}$ be any continuous function on a compact set $K \subset \mathbf{R}^m$.
  - Then $\forall \varepsilon > 0$, there exists an integer N (the number of hidden units), and parameters $v_i, b_i \in \mathbf{R}$ and $w_i \in \mathbf{R}^m$ such that the function

    $$F(x) = \sum_{i=1}^{N} v_i \varphi(w_i^T x + b_i)$$

    satisfies $|F(x) - f(x)| < \varepsilon$ for all $x \in K$.
- Leshno & Schocken note that this **doesn't work without the bias term** b_i (they call it the **threshold** term). (e.g. consider $\varphi = \sin$: then we always have $F(-x) = -F(x)$)

Review: Multinomial Logistic Regression

Recall: Multinomial Logistic Regression

- Setting: $\mathcal{X} = \mathbf{R}^d$, $\mathcal{Y} = \{1, \ldots, k\}$
- For each x, we want to produce a distribution on k classes.
- Such a distribution is called a "**multinoulli**" or "**categorical**" distribution.
- Represent categorical distribution by probability vector $\theta = (\theta_1, \ldots, \theta_k) \in \mathbf{R}^k$, where
  - $\sum_{y=1}^{k} \theta_y = 1$ and $\theta_y \geqslant 0$ for $y \in \{1, \ldots, k\}$.

Multinomial Logistic Regression

- From each x, we compute a linear score function for each class:

$$x \mapsto (\langle w_1, x \rangle, \ldots, \langle w_k, x \rangle) \in \mathbf{R}^k$$

- We need to map this $\mathbf{R}^k$ vector into a probability vector θ.
- The **softmax function** maps scores $s = (s_1, \ldots, s_k) \in \mathbf{R}^k$ to a categorical distribution:

$$(s_1, \ldots, s_k) \mapsto \theta = \textbf{Softmax}(s_1, \ldots, s_k) = \left(\frac{\exp(s_1)}{\sum_{i=1}^k \exp(s_i)}, \ldots, \frac{\exp(s_k)}{\sum_{i=1}^k \exp(s_i)} \right)$$

Multinomial Logistic Regression: Learning

- Let $y \in \{1, \ldots, k\}$ be an index denoting a class.
- Then predicted probability for class y given x is

$$\hat{p}(y \mid x) = \operatorname{Softmax}\left(\langle w_1, x\rangle, \ldots, \langle w_k, x\rangle\right)_y,$$

  where the y subscript indicates taking the y'th entry of the vector produced Softmax.
- **Learning**: Maximize the log-likelihood of training data

$$\underset{w_1,\ldots,w_k \in \mathbf{R}^d}{\arg\max} \sum_{i=1}^{n} \log\left[\operatorname{Softmax}\left(\langle w_1, x_i\rangle, \ldots, \langle w_k, x_i\rangle\right)_{y_i}\right].$$

- This objective function is concave in w's and straightforward to optimize.

Standard MLP for Multiclass

Nonlinear Generalization of Multinomial Logistic Regression

- **Key change**: Rather than k linear score functions

$$x \mapsto (\langle w_1, x \rangle, \ldots, \langle w_k, x \rangle) \in \mathbf{R}^k,$$

  use nonlinear score functions:

$$x \mapsto (f_1(x), \ldots, f_k(x)) \in \mathbf{R}^k,$$

- Then predicted probability for class $y \in \{1, \ldots, k\}$ given x is

$$\hat{p}(y \mid x) = \text{Softmax}(f_1(x), \ldots, f_k(x))_y .$$

Nonlinear Generalization of Multinomial Logistic Regression

- **Learning**: Maximize the log-likelihood of training data

$$\underset{f_1,\ldots,f_k}{\arg\max} \sum_{i=1}^{n} \log\left[\text{Softmax}\left(f_1(x),\ldots,f_k(x)\right)_{y_i}\right].$$

- We could use gradient boosting to get f_i's as ensembles of regression trees.
- Today we'll learn to use a multilayer perceptron for $f : \mathbf{R}^d \to \mathbf{R}^k$.
- Unfortunately, this objective function will not be concave or convex.
- But we can still use gradient methods to find a good local optimum.

