Data Mining
Classification: Alternative Techniques

Lecture Notes for Chapter 5

Introduction to Data Mining

by

Tan, Steinbach, Kumar

These slides have been modified for the CS 4232/ 5232 course

Part IIb

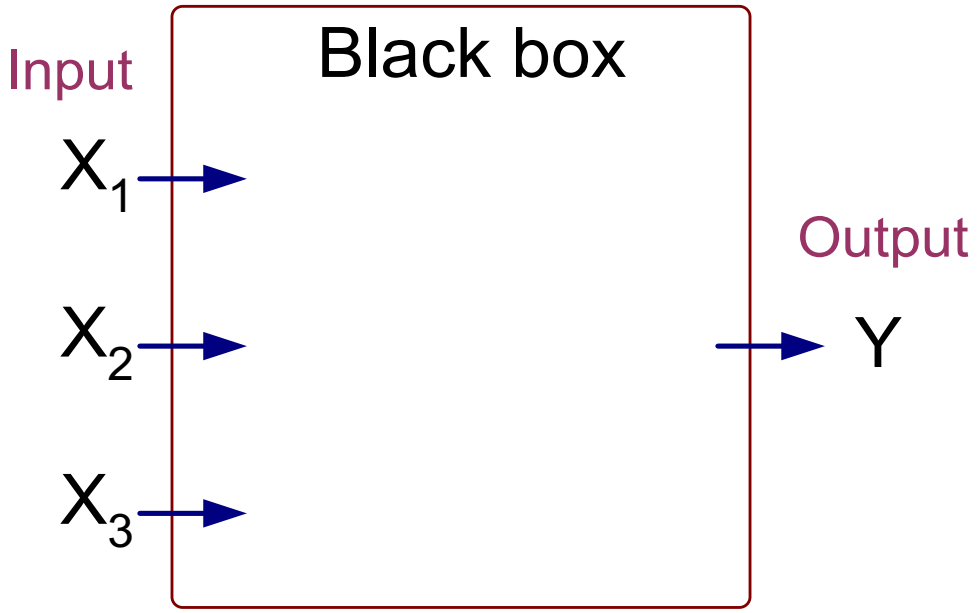
Neural Networks and Deep Learning

Classifiers

Perceptrons

The Perceptron

X_1	X_2	X_3	Y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0

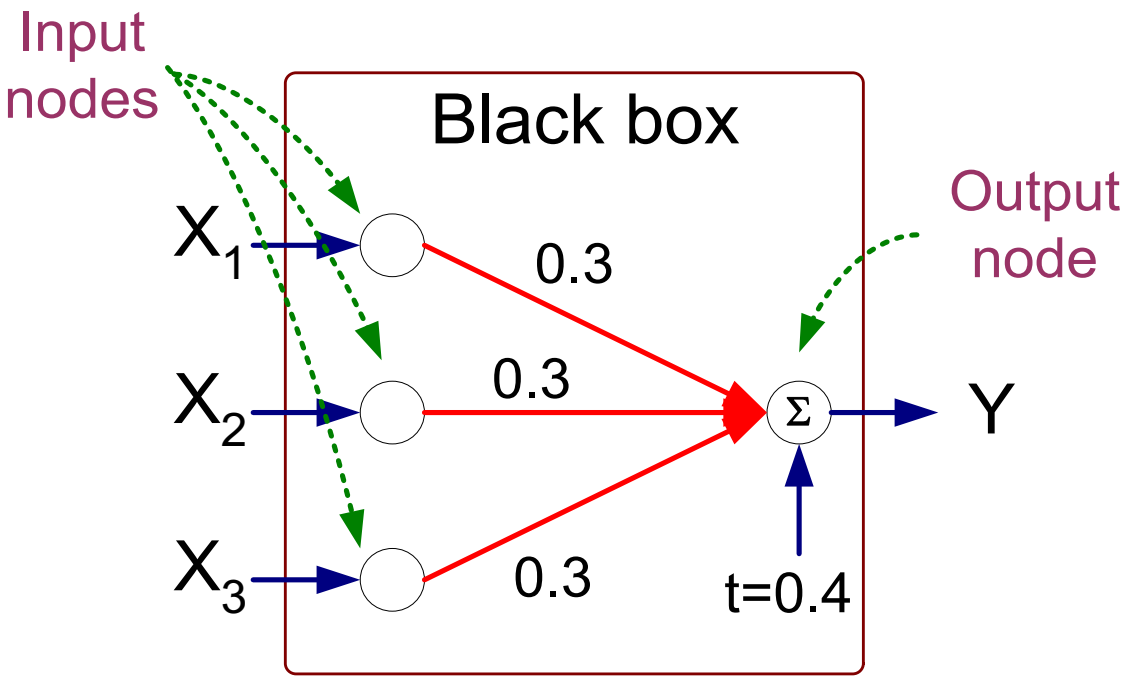


The inputs are binary

Goal of the blackbox: Output Y is 1 if at least two of the three inputs are equal to 1. So your black box should give the same results as the table on the left.

The Perceptron

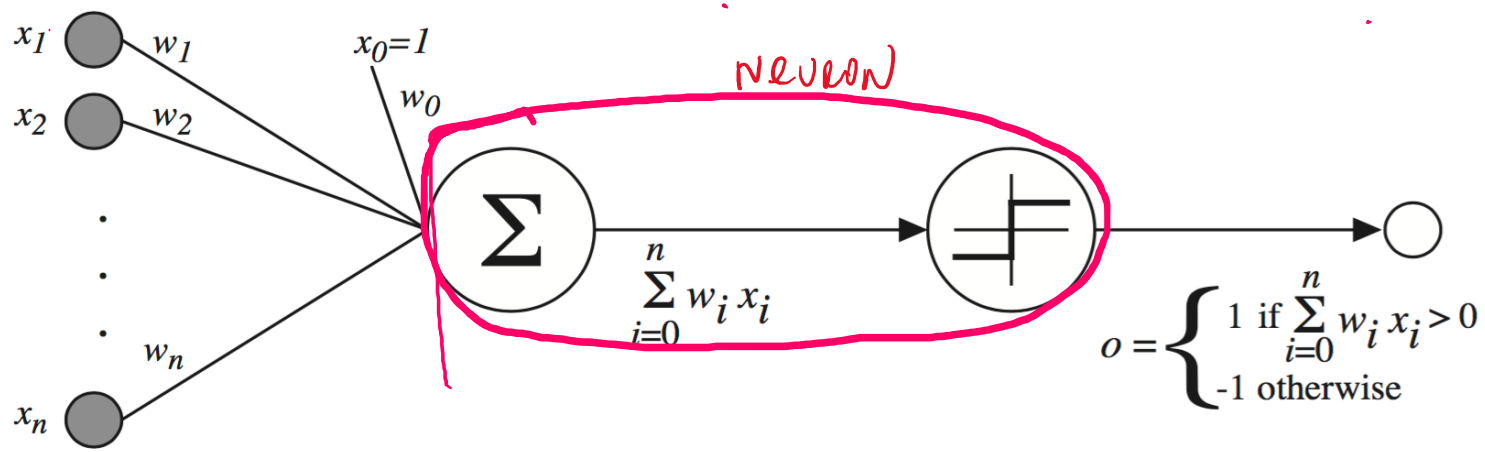
X_1	X_2	X_3	Y
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1
0	0	1	0
0	1	0	0
0	1	1	1
0	0	0	0



$$o(x_1, x_2, x_3) = \begin{cases} 1, & \text{if } (-0.4) + (0.3)x_1 + (0.3)x_2 + (0.3)x_3 > 0 \\ -1, & \text{otherwise} \end{cases}$$

The Perceptron

- In general, a perceptron has the following form:



$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

- Or in matrix form:

