Deep Neural Networks

ESM5205 Learning from Big Data | Sep 18, 2019

Seokho Kang

성균관대학교
SUNG KYUN KWAN UNIVERSITY(SKKU)

Review

- **Machine learning (supervised learning) basics**
  - Parameter/hyperparameter
  - Generalization
  - Overfitting/underfitting
  - Regularization
  - Training/validation/test

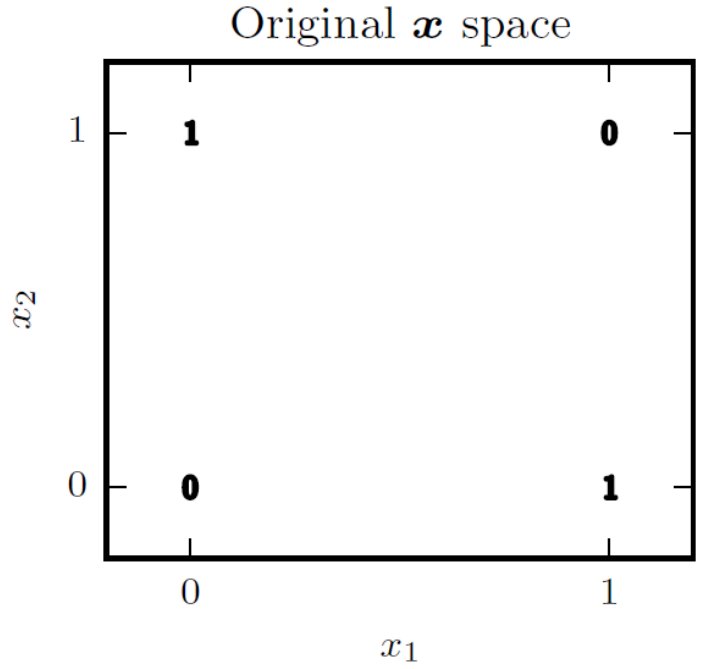
- **Learning algorithms**
  - Linear regression
  - Logistic regression
  - K-nearest neighbors
  - Support vector machines

Learning XOR

- **XOR function ("exclusive or")**

x_1	x_2
0	0
0	1
1	0
1	1

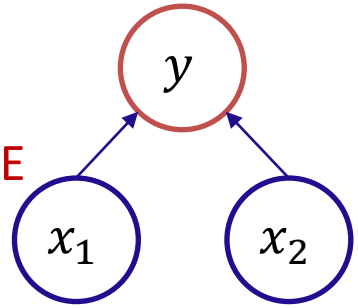
y
0
1
1
0



- training set: $D = \{([0,0]^T, 0), ([0,1]^T, 1), ([1,0]^T, 1), ([1,1]^T, 0)\}$
- model $f(\boldsymbol{x}; \boldsymbol{\theta})$ that fits D, where $\boldsymbol{\theta}$ is the vector of parameters

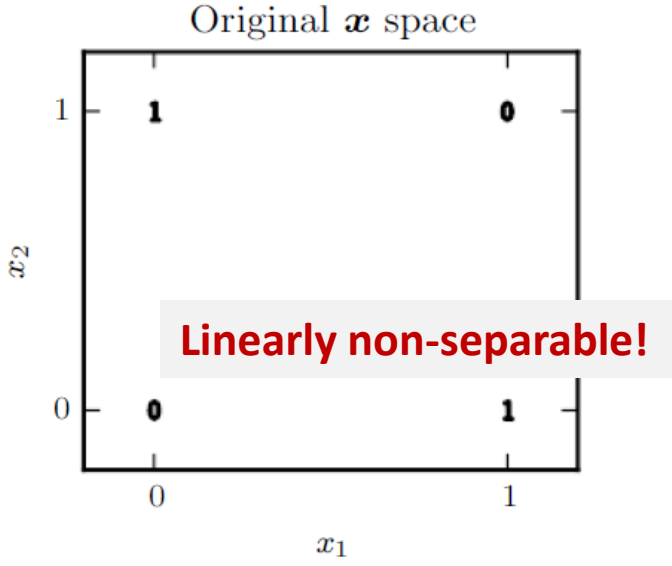
Learning XOR

- **If we choose a linear model,**
  - model $f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{w}^T \boldsymbol{x} + b$, $\boldsymbol{x} = (x_1, x_2)$
  - cost function $J(\boldsymbol{w}, b) = \frac{1}{4}\sum_{(\boldsymbol{x}_i, y_i) \in D} (y_i - f(x; \boldsymbol{w}, b))^2$ ← MSE



- **After training…** (how?)
  - Optimal parameters $\boldsymbol{w} = \boldsymbol{0}$ and $b = 0.5$
  - A linear model is not able to represent XOR, simply outputs 0.5 everywhere.

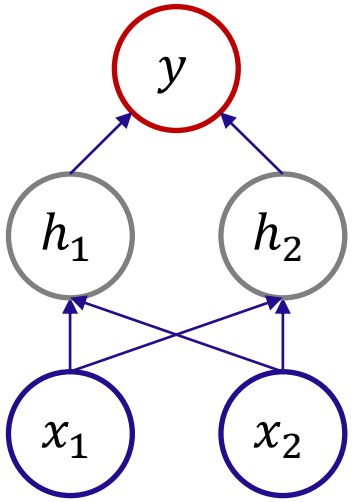
- **We need to use a different feature space in which a linear model will work**



Learning XOR

- **A very simple "neural network" with one hidden layer containing two hidden features $\boldsymbol{h} = (h_1, h_2)$**
  - hidden features $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \mathbf{W}, \mathbf{c})$ ← ?
  - output $\hat{y} = f^{(2)}(\boldsymbol{h}; \mathbf{w}, b) = \boldsymbol{w}^T \boldsymbol{h} + b$ ← linear function
  - Complete model: the two functions chained together

$$\hat{y} = f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)$$



x_1	x_2
0	0
0	1
1	0
1	1

h_1	h_2
?	?
?	?
?	?
?	?