Multilayer Perceptron: Standard Recipe

- **Input space**: $\mathcal{X} = \mathbf{R}^d$ **Action space** $\mathcal{A} = \mathbf{R}^k$ (for k-class classification).
- Let $\sigma : \mathbf{R} \to \mathbf{R}$ be a non-polynomial activation function (e.g. tanh or ReLU).
- Let's take all hidden layers to have m units.
- First hidden layer is given by

$$h^{(1)}(x) = \sigma\left(W^{(1)}x + b^{(1)}\right),$$

for parameters $W^{(1)} \in \mathbf{R}^{m \times d}$ and $b \in \mathbf{R}^m$, and where $\sigma(\cdot)$ is applied to each entry of its argument.

Multilayer Perceptron: Standard Recipe

- Each subsequent hidden layer takes the output $o \in \mathbf{R}^m$ of previous layer and produces

$$h^{(j)}(o) = \sigma\left(W^{(j)}o + b^{(j)}\right), \text{ for } j = 1, \ldots, D$$

  where $W^{(j)} \in \mathbf{R}^{m \times m}$, $b^{(j)} \in \mathbf{R}^m$, and D is the number of hidden layers.
- Last layer is an affine mapping:

$$a(o) = W^{(D+1)}o + b^{(D+1)},$$

  where $W^{(D+1)} \in \mathbf{R}^{k \times m}$ and $b^{(D+1)} \in \mathbf{R}^k$.

Multilayer Perceptron: Standard Recipe

- So the full neural network function is given by the composition of layers:

$$f(x) \quad = \quad \left(a \circ h^{(D)} \circ \cdots \circ h^{(1)}\right)(x)$$

- This gives us the k score functions we need.
- To train this we maximize the conditional log-likelihood for the training data:

$$J(\theta) = \frac{1}{n} \sum_{i=1}^{n} \log \left[\operatorname{Softmax}(f(x_i))_{y_i}\right],$$

where $\theta = \left(W^{(1)}, \ldots, W^{(D+1)}, b^{(1)}, \ldots, b^{(D+1)}\right)$.

Neural Networks for Features

OverFeat: Features

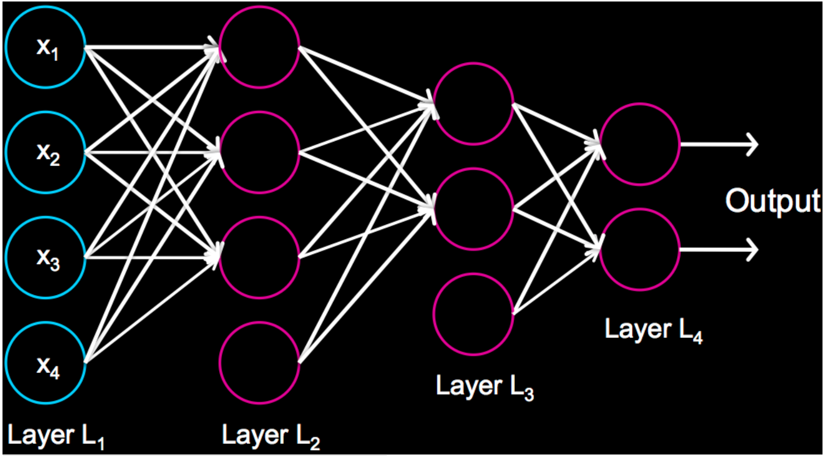
- OverFeat is a neural network for image classification
  - Trained on the huge ImageNet dataset
  - Lots of computing resources used for training the network.
- All those hidden layers of the network are very valuable **features**.
  - Paper: "*CNN Features off-the-shelf: an Astounding Baseline for Recognition*"
  - Showed that using features from OverFeat makes it easy to achieve state-of-the-art performance on new vision tasks.