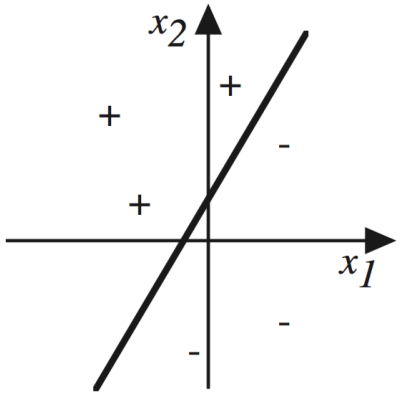
$$o(x_1, \ldots, x_n) = \begin{cases} 1, & \text{if } w^T x > 0 \\ -1, & \text{otherwise} \end{cases}$$

where

$$w = (w_0, \ldots, w_n)^T$$
$$x = (1, x_1, x_2, \ldots, x_n)^T$$

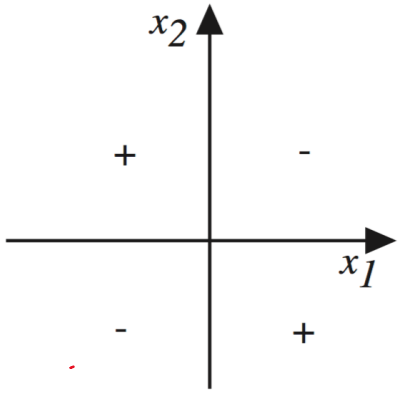
Limitations of The Perceptron

- There are functions that the perceptron cannot "learn", like non-linearly separable cases:



(a)

Linearly separable set of examples



(b)

Non linearly separable set of examples

- In both these cases the perceptron has the form:

$$o(x_1, x_2) = \begin{cases} 1, & \text{if } w_0 + w_1 x_1 + w_2 x_2 > 0 \\ -1, & \text{otherwise} \end{cases}$$

The Unthresholded Perceptron

- The normal perceptron has the form

$$o(x_1, \ldots, x_n) = \begin{cases} 1 & \text{if } w_0 + w_1 x_1 + \cdots + w_n x_n > 0 \\ -1 & \text{otherwise.} \end{cases}$$

- While the unthresholded perceptron (also called linear unit) has the form:

$$o(x_1, x_2, \ldots, x_n) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

Training the Unthresholded Perceptron

- So we need to compute the gradient of *E(w)* with respect to *w*: $\nabla E(w) = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n} \right)$

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} \frac{\partial}{\partial w_i} (t_d - o_d)^2 \\
&= \frac{1}{2} \sum_{d \in D} 2(t_d - o_d) \frac{\partial}{\partial w_i} (t_d - o_d) \\
&= \sum_{d \in D} (t_d - o_d) \frac{\partial}{\partial w_i} (t_d - w^T x_d) \\
&= \sum_{d \in D} (t_d - o_d)(-x_{id})
\end{aligned}$$

$$o(x_1, x_2, \ldots, x_n) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

Where x_d is the d-th training example:

$$x_d = (1, x_{1d}, x_{2d}, \ldots, x_{nd})^T$$

- x_{id} is the i-th component of the d-th input row

E(w) is the error function, also called the loss function L(w).

Training the Unthresholded Perceptron

- So we know how to compute the gradient of *E(w)* with respect to *w*:

$$o(x_1, x_2, \ldots, x_n) = w_0 + w_1 x_1 + w_2 x_2 + \cdots + w_n x_n$$

Where x_d is the d-th training example:

$$x_d = (1, x_{1d}, x_{2d}, \ldots, x_{nd})^T$$

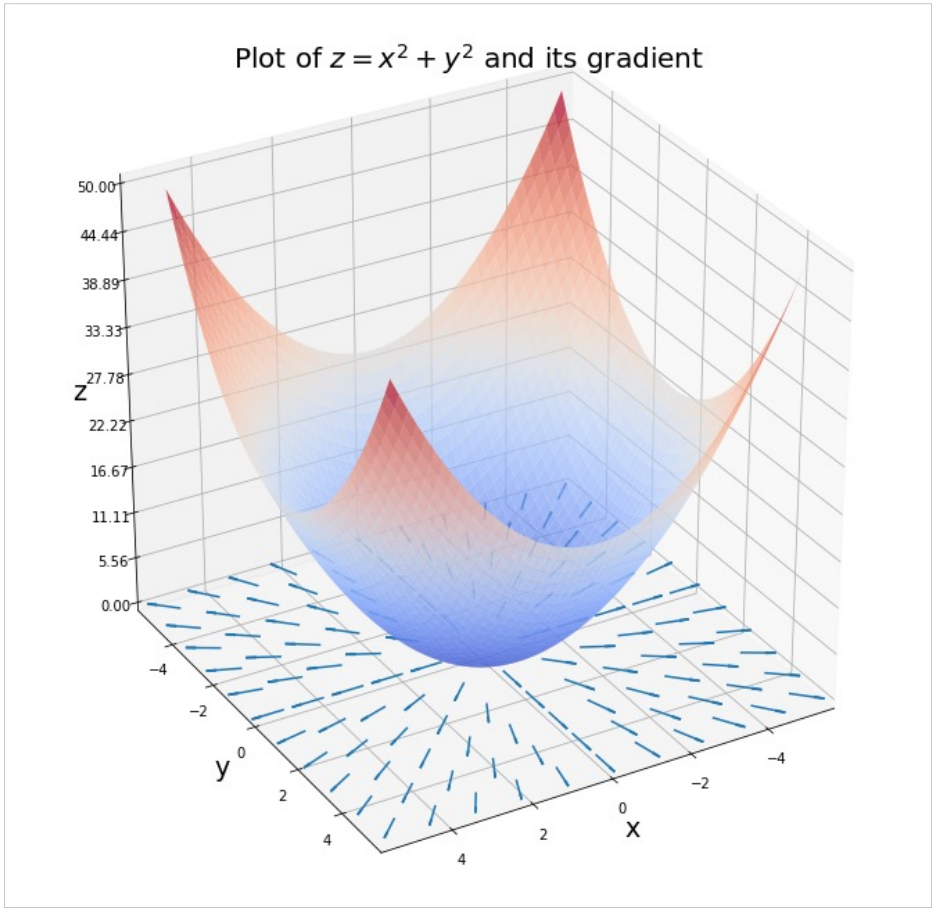
- x_{id} is the i-th component of the d-th input row

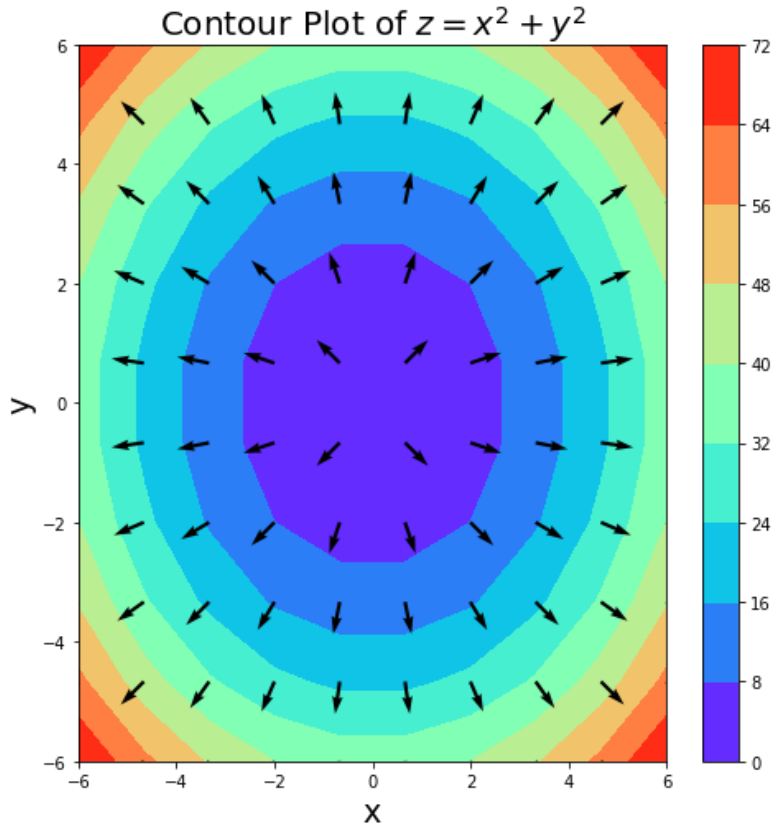
$$\nabla E(w) = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n} \right)$$