y
0
1
1
0

Learning XOR

- **For the model $\hat{y} = f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)$, what should be $f^{(1)}$?**
  What will happen if $f^{(1)}$ is linear (*i.e.*, $f^{(1)}(\boldsymbol{x})=\mathbf{W}^T\boldsymbol{x}+\boldsymbol{c}$)**?**

$$\boldsymbol{h} = \mathbf{W}^T\boldsymbol{x} + \boldsymbol{c}$$

$$\hat{y} = f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right) = \mathbf{w}^{\mathrm{T}}\boldsymbol{h} + b = \mathbf{w}^T(\mathbf{W}^T\boldsymbol{x} + \boldsymbol{c}) + b$$
$$= \left(\mathbf{w}^{\mathbf{T}}\boldsymbol{W}^T\right)\boldsymbol{x} + \left(\mathbf{w}^{\mathrm{T}}\boldsymbol{c} + b\right) = \boxed{\mathbf{w}'^{\mathbf{T}}\boldsymbol{x} + b'}$$

- **Non-linearity is needed to learn complex (non-linear) representations of data, otherwise the neural network would be just a linear function.**
  For $f^{(1)}$, we must use a nonlinear "activation function" g!

$$\boldsymbol{h} = g(\mathbf{W}^T\boldsymbol{x} + \boldsymbol{c})$$

Learning XOR

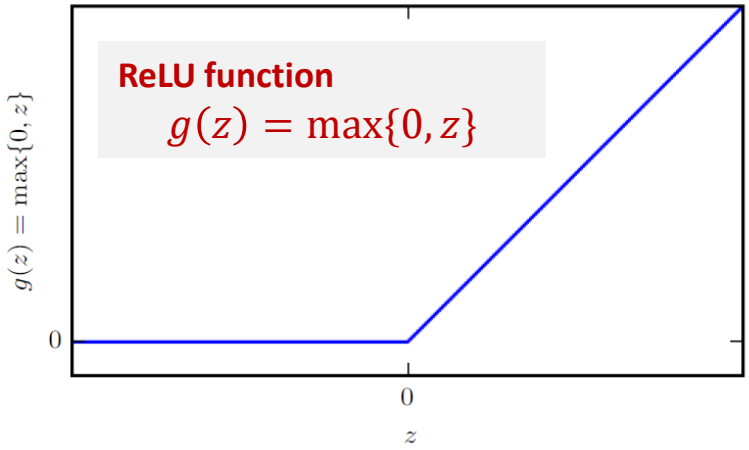
- ***Example***: rectified linear unit (ReLU) $g(z) = \max\{0, z\}$ as $f^{(1)}$: $\hat{y} = \mathbf{w}^{\mathrm{T}} f^{(1)}(\boldsymbol{x}) + b = \max\{0, \mathbf{W}^{\mathrm{T}}\boldsymbol{x} + \mathbf{c}\} + b$

  - Let's have a hand-coded solution (not learned) of the parameters $\mathbf{W}, \mathbf{c}, \mathbf{w}, b$ below

$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

  - The matrix representation of the training set is

$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$



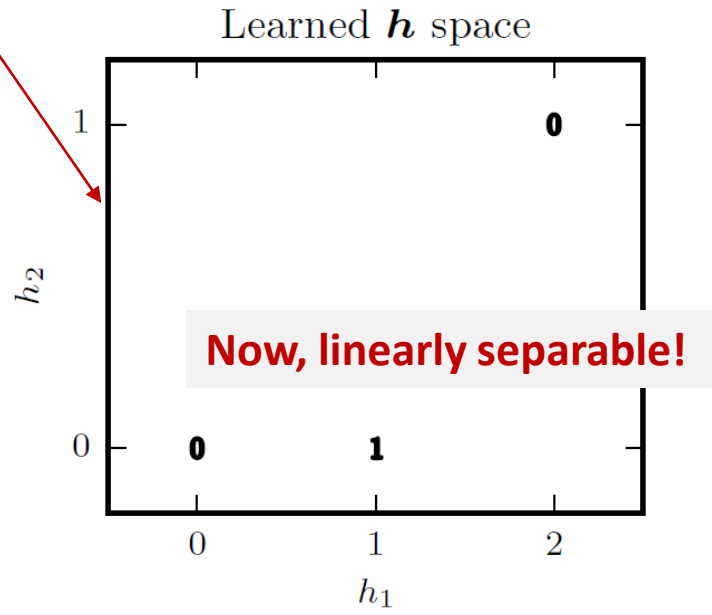
Learning XOR

- ***Example***: rectified linear unit (ReLU) $g(z) = \max\{0, z\}$ as $f^{(1)}$: $\hat{y} = \mathbf{w}^{\mathrm{T}} f^{(1)}(\boldsymbol{x}) + b = \max\{0, \mathbf{W}^{\mathrm{T}}\boldsymbol{x} + \mathbf{c}\} + b$

  - Then…

$$XW + \mathbf{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix} \qquad \max\{0, XW + \mathbf{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

$$\mathbf{w}^T \max\{0, XW + \mathbf{c}\} + b = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

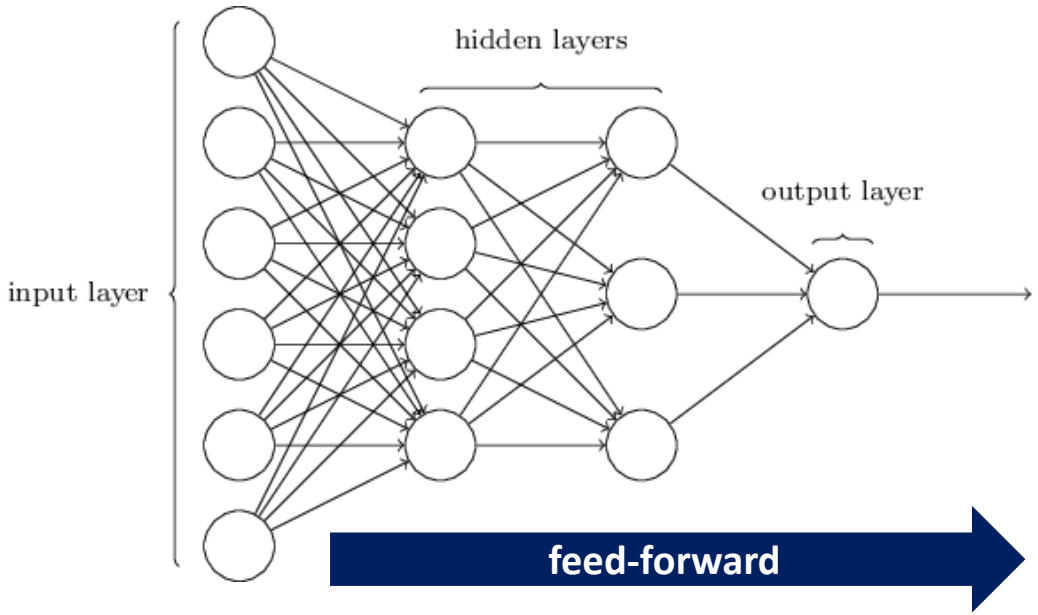


  - Now, $\hat{y}_i = y_i, \forall i$!