OverFeat code is at https://github.com/sermanet/OverFeat

Multiple Output Networks

Multiple Output Neural Networks

- Very easy to add extra outputs to neural network structure.



From Andrew Ng's CS229 Deep Learning slides (http://cs229.stanford.edu/materials/CS229-DeepLearning.pdf)

Multitask Learning

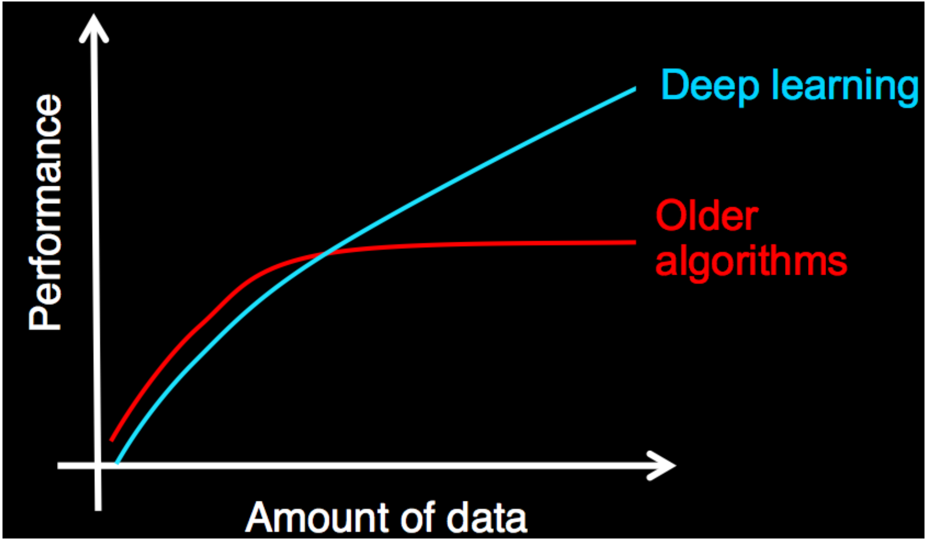
- Suppose $\mathcal{X} = \{\text{Natural Images}\}$.
- We have two tasks:
  - Does the image have a cat?
  - Does the image have a dog?
- Can have one output for each task.
- Seems plausible that basic pixel features would be shared by tasks.
- Learn them on the same neural network – benefit both tasks.
- Objective function must combine losses from both predictions, e.g. by averaging.

Single Task with "Extra Tasks"

- Only one task we're interested in.
- Gather data from related tasks.
- Train them along with the task you're interested in.
- No related tasks? Another trick:
    - Choose any input feature.
    - Change it's value to zero.
    - Make the prediction problem to predict the value of that feature.
    - Can help make model more robust (not depending too heavily on any single input).

Neural Networks: When and why?

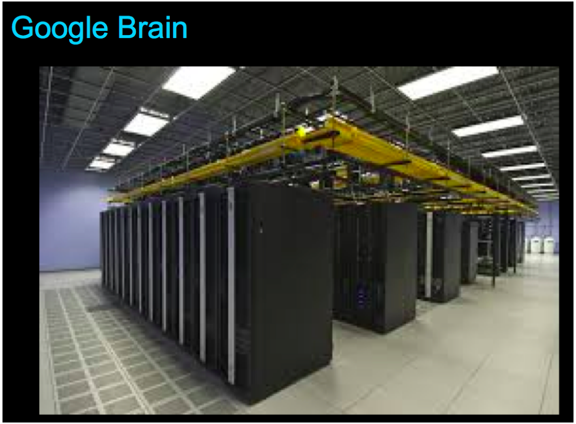
Neural Networks Benefit from Big Data



From Andrew Ng's CS229 Deep Learning slides (http://cs229.stanford.edu/materials/CS229-DeepLearning.pdf)

Big Data Requires Big Resources

- Best results always involve GPU processing.
- Often on large networks.



From Andrew Ng's CS229 Deep Learning slides (http://cs229.stanford.edu/materials/CS229-DeepLearning.pdf)