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})$$

Gradient of a Function

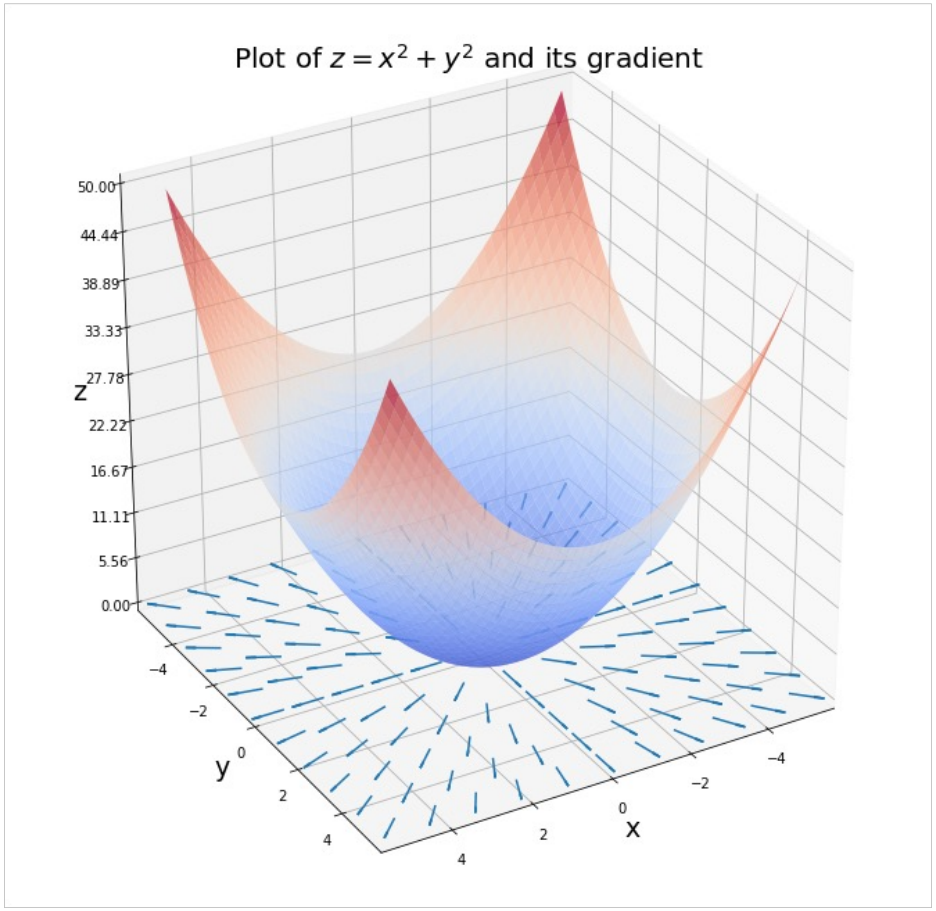
- Remember the gradient is $\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$

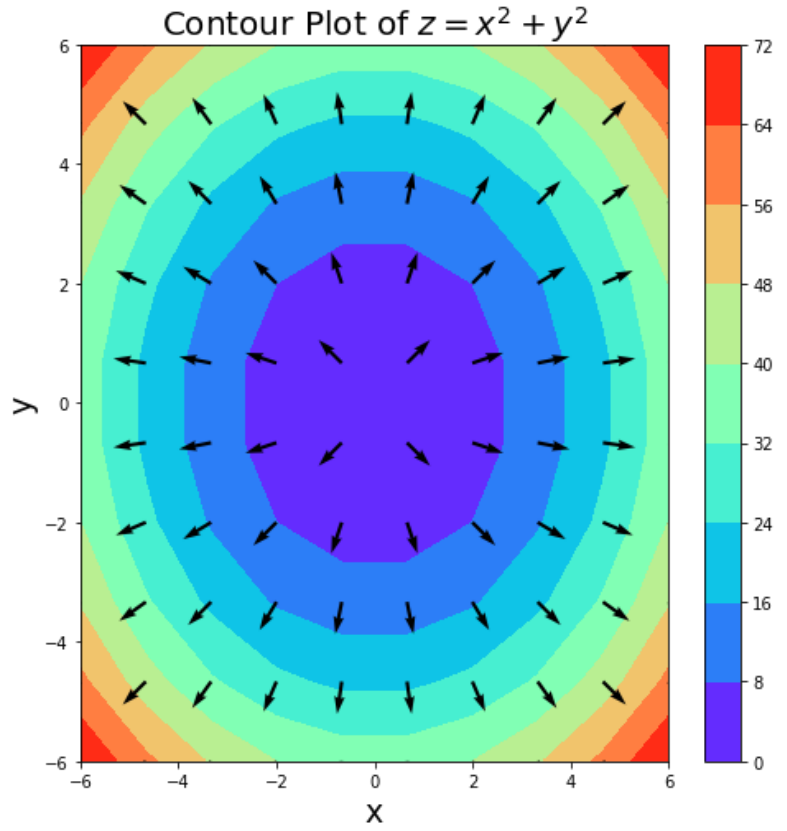




Gradient Descent

- Where does the gradient point? $\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \ldots, \frac{\partial f}{\partial x_n} \right)$





In the direction of steepest ascent

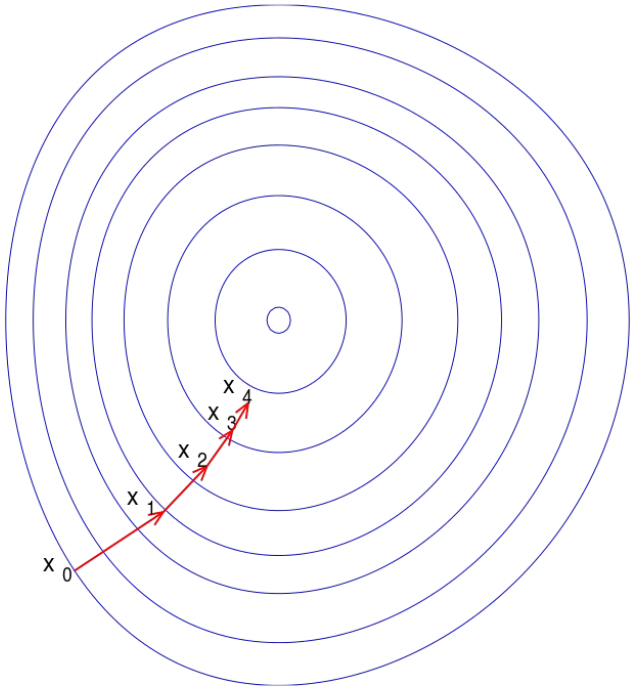
(General) Gradient Descent Algorithm

- **Input**: $f(x) : \mathbb{R}^n \longrightarrow \mathbb{R}, x_0 \in \mathbb{R}^n, \lambda \in \mathbb{R}$ (Initial guess; Learning rate)
- **Output**: A local minimum of f

- For k = 1, 2, …:
  - Compute the gradient ∇f
  - Take the step

$$x_{k+1} = x_k - \lambda(\nabla f)(x_k)$$

  - Test for convergence. If yes, then exit.



The gradient is always normal to the level sets

Gradient Descent Rule for Unthresholded Perceptrons

- Start by specifying a measure for the training loss function:

$$L(w) = E(w) = \frac{1}{2}\sum_{d \in D}(t_d - o_d)^2$$

TRUE LABEL OF EXAMPLE $d \in D$.

OUTPUT OF PERCEPTRON FOR EXAMPLE $d \in D$

In the logistic regression section, we used L for the loss instead of E

- Where D is the set of training samples, t_d is the target output (the "label" in the training set) and o_d is the output of the unthresholded perceptron of the d-th sample.

