

# Neural Networks

---

David S. Rosenberg

Bloomberg ML EDU

December 19, 2017

# Neural Networks Overview

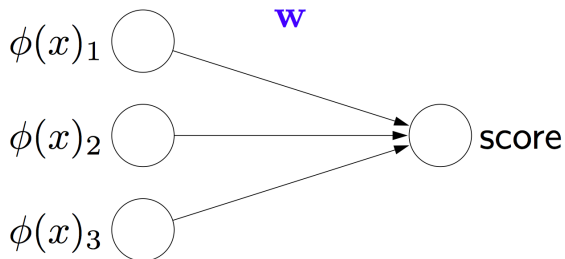
---

# Objectives

- What are neural networks?
- How do they fit into our toolbox?
- When should we consider using them?

# 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  $w \in \mathbf{R}^3$ ,

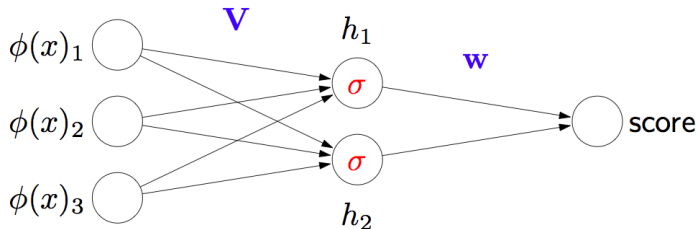
$$\text{score} = w^T \phi(x)$$

---

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  $h_2$ :

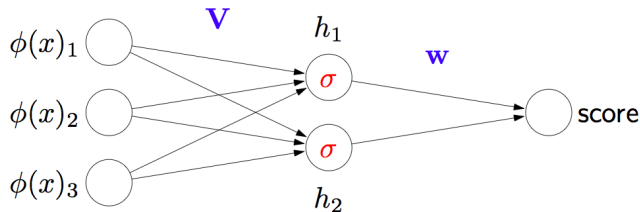


- For parameter vector  $v_i \in \mathbf{R}^3$ , define

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

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

# Basic Neural Network



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

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

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

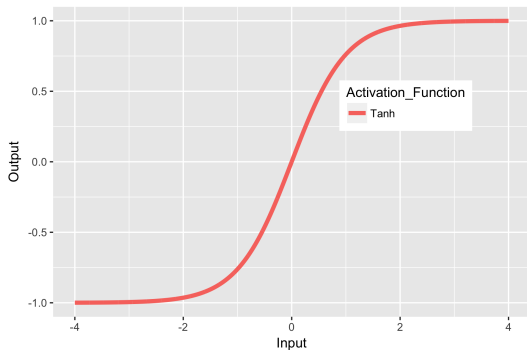
---

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:

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

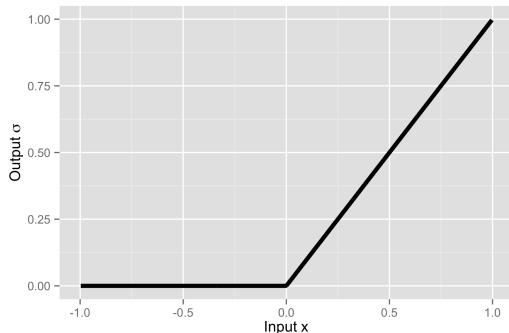


# Activation Functions

- 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)

---

- **Input space:**  $\mathcal{X} = \mathbb{R}$
- **Action Space / Output space:**  $\mathcal{A} = \mathcal{Y} = \mathbb{R}$
- **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