Feedforward Neural Network

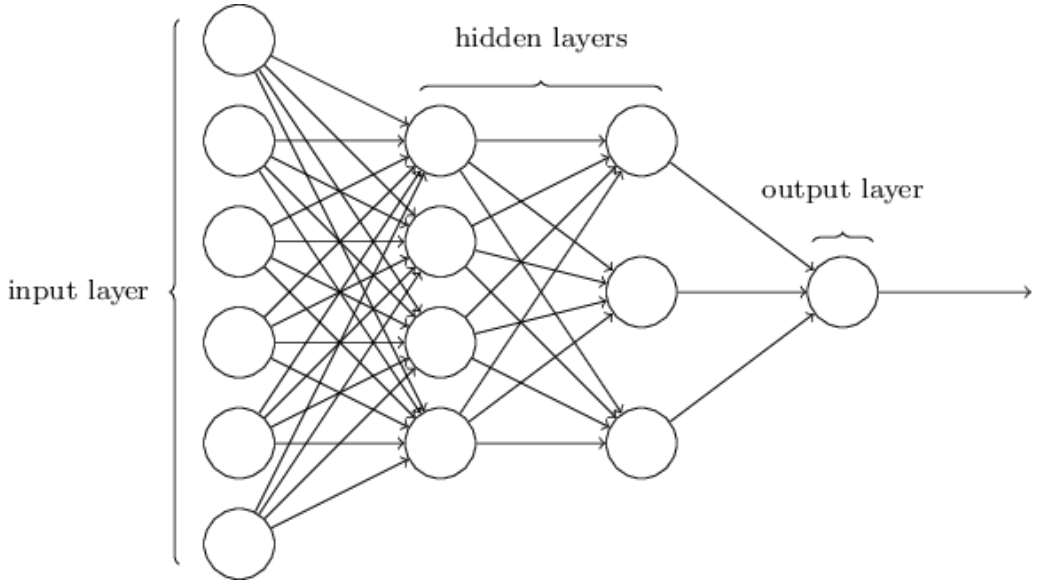
- **Feedforward Neural Network** (*a.k.a.* MultiLayer Perceptron)
  - Input Layer, Hidden Layers (0 to many), and Output Layer
  - **"feedforward":** information flows through the network from input to output (No feedback/recurrent connections).
  - Multiple layers $f^{(1)}, f^{(2)}, \ldots, f^{(l)}$ are connected in a chain to form

$$f(\boldsymbol{x}) = f^{(l)}\left(\ldots\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right)\right)$$



Feedforward Neural Network

- **Feedforward Neural Network** (*a.k.a.* MultiLayer Perceptron)
  - **Relation to other learning algorithms?**
    - A feedforward neural network with a single linear output unit and no hidden layer
      → linear regression
    - A feedforward neural network with a single sigmoid output unit and no hidden layer
      → logistic regression



Feedforward Neural Network

- Training a neural network is not much different from training any other machine learning model with gradient descent-based optimization.
  - *e.g.,* logistic regression, support vector machine, …

- To apply gradient descent, we must choose a **cost function** and how to represent the **output/hidden units** of a neural network.

- The largest difference is that the non-linearity of a neural network causes the **cost function** to become **nonconvex**.
  - many local optima may exist, global optimum cannot be guaranteed.

Training a Neural Network

- Given a training dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (x_{i1}, \dots, x_{id}) \in \mathbb{R}^d$ is the *i*-th input vector of the *d* input variables and y_i is the corresponding label of the output variable.

- The model: $\hat{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- The cost function (to be minimized)

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{(\boldsymbol{x}_i, y_i) \in D} L(y_i, \hat{y}_i)$$

- Training: let's consider simple gradient descent

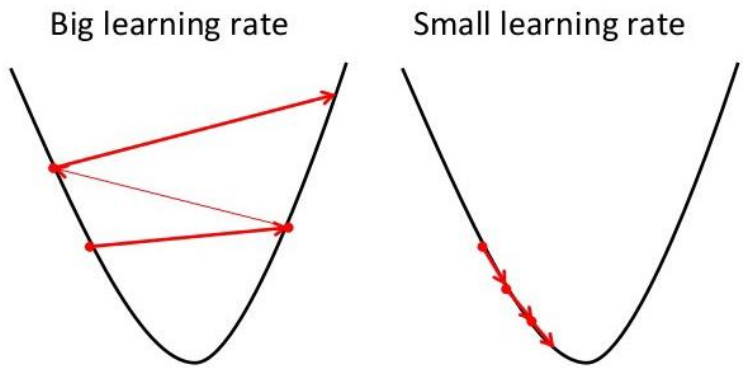
$$\boldsymbol{\theta} \coloneqq \boldsymbol{\theta} - \epsilon \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$$

$$\rightarrow \theta_j \coloneqq \theta_j - \epsilon \frac{\partial}{\partial \theta_j} J(\boldsymbol{\theta}), \forall \theta_j \in \boldsymbol{\theta}$$

$\epsilon > 0$ is the learning rate

how to calculate $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$ for a deep neural network?

the cost function for a deep neural network is non-convex.



Training a Neural Network

- Given a training dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (x_{i1}, \dots, x_{id}) \in \mathbb{R}^d$ is the *i*-th input vector of the *d* input variables and y_i is the corresponding label of the output variable.