Goal:

We wish to find the weights $w = (w_1,\ldots,w_n)^T$ that minimize E(w)

Gradient Descent Rule for Unthresholded Perceptrons

We wish to find the weights $w = (w_1,\ldots,w_n)^T$ that minimize $E(w)$ ~ $L(w)$

$$L(w) = E(w) = \frac{1}{2}\sum_{d\in D}(t_d - o_d)^2$$

- Gradient descent says that we need to do:

$$x_{k+1} = x_k - \lambda \nabla f$$

- So, for *E(w)* this assignment looks like:

$$w_{k+1} = w_k - \lambda \nabla E(w) \quad = w_k - \lambda \nabla L(w)$$

- So we need to compute the gradient of *E(w)* with respect to *w*:

$$\nabla L(w) = \nabla E(w) = \left(\frac{\partial E}{\partial w_1}, \frac{\partial E}{\partial w_2}, \ldots, \frac{\partial E}{\partial w_n}\right)$$

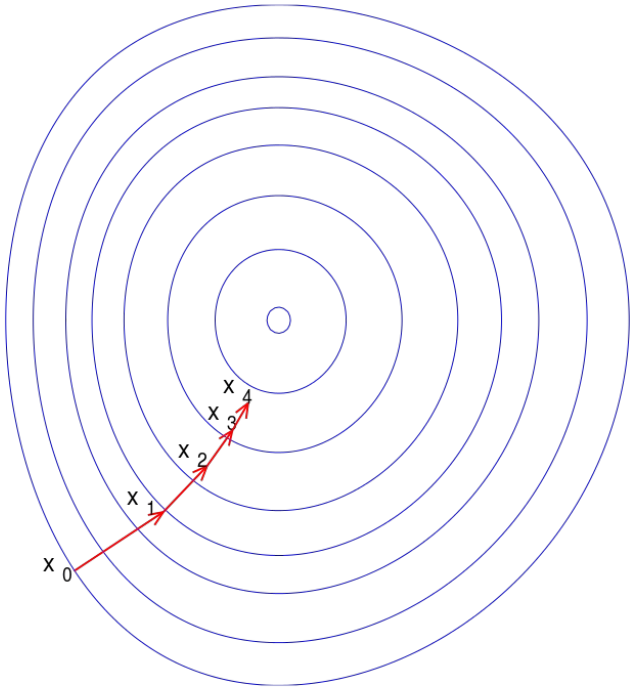
(General) Gradient Descent Algorithm

- **Input**: $f(x) : \mathbb{R}^n \longrightarrow \mathbb{R}, x_0 \in \mathbb{R}^n, \lambda \in \mathbb{R}$ (Initial guess; Learning rate)
- **Output**: A local minimum of f

- For k = 1, 2, …, :
  - Compute the gradient ∇f
  - Take the step

$$x_{k+1} = x_k - \lambda \nabla f$$

  - Test for convergence. If yes, then exit.



The gradient is always normal to the level sets

Gradient Descent Rule for the Unthresholded Perceptron

- **Input**: A training set *D* such that each element is of the form (x_d, t_d) (with t_d being the labels), λ learning rate
- **Output**: A local minimum w of the cost function $E(w)$

- Initialize the weights w_i with small random values
- Until convergence, do:
  - For each i, do $\Delta w_i = 0$
  - For each (x_d, t_d) in D do:
    - Input x_d to the perceptron and obtain o_d
    - For each i, do $\boldsymbol{\Delta w_i} = \Delta w_i + \lambda(t_d - o_d)x_{id}$
  - For each weight i, do: $w_i = w_i + \Delta w_i$

Epoch

Add the contribution of each point to the differential (gradient)

Remember the gradient

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})$$

Remember the gradient descent update rule

$$w_{k+1} = w_k - \lambda \nabla E(w)$$

X_{id} is the i-th component of the d-th input row

Stochastic Gradient Descent Rule for the Unthresholded Perceptron

- **Input**: A training set *D* such that each element is of the form (x_d, t_d) (with t_d being the labels), λ learning rate
- **Output**: A local minimum *w* of the cost function *E(w)*

- Initialize the weights w_i with small random values
- Until convergence, do:
  - For each (x_d, t_d) in *D* do:
    - Input x_d to the perceptron and obtain o_d
    - For each *i*, do $\boldsymbol{w_i} = w_i + \lambda(t_d - o_d)x_{id}$