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

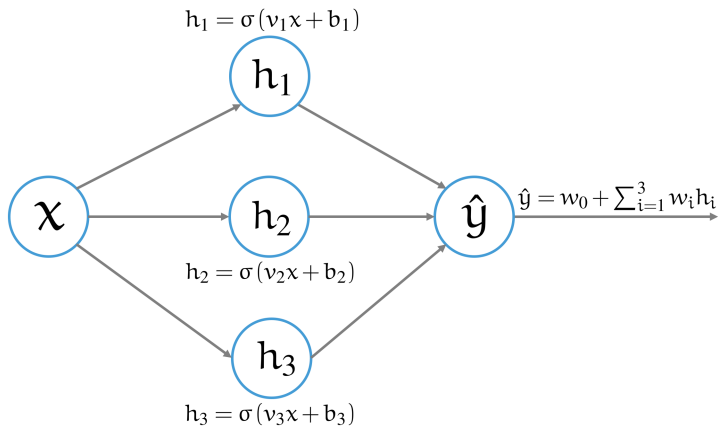
for some fixed nonlinear “activation function”  $\sigma: \mathbb{R} \rightarrow \mathbb{R}$ .

- What are the parameters we need to fit?

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

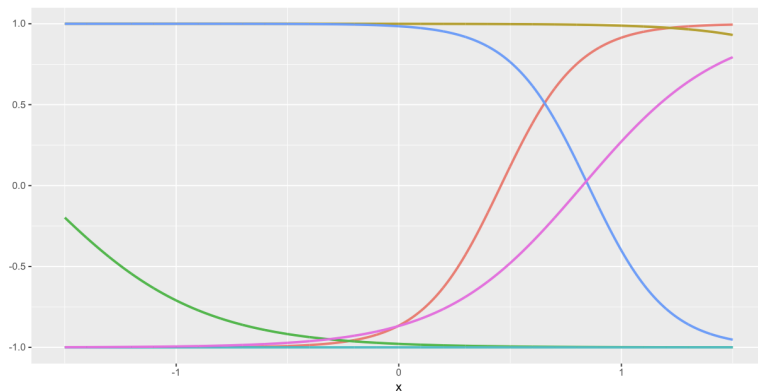
# Multilayer Perceptron for $f : \mathbf{R} \rightarrow \mathbf{R}$

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



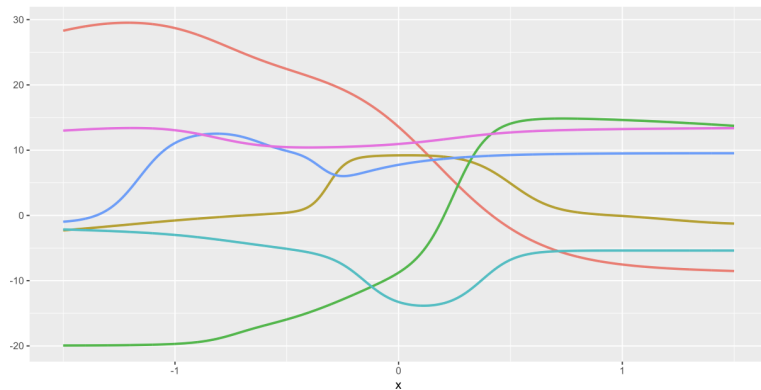
# Hidden Layer as Feature/Basis Functions

- 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  $(x_1, y_1), \dots, (x_n, y_n)$ , find

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

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

- 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.

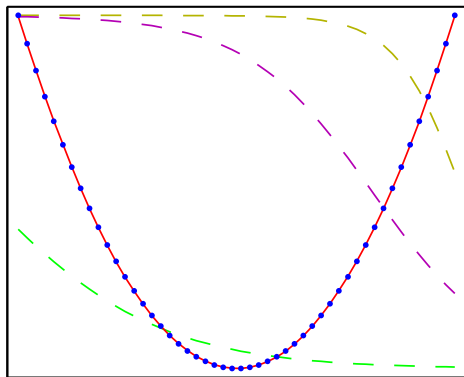
# Approximation Properties of Multilayer Perceptrons