- **For the training set D,**
  - **Forward propagation**: The information from **input $\boldsymbol{x}$** flows **forward** through the network to get **prediction $\hat{y}$** and to compute the **cost $J(\boldsymbol{\theta})$**
  - **Backpropagation**: The information from $J(\boldsymbol{\theta})$ flows **backward** through the network to compute the **gradient** of the cost with respect to the parameters $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})$

Backpropagation

- **Let's recall "chain rule" of calculus!**
  - ***Example 1***: Univariate case
    - Let $x, y \in \mathbb{R}$, and $f, g: \mathbb{R} \to \mathbb{R}$
    - Suppose that $y = g(x)$ and $z = f\big(g(x)\big) = f(y)$
    - Then, the chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

$$x \to y \to z$$

  - ***Example 2***: Bivariate case
    - Let $\boldsymbol{x} \in \mathbb{R}^2$, $\boldsymbol{y} \in \mathbb{R}^2$, and $f: \mathbb{R}^2 \to \mathbb{R}$, $g: \mathbb{R}^2 \to \mathbb{R}^2$
    - Suppose that $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f\big(g(\boldsymbol{x})\big) = f(\boldsymbol{y})$
    - Then, the chain rule states that

$$\begin{matrix} x_1 \\ x_2 \end{matrix} \begin{matrix} \nearrow \\ \searrow \end{matrix} \begin{matrix} y_1 \\ y_2 \end{matrix} \begin{matrix} \searrow \\ \nearrow \end{matrix} z$$

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial z}{\partial x_1}, \frac{\partial z}{\partial x_2} \right), \qquad \frac{\partial z}{\partial x_1} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_1} + \frac{\partial z}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_1}, \qquad \frac{\partial z}{\partial x_2} = \frac{\partial z}{\partial y_1} \cdot \frac{\partial y_1}{\partial x_2} + \frac{\partial z}{\partial y_2} \cdot \frac{\partial y_2}{\partial x_2}$$

Backpropagation

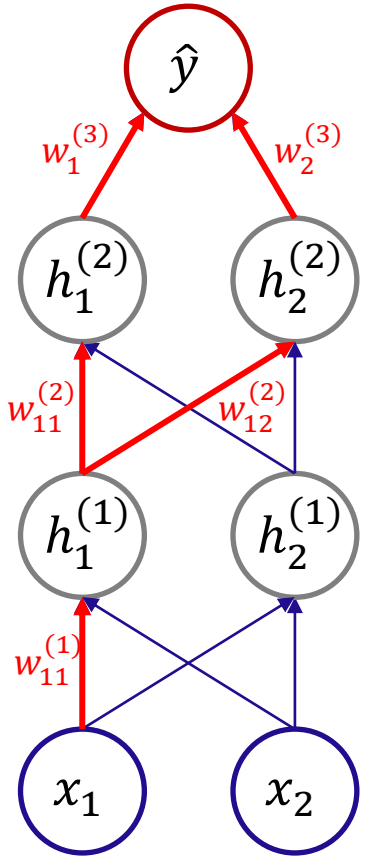
- **Let's recall "chain rule" of calculus!**
  - ***Example 3***: Multivariate case
    - Let $\boldsymbol{x} \in \mathbb{R}^m$, $\boldsymbol{y} \in \mathbb{R}^n$, and $f: \mathbb{R}^n \to \mathbb{R}$, $g: \mathbb{R}^m \to \mathbb{R}^n$
    - Suppose that $y = g(\boldsymbol{x})$ and $z = f\big(g(\boldsymbol{x})\big) = f(\boldsymbol{y})$
    - Then, the chain rule states that

$$\nabla_{\boldsymbol{x}} z = \left(\frac{\partial z}{\partial x_1}, \frac{\partial z}{\partial x_2}, \ldots, \frac{\partial z}{\partial x_m} \right),$$

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \cdot \frac{\partial y_j}{\partial x_i}$$

Backpropagation

- **Applying "chain rule" to the training of a neural network**
  - ***Example:*** a simple neural network with 10 parameters, training set D



$$J(\boldsymbol{w}) = \frac{1}{n} \sum_{(x_i, y_i) \in D} L(y_i, \hat{y}_i)$$

1. Calculate gradients

$$\frac{\partial}{\partial w_1^{(3)}} J(\boldsymbol{w}) = \sum_{i=1}^{n} \frac{\partial J(\boldsymbol{w})}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial w_1^{(3)}}$$

$$\frac{\partial}{\partial w_{11}^{(2)}} J(\boldsymbol{w}) = \sum_{i=1}^{n} \left(\frac{\partial J(\boldsymbol{w})}{\partial \hat{y}_i} \frac{\partial \hat{y}_i}{\partial h_1^{(2)}} \frac{\partial h_1^{(2)}}{\partial w_{11}^{(2)}} \right)$$

$$\frac{\partial}{\partial w_{11}^{(1)}} J(\boldsymbol{w}) = \ ?$$

2. Update parameters

$$w_j := w_j - \epsilon \frac{\partial}{\partial w_j} J(\boldsymbol{w}), \qquad \forall w_j \in \boldsymbol{w}$$

Training a Neural Network

1. **Design a neural network** → following slides
2. **Initialize parameters to small random numbers**
3. **Repeat following until terminating condition is met (when error is very small, etc.)**
   A. (forward propagation) Propagate the inputs forward
   B. (backpropagation) Backpropagate the cost and update parameters

→ **Lecture 3 Optimization**

Designing a Neural Network

- **We need to determine…**
  - Cost Functions
  - Output Units
  - Hidden Units
  - Depth (No. hidden layers)
  - Width (No. hidden units in each hidden layer)
  - …

They are all hyperparameters.

Cost Functions

- Given a training dataset $D = \{(\boldsymbol{x}_1, y_1), (\boldsymbol{x}_2, y_2), \dots, (\boldsymbol{x}_n, y_n)\}$ such that $\boldsymbol{x}_i = (x_{i1}, \dots, x_{id}) \in \mathbb{R}^d$ is the *i*-th input vector of the *d* input variables and y_i is the corresponding label of the output variable.