Epoch

Change the weight using the gradient computed at only one point

Stochastic Gradient Descent

Remember the gradient

$$\frac{\partial E}{\partial w_i} = \sum_{d \in D} (t_d - o_d)(-x_{id})$$

Remember the gradient descent update rule

$$w_{k+1} = w_k - \lambda \nabla E(w)$$

X_{id} is the i-th component of the d-th input row

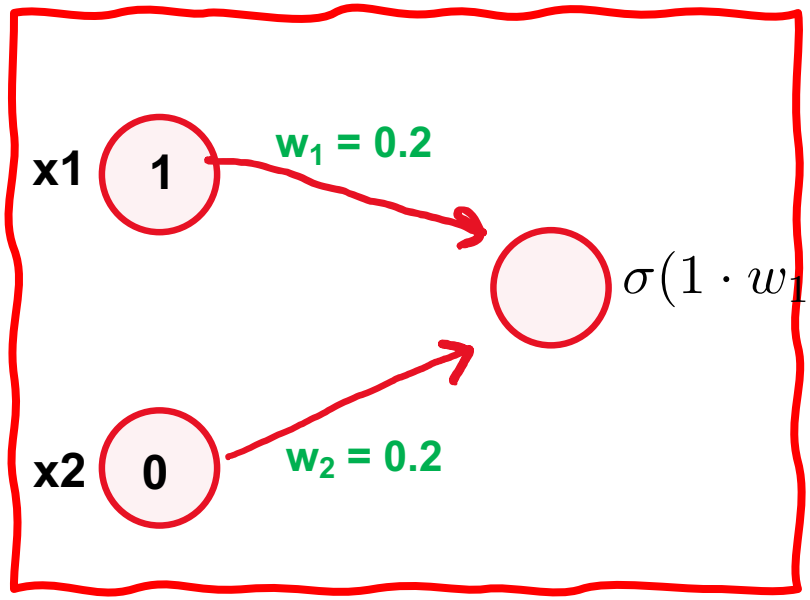
Classifiers

General Neural Networks

Computing the Output of a NN

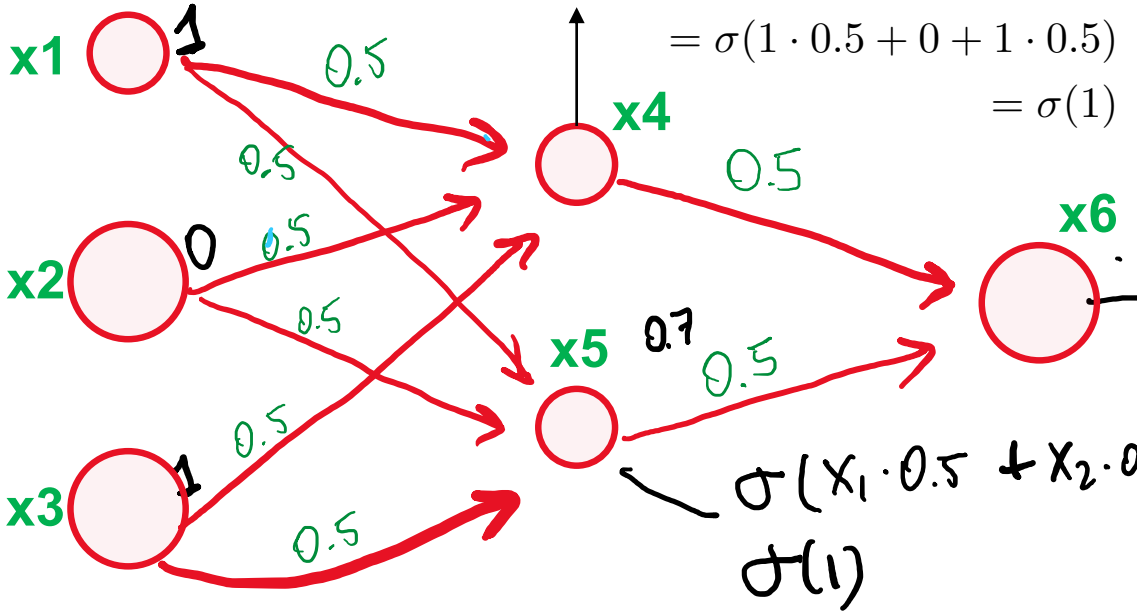
Dataset

x1	x2	y
1	0	0.7
1	1	0.5



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

Computing the Output of a NN

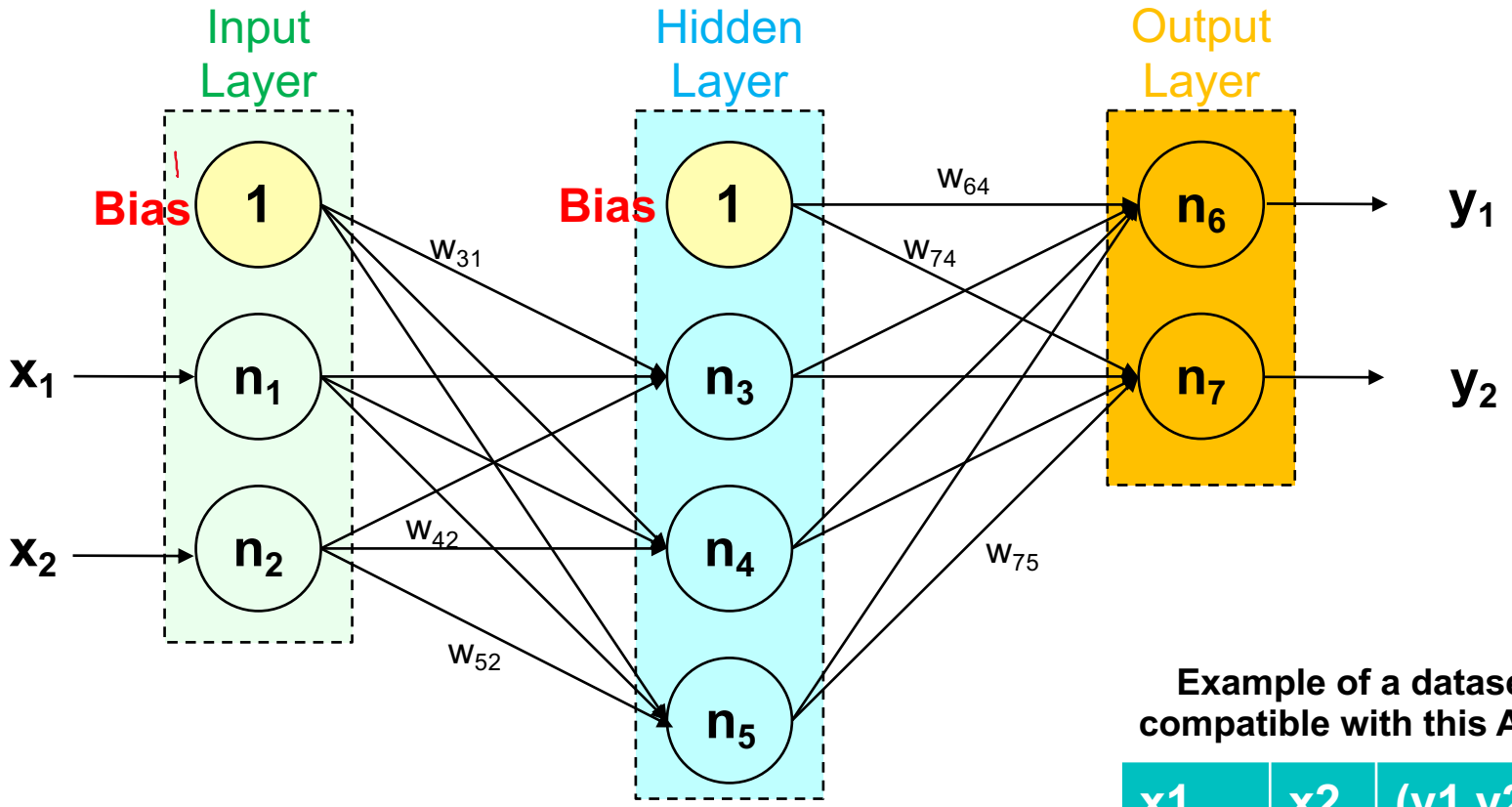


Dataset

x1	x2	x3	y
1	0	1	0.7
1	1	1	0.5

$$\text{Output of } x_4 = \sigma(1) = \frac{1}{1+e^{-1}} = 0.7$$

General Structure of 3-Layer ANNs

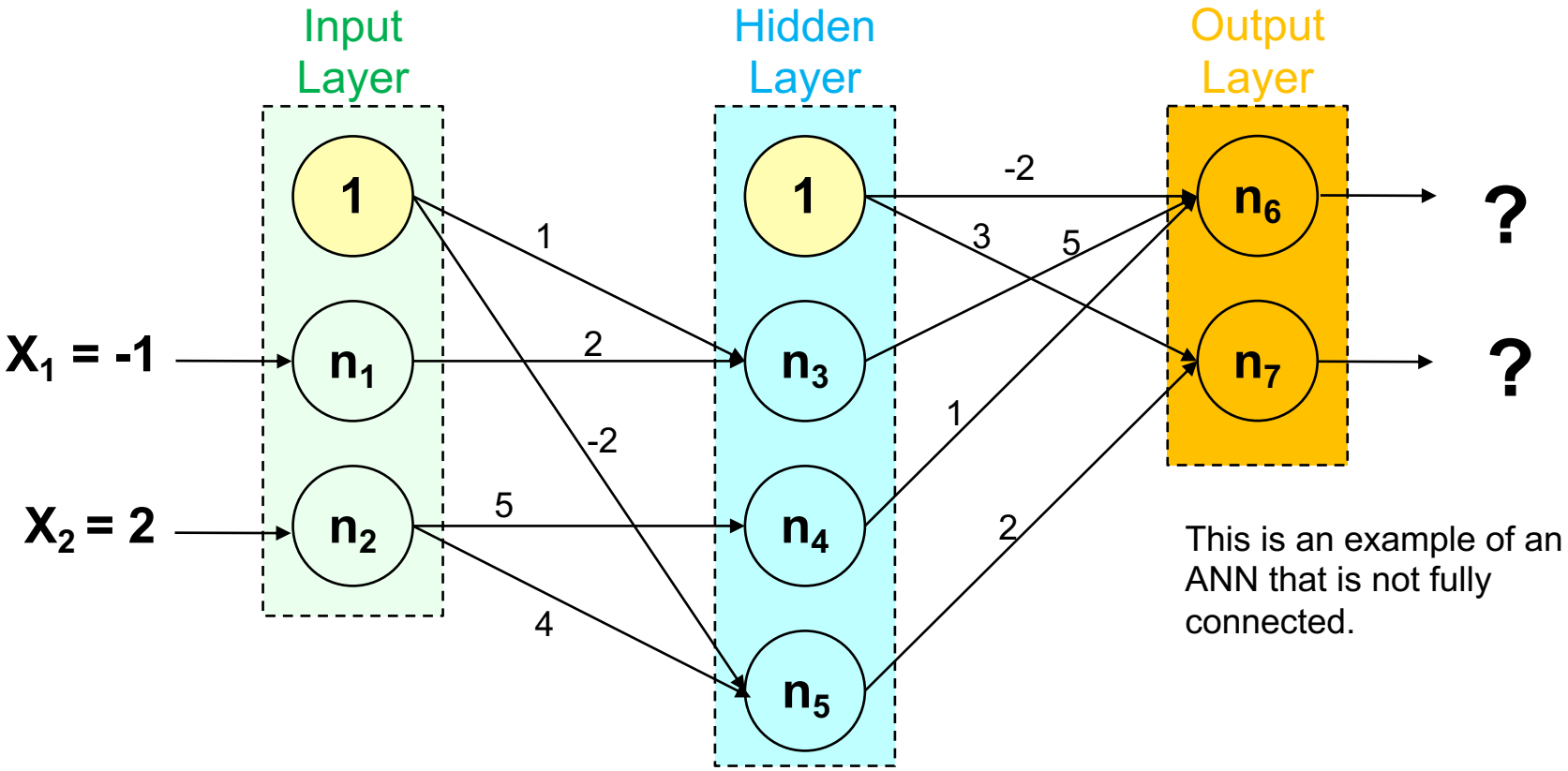