---



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

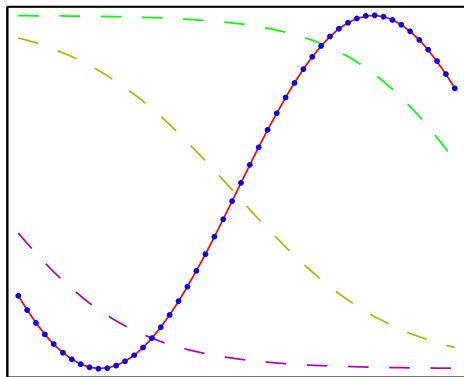
- 3 hidden units; tanh activation functions
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



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

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

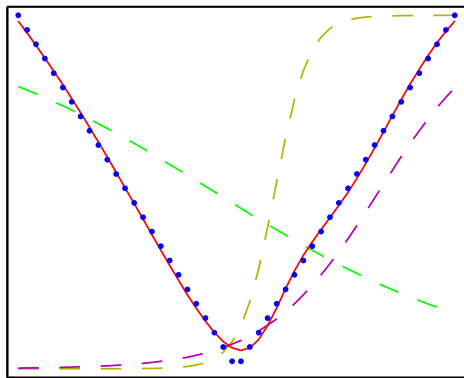
- 3 hidden units; logistic activation function
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



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

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

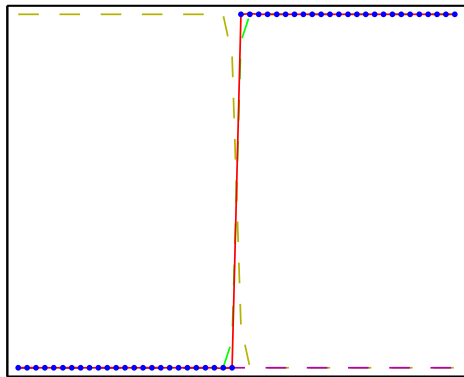
- 3 hidden units; logistic activation functions
- Blue dots are training points; Dashed lines are hidden unit outputs; Final output in Red.



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  $f : K \rightarrow \mathbf{R}$  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:  $\mathcal{X} = \mathbf{R}^d$ ,  $\mathcal{Y} = \{1, \dots, k\}$
- 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  $\theta = (\theta_1, \dots, \theta_k) \in \mathbf{R}^k$ , where
  - $\sum_{y=1}^k \theta_y = 1$  and  $\theta_y \geq 0$  for  $y \in \{1, \dots, k\}$ .

# Multinomial Logistic Regression

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

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

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

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



# Multinomial Logistic Regression: Learning

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

$$\hat{p}(y | x) = \text{Softmax}(\langle w_1, x \rangle, \dots, \langle w_k, x \rangle)_y,$$

where the  $y$  subscript indicates taking the  $y$ 'th entry of the vector produced Softmax.

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

$$\arg \max_{w_1, \dots, w_k} \sum_{i=1}^n \log \left[ \text{Softmax}(\langle w_1, x_i \rangle, \dots, \langle w_k, x_i \rangle)_{y_i} \right].$$

- 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

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

use nonlinear score functions:

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

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

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

# Nonlinear Generalization of Multinomial Logistic Regression

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

$$\arg \max_{w_1, \dots, w_k} \sum_{i=1}^n \log \left[ \text{Softmax}(f_1(x), \dots, f_k(x))_{y_i} \right].$$

- 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  $f : \mathbf{R}^d \rightarrow \mathbf{R}^k$ .
- 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  $\sigma: \mathbf{R} \rightarrow \mathbf{R}$  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

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

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:

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

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

$$J(\theta) = \frac{1}{n} \sum_{i=1}^n \text{Softmax}(f(x_i))_{y_i},$$

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

# Neural Network Regularization

---



# Neural Network Regularization

- Neural networks are very expressive.
- Correspond to big hypothesis spaces.
- Many approaches are used for regularization.

## Tikhonov Regularization? Sure.

- Can add an  $\ell_2$  and/or  $\ell_1$  regularization terms to our objective:

$$J(w, v) = \sum_{i=1}^n (y_i - f_{w,v}(x_i))^2 + \lambda_1 \|w\|^2 + \lambda_2 \|v\|^2$$

- In neural network literature, this is often called **weight decay**.