- In a probabilistic view, a neural network estimates a probability distribution $p(y|\boldsymbol{x}; \boldsymbol{\theta})$, where $\boldsymbol{\theta}$ are the parameters of the neural network to be learned.
- We can simply use maximum likelihood estimation to find the best parameters $\boldsymbol{\theta}^*$.

$$\boldsymbol{\theta}^* = \underset{\boldsymbol{\theta}}{\operatorname{argmax}} \sum_{(\boldsymbol{x}_i, y_i) \in D} \log p(y_i|\boldsymbol{x}_i; \boldsymbol{\theta})$$

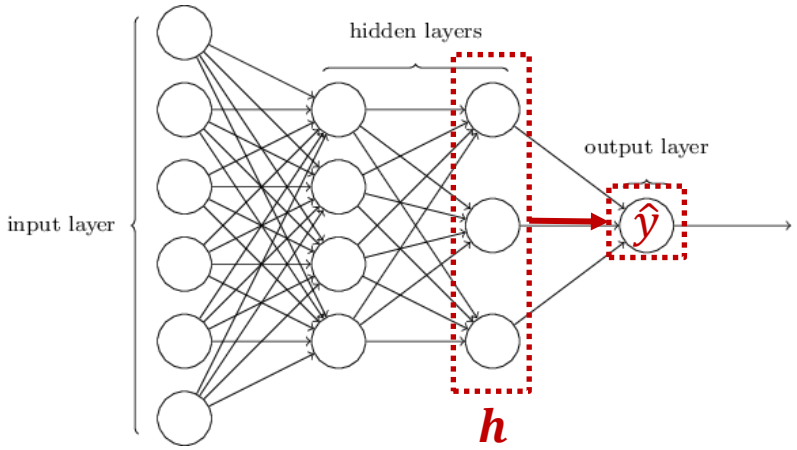
- The cost function (to be minimized)

$$J(\boldsymbol{\theta}) = \sum_{(\boldsymbol{x}_i, y_i) \in D} -\log p(y_i|\boldsymbol{x}_i; \boldsymbol{\theta})$$

- The choice of cost function is tightly coupled with the choice of **output unit**.

Output Units

- Output layer provides **additional transformation from the last hidden features $\boldsymbol{h}$** to complete **the target task**.
  : $\hat{y} = f^{(l)}(\mathbf{h})$



- **Linear Units** (for regression, $y \in \mathbb{R}$)
  - $\hat{y} = \mathbf{w}^{\mathrm{T}}\mathbf{h} + b$
  - Often used to produce the mean of a Normal distribution:

$$p(y|\boldsymbol{x};\boldsymbol{\theta}) = \mathcal{N}(y|\hat{y},\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}}\exp\left(-\frac{\|y-\hat{y}\|^2}{2\sigma^2}\right)$$

  - The corresponding cost function? Maximizing the log likelihood: $\log\prod_{(\boldsymbol{x}_i,y_i)\in D} p(y_i|\boldsymbol{x}_i;\boldsymbol{\theta})$?

Output Units

- **Sigmoid Units** (for binary classification, $y \in \{0,1\}$)
  - $\hat{y} = P(y = 1|\boldsymbol{x}; \boldsymbol{\theta}) = \sigma(z) = \sigma\left(\mathbf{w}^{\mathrm{T}}\mathbf{h} + b\right)$
  - $\hat{y}$ saturates to 1 if $z \to \infty$, $\hat{y}$ saturates to 0 if $z \to -\infty$ (sigmoid function)
  - Often used to define a Bernoulli distribution:

$$p(y|\boldsymbol{x}; \boldsymbol{\theta}) = Bernoulli(y|\hat{y}) = (\hat{y})^{y}(1 - \hat{y})^{1-y}$$

  - The corresponding cost function? Maximizing the log-likelihood: $\log \prod_{(\boldsymbol{x}_i, y_i) \in D} p(y_i|\boldsymbol{x}_i; \boldsymbol{\theta})$?

- **Softmax Units** (for multi-class classification , $y \in \{1,2, \dots, c\}$, $\boldsymbol{y} = \text{one\_hot}(y)$)
  - $\hat{\boldsymbol{y}} = (\hat{y}_1, \dots, \hat{y}_c)$, where $\hat{y}_k = p(y = k|\boldsymbol{x}; \boldsymbol{\theta})$
  - $\boldsymbol{z} = (z_1, \dots, z_c) = \mathbf{W}^{\mathrm{T}}\mathbf{h} + \boldsymbol{b}$, $\hat{y}_k = \text{softmax}(\boldsymbol{z})_k = \frac{\exp(z_k)}{\sum_j \exp(z_j)}$ so that $\sum_k \hat{y}_k = 1$
  - Often used to define a Multinoulli distribution:

$$p(y|\boldsymbol{x}; \boldsymbol{\theta}) = \prod_{k=1}^{c} \hat{y}_k^{\mathrm{I}(y=k)}$$

  - The corresponding cost function? Maximizing the log-likelihood: $\log \prod_{(\boldsymbol{x}_i, y_i) \in D} p(y_i|\boldsymbol{x}_i; \boldsymbol{\theta})$?

Output Units

- **Summary (Typical Choice)**
  - For binary classification ($y_i \in \{0,1\}$), use binary cross-entropy

$$L(y_i, \hat{y}_i) = [-y_i \log \hat{y}_i - (1 - y_i)\log(1 - \hat{y}_i)]$$

  - For multi-class classification ($y_i \in \{1,2, \dots, c\}$, $\boldsymbol{y}_i = \text{one\_hot}(y_i) = (y_{i1}, \dots, y_{ic})$), use categorical cross-entropy

$$L(\boldsymbol{y}_i, \hat{\boldsymbol{y}}_i) = -\sum_{k=1}^{c} y_{ik} \log \widehat{y_{ik}}$$