- Different layers can have different numbers of neurons

Example of a dataset compatible with this ANN

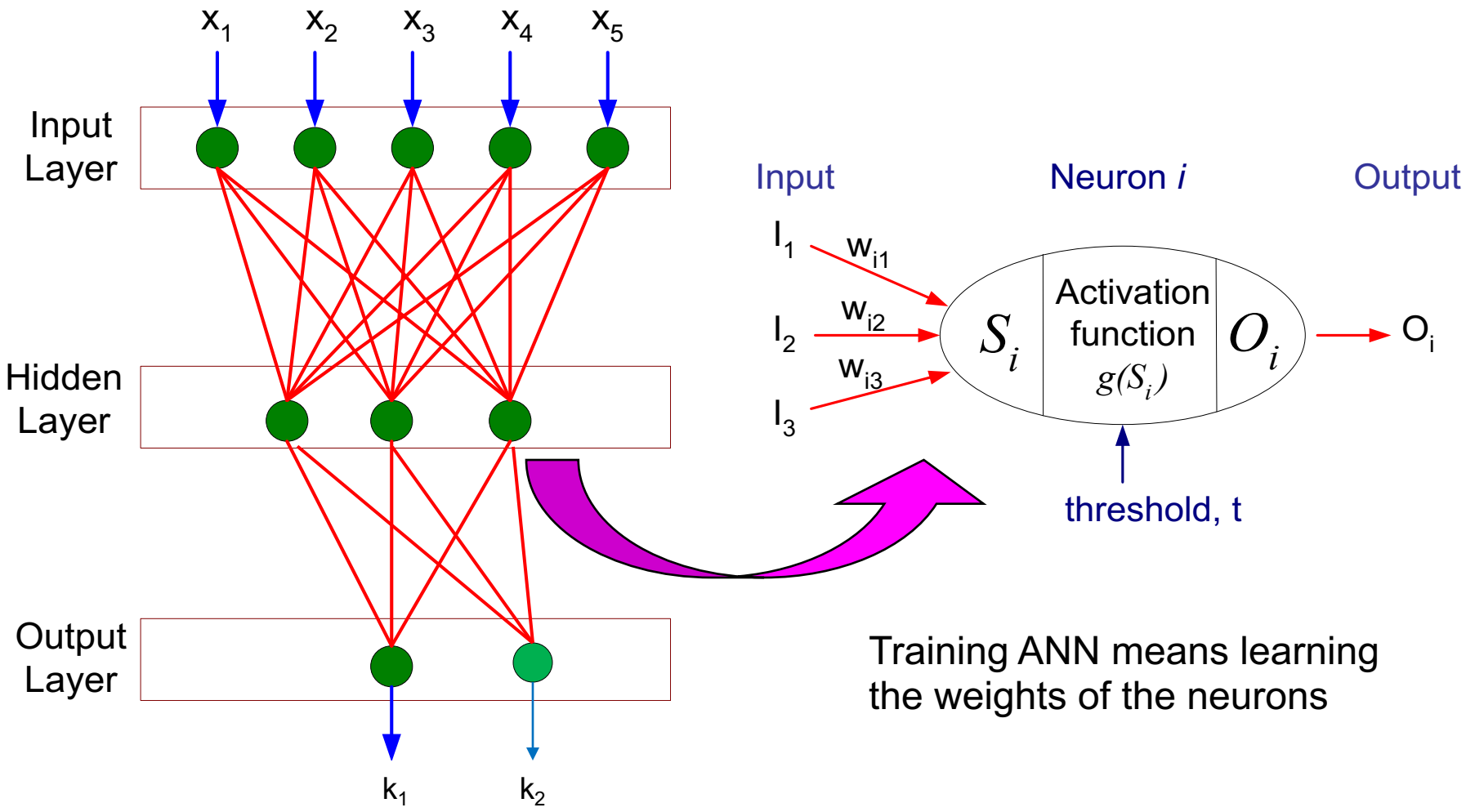
x1	x2	(y1,y2)
1	-3	(1,1)
2	7	(0,1)
-1	5	(1,0)

General Structure of 3-Layer ANNs



- Compute the output for n_6, n_7 given the input x1 = 1, x2 = 2. Assume a sigmoid activation function.

General Structure of 3-layer ANNs



Training ANN means learning the weights of the neurons

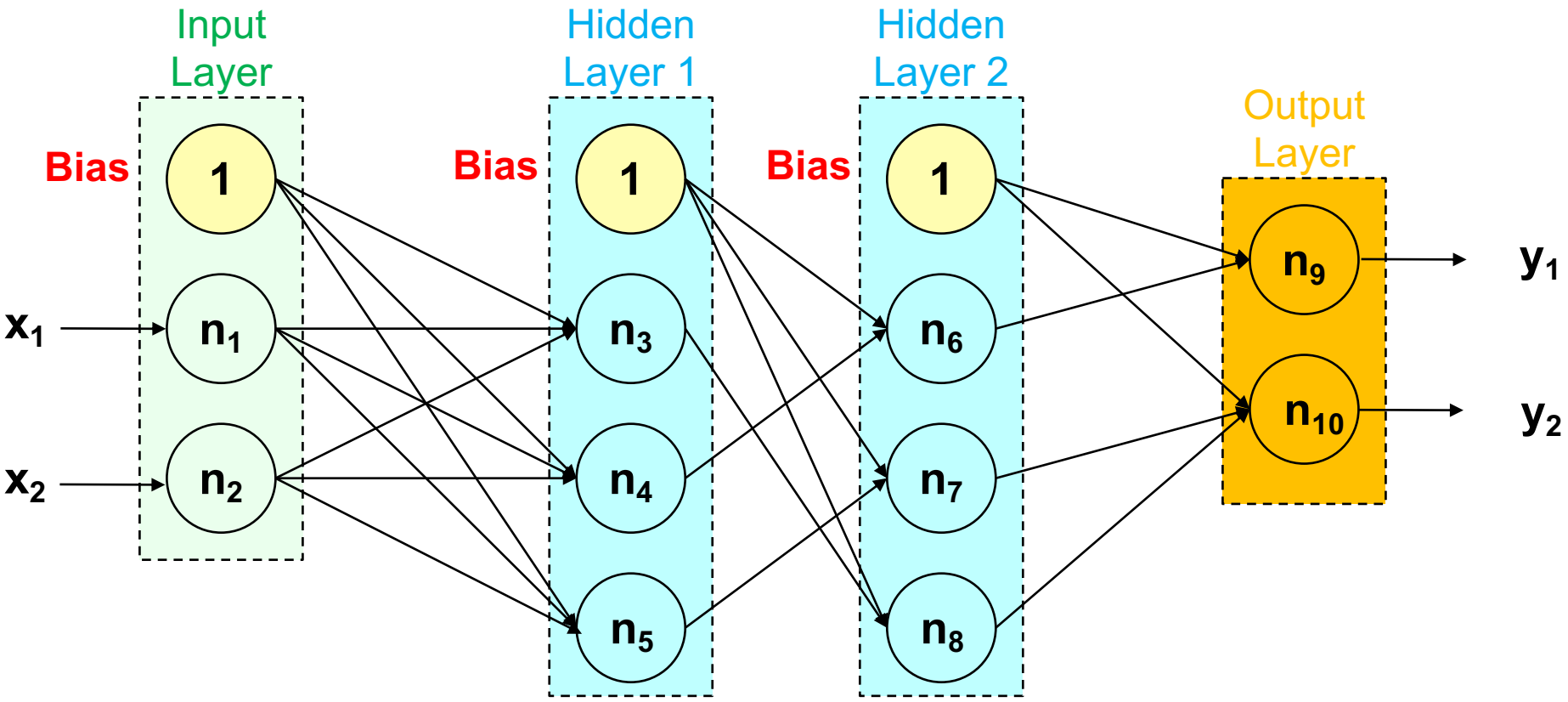
Design Issues in ANN

- Number of nodes in input layer
  - One input node per binary/continuous attribute
  - k or $\log_2$ k nodes for each categorical attribute with k values
- Number of nodes in output layer
  - One output for binary class problem
  - k or $\log_2$ k nodes for k-class problem
- Number of nodes in hidden layer
- Initial weights and biases