# Regularization by Early Stopping

A particular recipe for early stopping:

- As we train, check performance on validation set every once in a while.
- Don't stop immediately after validation error goes back up.
- The “**patience**” parameter: the number of training steps to continue after finding a minimum of validation error.
  - Start with  $\text{patience} = 10000$ .
  - Whenever we find a minimum at step  $T$ ,
    - Set  $\text{patience} \leftarrow \text{patience} + cT$ , for some constant  $c$ .
    - Then run at least  $\text{patience}$  extra steps before stopping.

---

See <http://arxiv.org/pdf/1206.5533v2.pdf> for details.

# Max-Norm Regularization

- **Max-norm regularization:** Enforce max norm of incoming weight vector at every hidden node to be bounded:

$$\|w\|_2 \leq c.$$

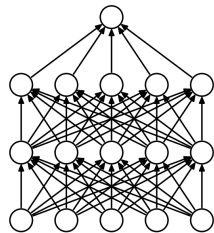
- Project any  $w$  that's too large onto ball of radius  $c$ .
- It's like  $\ell_2$ -complexity control, but locally at each node.
- Why?
  - There are heuristic justifications, but proof is in the performance.
  - We'll see below.

---

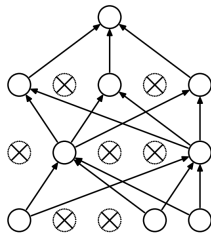
See <http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf> for details.

# Dropout for Regularization

- A recent trick for improving generalization performance is **dropout**.
- A fixed probability  $p$  is chosen.
- Before every stochastic gradient step,
  - each node is selected for “dropout” with probability  $p$
  - a dropout node is removed, along with its links
  - after the stochastic gradient step, all nodes are restored.



(a) Standard Neural Net

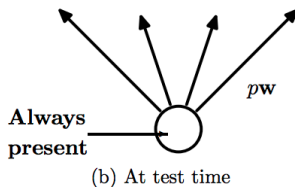
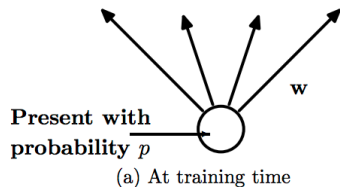


(b) After applying dropout.

Figure from <http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>.

# Dropout for Regularization

- At prediction time
  - all nodes are present
  - outgoing weights are multiplied by  $p$ .



- Dropout probability set using a validation set, or just set at 0.5.
  - Closer to 0.8 usually works better for input units.

Figure from <http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>.

## Dropout: Why might this help?

- Since any node may randomly disappear,
  - forced to “spread the knowledge” across the nodes.
- Each hidden node only gets a randomly chosen sample of its inputs,
  - so won't become too reliant on any single input.
  - More robust.

## Dropout: Does it help?

Results from MNIST (handwritten digit recognition)

Method	Unit Type	Architecture	Error %
Standard Neural Net (Simard et al., 2003)	Logistic	2 layers, 800 units	1.60
SVM Gaussian kernel	NA	NA	1.40
Dropout NN	Logistic	3 layers, 1024 units	1.35
Dropout NN	ReLU	3 layers, 1024 units	1.25
Dropout NN + max-norm constraint	ReLU	3 layers, 1024 units	1.06
Dropout NN + max-norm constraint	ReLU	3 layers, 2048 units	1.04
Dropout NN + max-norm constraint	ReLU	2 layers, 4096 units	1.01
Dropout NN + max-norm constraint	ReLU	2 layers, 8192 units	0.95
Dropout NN + max-norm constraint (Goodfellow et al., 2013)	Maxout	2 layers, (5 × 240) units	0.94

Figure from <http://www.cs.toronto.edu/~rsalakhu/papers/srivastava14a.pdf>.