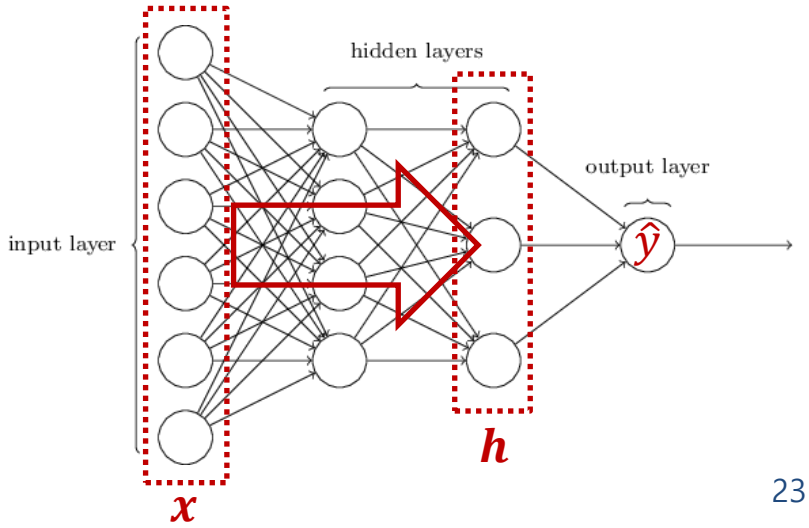
  - For regression ($y_i \in \mathbb{R}$), use squared error

$$L(y_i, \hat{y}_i) = (\widehat{y_i} - y_i)^2$$

Output Type	Output Distribution	Output Layer	Cost Function
Binary	Bernoulli	Sigmoid	Binary cross-entropy
Discrete	Multinoulli	Softmax	Discrete cross-entropy
Continuous	Gaussian	Linear	Gaussian cross-entropy (MSE)

Hidden Units

- Hidden Units should be **non-linear** to deal with non-linearity of data
  - Use non-linear activation function $g(z)$
  - More hidden layers and units result in a more complex model.
- How do they work? Chain-based architecture
  - The first hidden layer accepts a vector of inputs $\boldsymbol{x}$, computing $\boldsymbol{z}^{(1)} = \mathbf{W}^{(1)^T}\boldsymbol{x} + \boldsymbol{b}^{(1)}$, then element-wise non-linear function $g^{(1)}\left(\boldsymbol{z}^{(1)}\right)$.
  - The *i* th (*i*>1) hidden layer accepts the vector of the *i*-1 th hidden layer $\boldsymbol{h}^{(i-1)}=g^{(i-1)}\left(\boldsymbol{z}^{(i-1)}\right)$, computing $\boldsymbol{z}^{(i)} = \mathbf{W}^{(i)^T}\boldsymbol{h}^{(i-1)} + \boldsymbol{b}^{(i)}$, then $\boldsymbol{h}^{(i)} = g^{(i)}\left(\boldsymbol{z}^{(i)}\right)$.
- Important consideration
  - efficiency, robustness, expressiveness, etc…
  - vanishing/exploding gradient problem…

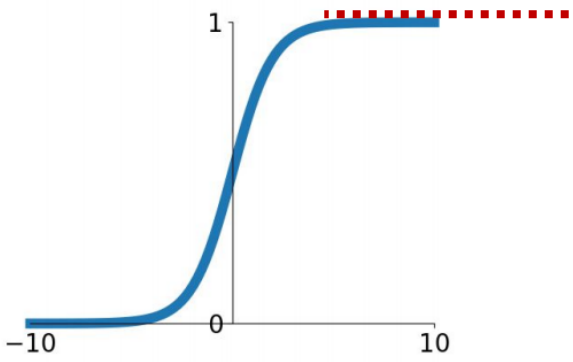


Hidden Units

from ***Lecture 6 of Stanford CS231n:*** ***Convolutional Neural Networks for Visual Recognition***

Activation Functions

$$\sigma(x) = 1/(1+e^{-x})$$



Sigmoid

- Squashes numbers to range [0,1]
- Historically popular since they have nice interpretation as a saturating “firing rate” of a neuron

3 problems:

1. Saturated neurons “kill” the gradients
2. Sigmoid outputs are not zero-centered
3. exp() is a bit compute expensive

Hidden Units

* ***e.g.*** **logistic regression, if** $x_j > 0, \forall j$?

$$L(y, \hat{y}) = -y \log \sigma(z) - (1 - y) \log\big(1 - \sigma(z)\big),$$

where $\hat{y} = \sigma(z), z = \mathbf{w}^T \boldsymbol{x} = w_0 + w_1 x_1 + \cdots + w_d x_d$

$$\frac{\partial L(\boldsymbol{w})}{\partial w_j} = \frac{\partial L(\boldsymbol{w})}{\partial z} \frac{\partial z}{\partial w_j} = (\hat{y} - y) x_j$$

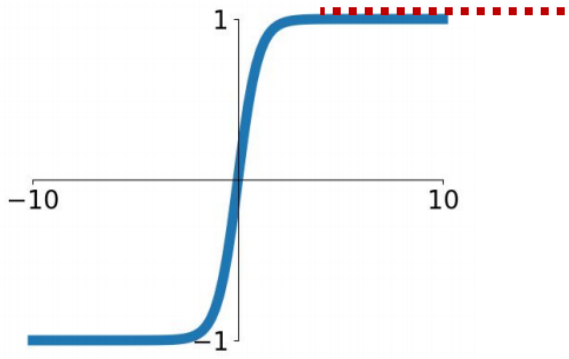
$$\begin{aligned}
\frac{\partial L(\boldsymbol{w})}{\partial z} &= -y \frac{\partial \log \sigma(z)}{\partial z} - (1 - \mathrm{y}) \frac{\partial \log\big(1 - \sigma(z)\big)}{\partial z} \\
&= -y \frac{1}{\sigma(z)} \frac{\partial \sigma(z)}{\partial z} - (1 - \mathrm{y}) \frac{-1}{1 - \sigma(z)} \frac{\partial \sigma(z)}{\partial z} \\
&= -y \frac{1}{\sigma(z)} \sigma(z)\big(1 - \sigma(z)\big) - (1 - \mathrm{y}) \frac{-1}{1 - \sigma(z)} \sigma(z)\big(1 - \sigma(z)\big) \\
&= -y + \sigma(z) = \hat{y} - y
\end{aligned}$$

$$\frac{\partial z}{\partial w_j} = \frac{\partial (w_0 + w_1 x_1 + \cdots + w_d x_d)}{\partial w_j} = x_j$$

Hidden Units

from ***Lecture 6 of Stanford CS231n:*** ***Convolutional Neural Networks for Visual Recognition***

Activation Functions



tanh(x)