Characteristics of ANNs

- Multilayer ANN are **universal approximators** but could suffer from overfitting if the network is too large
- Backpropagation is based on (stochastic) gradient descent; therefore, it may only converge to a local minimum, instead of to a global minimum.
- Model building can be very time consuming, but testing can be very fast
- Sensitive to noise in the training data
- Difficult to handle missing attributes in the training or testing phase

General Structure of N-Layer ANNs



- Neural networks can have many hidden layers

Uses of Neural Networks (MLPs)

- Just like decision trees, neural networks can be used for either
  - Regression
  - Classification
    - Binary
    - Multi-class

One-hot Encoding

- In general, neural networks prefer to have inputs that are numbers. This means that if we have categorical attributes, we have to convert them.
- One way to deal with this problem is to use create dummy variables (one-hot encoding)

One-hot Encoding

- We can represent categorical attributes using a **One-hot Encoding**, also known as *Dummy Variables*.
- This encoding is used in algorithms that require numeric matrices as inputs.
- Each categorical variable with *L* levels is replaced with *L* binary variables.

Two categorical attributes in the dataset

	Pclass <fctr>	Sex <fctr>	Age <dbl>
1	3	male	22
2	1	female	38
3	3	female	26
4	1	female	35
5	3	male	35

5 rows

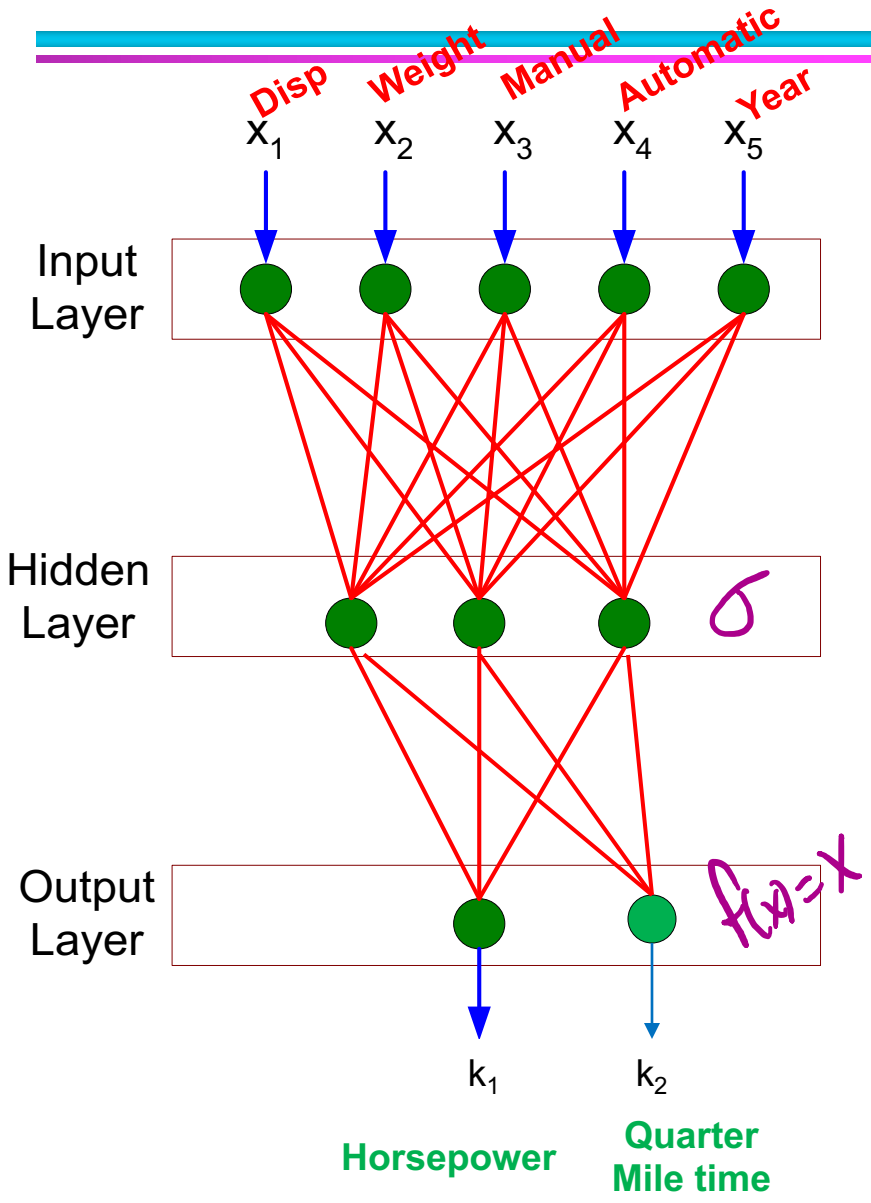
Let's make dummy variables with the Pclass and the Sex attributes:

```{r}
simpleMod <- dummyVars(~., data = trainX, levelsOnly = TRUE)
head(predict(simpleMod, trainX))
```

	1	2	3	female	male	Age
1	0	0	1	0	1	22
2	1	0	0	1	0	38
3	0	0	1	1	0	26
4	1	0	0	1	0	35
5	0	0	1	0	1	35
7	1	0	0	0	1	54

Pclass has 3 levels: 1, 2, and 3, so it is replaced with variables 1, 2, 3.
Since first row has a value of 3 for attribute 'Pclass' then that row has a value of 1 for the new attribute '3'.

Regression MLPs



- Example problem:
  - Predict the horsepower and quarter mile time of a car, given its engine displacement, weight, type of transmission, etc.
- The output needs to be a number: any number. So, it needs **no special activation function** in the output.
- The **loss function** can be **MSE or MAE**