# How big a network?

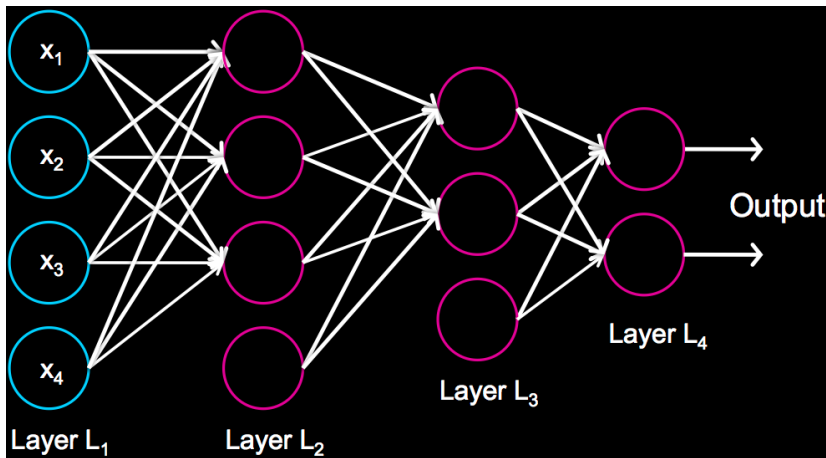
- How many hidden units?
- Current conventional wisdom:
  - With proper regularization, too many doesn't hurt.
  - Except in computation time.

## 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>)

- 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.

# 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 for Features

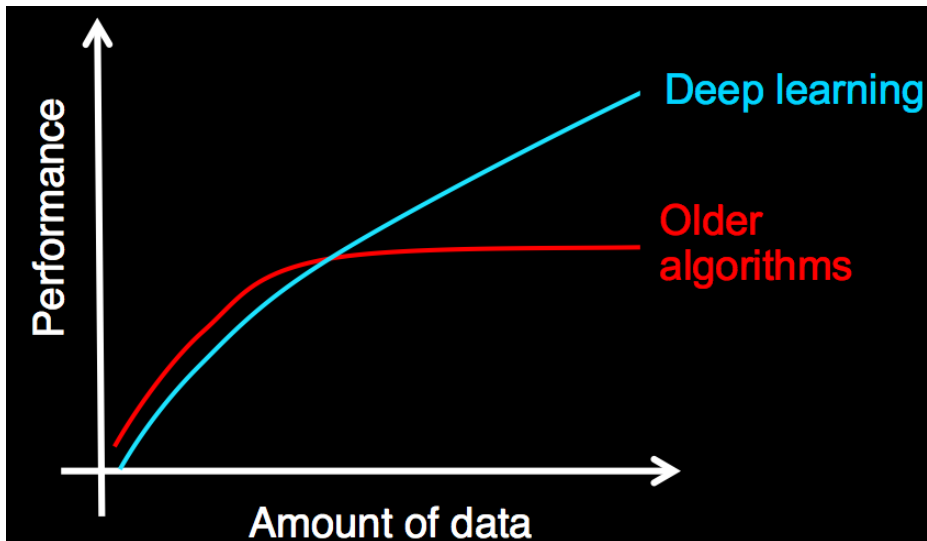
- OverFeat is a neural network for image classification
  - Trained on the huge ImageNet dataset
  - Lots of computing resources used for training the network.
- All those hidden layers of the network are very valuable **features**.
  - Paper: “*CNN Features off-the-shelf: an Astounding Baseline for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

## 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.
- Typically on huge networks.

## Google Brain



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

# Neural Networks: When to Use?

- Computer vision problems
  - All state of the art methods use neural networks
- Speech recognition
  - All state of the art methods use neural networks
- Natural Language problems
  - Maybe. State-of-the-art, but not as large a margin.
  - Check out “word2vec” <https://code.google.com/p/word2vec/>.
  - Represents words using real-valued vectors.
    - Potentially much better than bag of words.