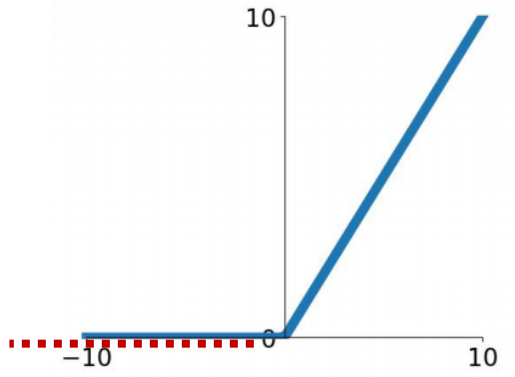
- Squashes numbers to range [-1,1]
- zero centered (nice)
- still kills gradients when saturated :(

$$f(x) = \tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$

Hidden Units

from ***Lecture 6 of Stanford CS231n:*** ***Convolutional Neural Networks for Visual Recognition***

Activation Functions



ReLU
(Rectified Linear Unit)

- Computes **f(x) = max(0,x)**

- Does not saturate (in +region)
- Very computationally efficient
- Converges much faster than sigmoid/tanh in practice (e.g. 6x)
- Actually more biologically plausible than sigmoid

- Not zero-centered output
- An annoyance:

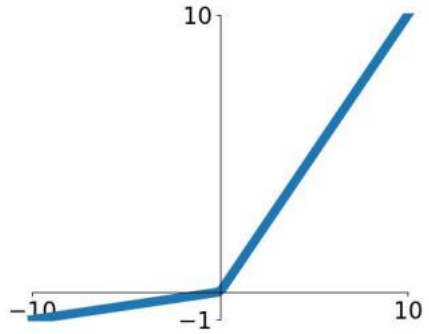
hint: what is the gradient when x < 0?

Hidden Units

from ***Lecture 6 of Stanford CS231n:*** ***Convolutional Neural Networks for Visual Recognition***

Activation Functions

[Mass et al., 2013]
[He et al., 2015]



- Does not saturate
- Computationally efficient
- Converges much faster than sigmoid/tanh in practice! (e.g. 6x)
- **will not "die".**

Leaky ReLU

$$f(x) = \max(0.01x, x)$$

Parametric Rectifier (PReLU)

$$f(x) = \max(\alpha x, x)$$

backprop into \alpha (parameter)

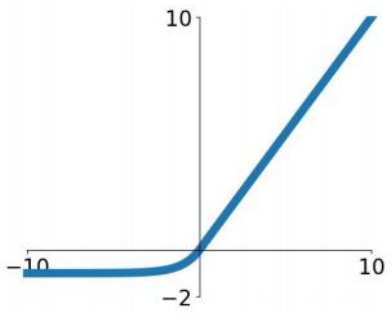
Hidden Units

from ***Lecture 6 of Stanford CS231n:*** ***Convolutional Neural Networks for Visual Recognition***

Activation Functions

[Clevert et al., 2015]

Exponential Linear Units (ELU)



- All benefits of ReLU
- Closer to zero mean outputs
- Negative saturation regime compared with Leaky ReLU adds some robustness to noise

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha\,(\exp(x) - 1) & \text{if } x \leq 0 \end{cases}$$

- Computation requires exp()

Hidden Units

from ***Lecture 6 of Stanford CS231n:*** ***Convolutional Neural Networks for Visual Recognition***

Maxout "Neuron" [Goodfellow et al., 2013]

- Does not have the basic form of dot product -> nonlinearity
- Generalizes ReLU and Leaky ReLU
- Linear Regime! Does not saturate! Does not die!

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

Problem: doubles the number of parameters/neuron :(

Hidden Units

from ***Lecture 6 of Stanford CS231n:*** ***Convolutional Neural Networks for Visual Recognition***

TLDR: In practice:

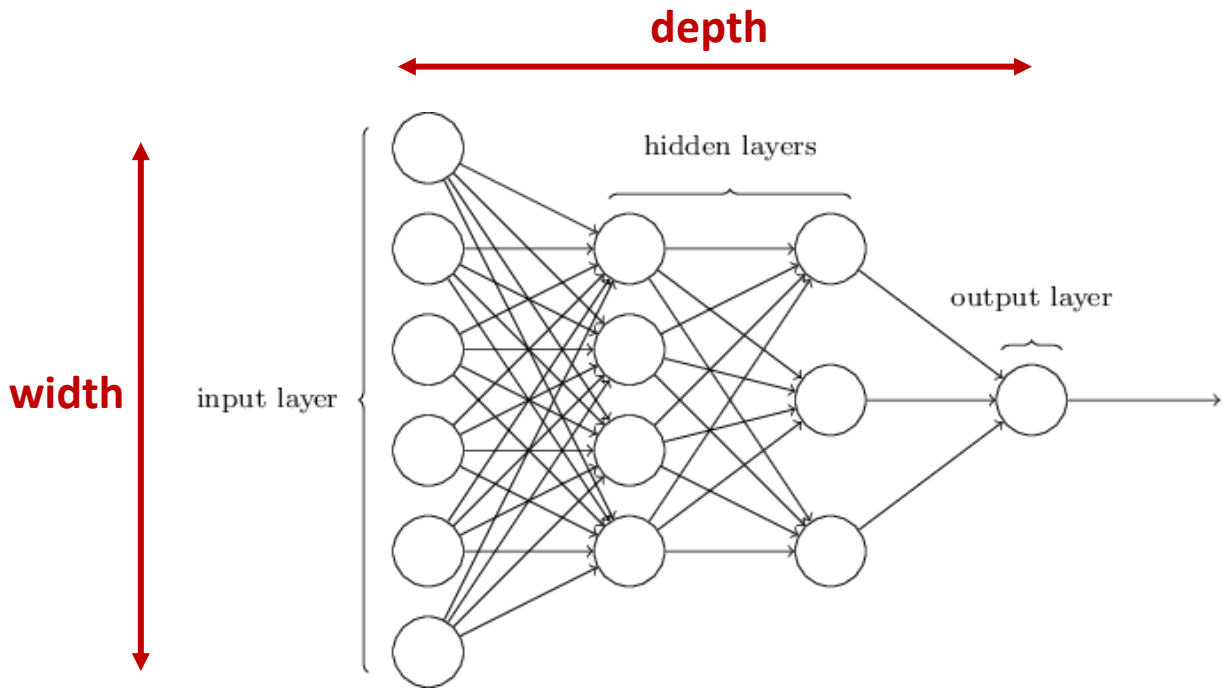
- Use ReLU. Be careful with your learning rates
- Try out Leaky ReLU / Maxout / ELU
- Try out tanh but don't expect much
- Don't use sigmoid

but, when the data are too noisy with extremely large or small values?

→ *tanh and sigmoid are less sensitive to the noise*

Architectural Considerations

- **Depth:** the number of hidden layers
- **Width:** the dimensionality (the number of hidden units) of each hidden layer



Architectural Considerations

- **Universal Approximation Theorem:** A shallow neural network with **one hidden layer** with enough number of "squashing" hidden units (*e.g.*, sigmoid) is enough to represent (not learn!) an approximation of any continuous function.

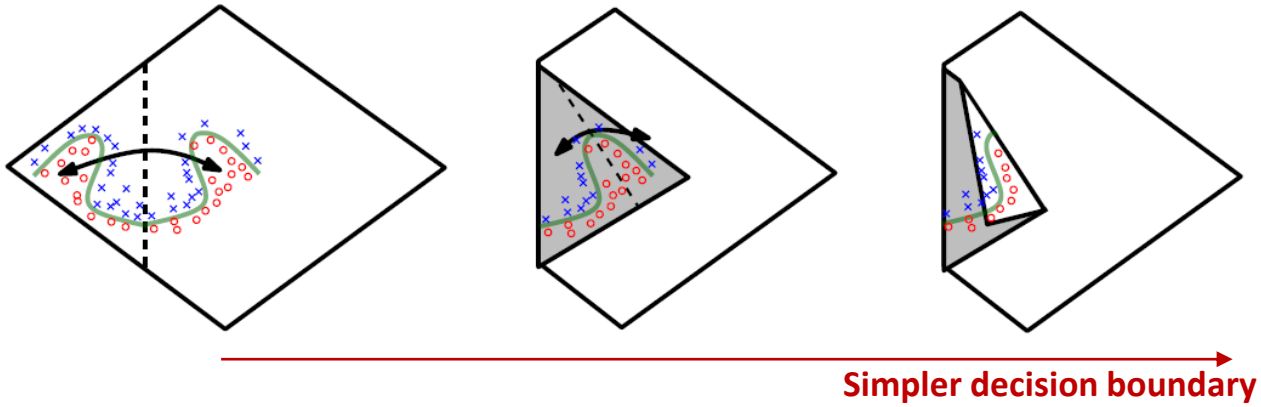
  ** Every continuous function is a Borel measurable function*

- We know that a neural network with one hidden layer will be able to represent any function.

- **Practical Issues:** no. hidden units, training data, training algorithm, etc.
  - We don't know how many hidden units can be considered as 'enough'.
  - Our training data may not properly represent the underlying true function.
  - We are not guaranteed that our training algorithm will be able to learn that function.

Architectural Considerations

- **So, why deeper?**
  - An intuitive explanation of the advantage of deeper network.



  - A special case (Montufar et al., 2014): The number of linear regions carved out by a deep rectifier network *with d inputs, depth l, and n units per hidden layer* is

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right)$$

Architectural Considerations

- **So, why deeper?**
  - Empirical results (Montufar et al., 2014)

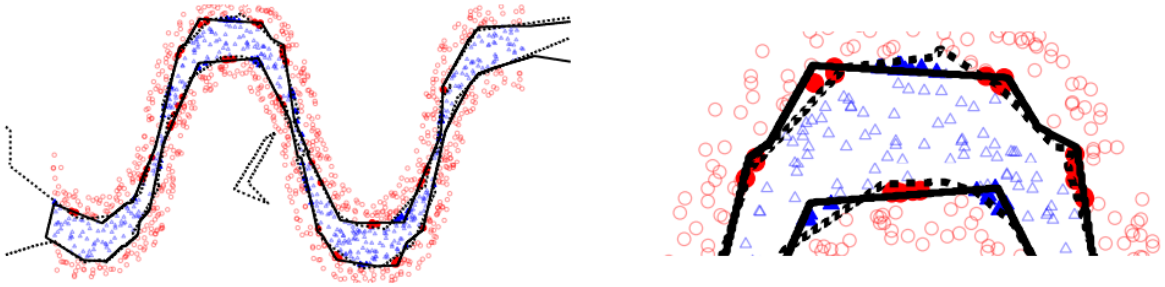


Figure 1: Binary classification using a shallow model with 20 hidden units (solid line) and a deep model with two layers of 10 units each (dashed line). The right panel shows a close-up of the left panel. Filled markers indicate errors made by the shallow model.

  - Empirical results (Goodfellow et al., 2014)

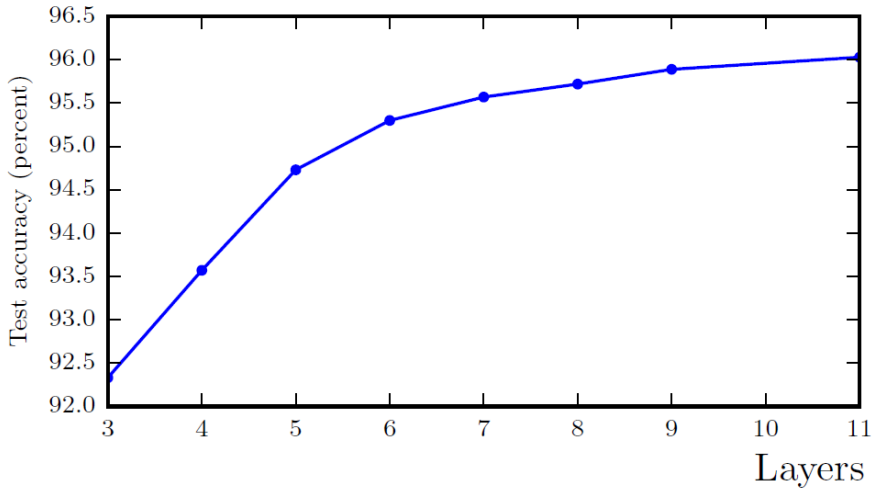


Figure 6.6: Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.