Parenthesis: MSE vs. SSE vs. MAE

MSE = Mean Squared Error

$$\frac{1}{n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

$\hat{y}_i$: Output of the model

y_i: Observed Value

SSE = Sum Squared Error

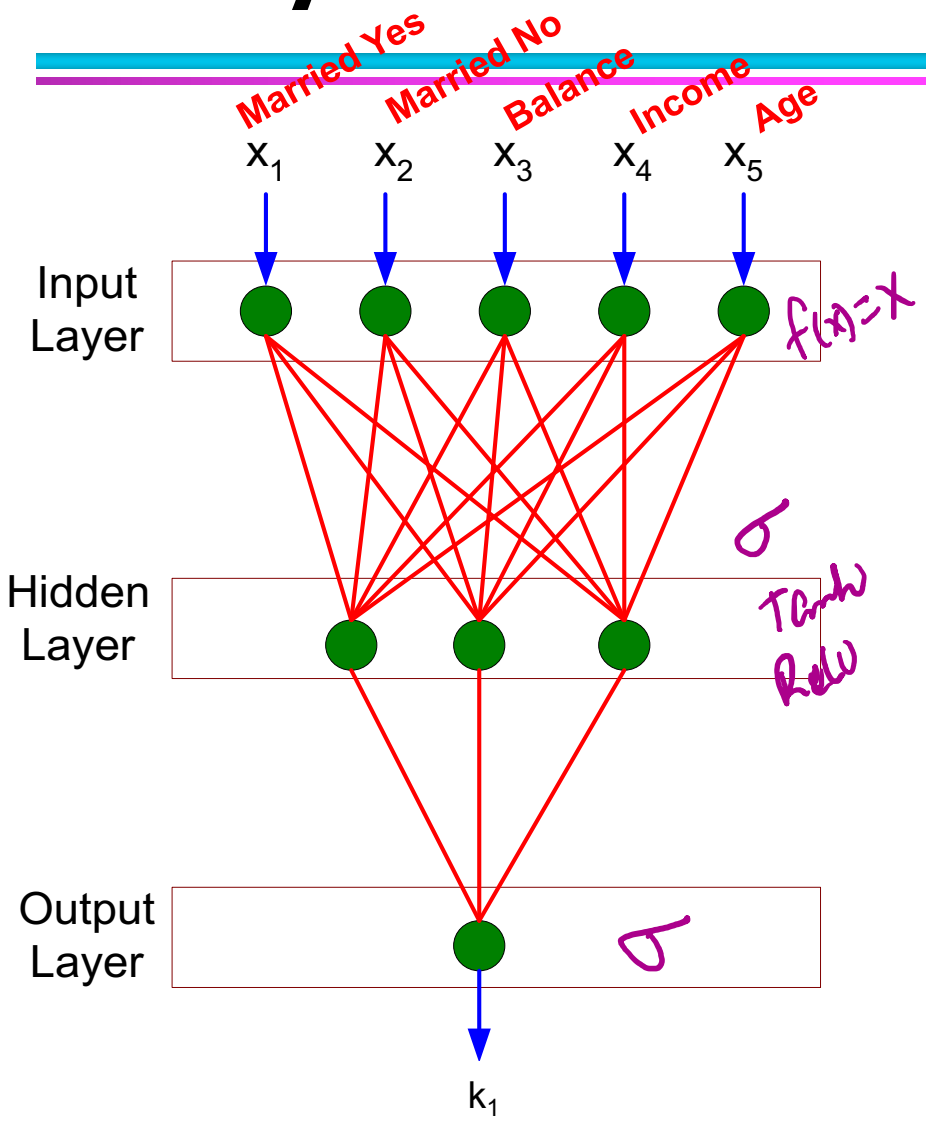
$$\sum_{i=1}^{n}(\hat{y}_i - y_i)^2$$

MAE = Mean Absolute Error

$$\sum_{i=1}^{n}|\hat{y}_i - y_i|$$

Note: Take the average of this

Binary Classification MLPs



- Example problem:
  - Predict of a person is likely to default, given his/her balance, income, etc.

id	Married	income	balance	default
1	No	23,000	2500	no
2	Yes	57,000	535	yes
3	Yes	29,000	3000	yes

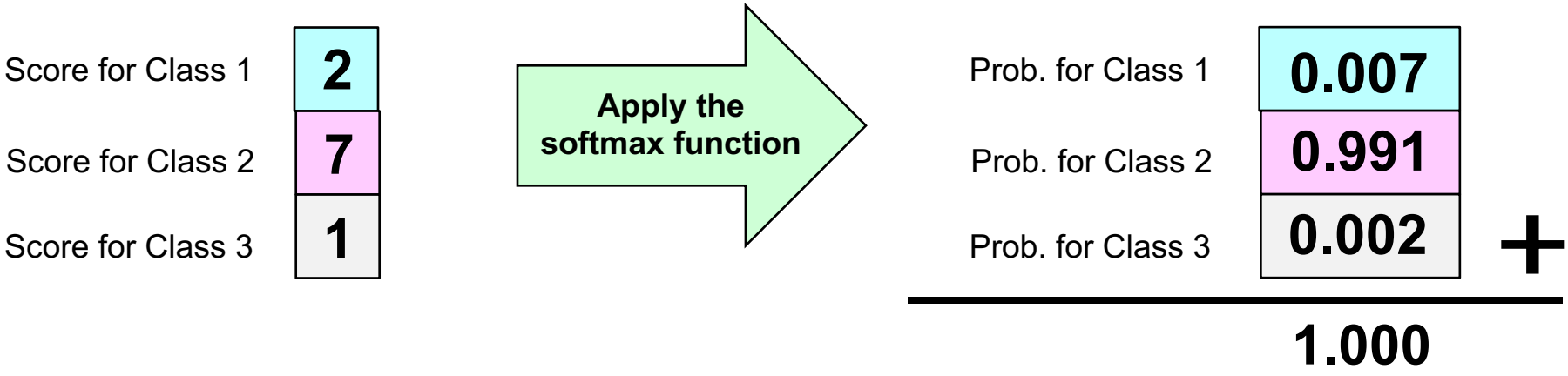
- The output needs to be a number from 0 to 1.
- The **output activation function** can be the **sigmoid**
- The **loss function** can be the **binary cross entropy**

Softmax Function

- The **softmax function**, also called the Gibbs function, is defined as follows

$$s : (x_1, x_2, \ldots, x_n) \mapsto \frac{1}{e^{x_1} + e^{x_2} + \cdots + e^{x_n}} (e^{x_1}, e^{x_2}, \cdots, e^{x_n})$$

- It is useful to convert logits (scores) into probabilities



Softmax Function Implementation

$$s : (x_1, x_2, \ldots, x_n) \mapsto \frac{1}{e^{x_1} + e^{x_2} + \cdots + e^{x_n}} (e^{x_1}, e^{x_2}, \cdots, e^{x_n})$$

- The softmax function is very simple to implement:

R Implementation

```r
softmax <- function(x) {
  exp(x) / sum(exp(x))
}
```

```r
x <- c(2, 7, 1)
softmax(x)
```

```
[1] 0.006676413 0.990867473 0.002456115
```

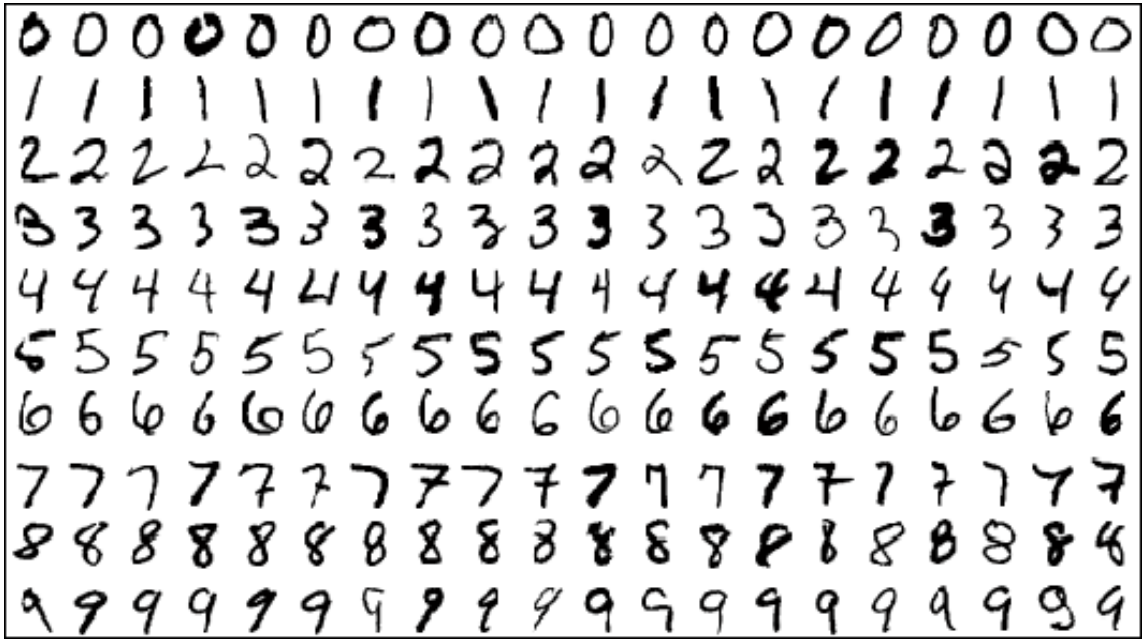
Python Implementation

```
In [7]: import numpy as np

In [8]: def softmax(x):
   ...:     return np.exp(x) / np.sum(np.exp(x))
   ...:

In [9]: softmax(np.array([2., 7., 1.]))
Out[9]: array([0.00667641, 0.99086747, 0.00245611])
```

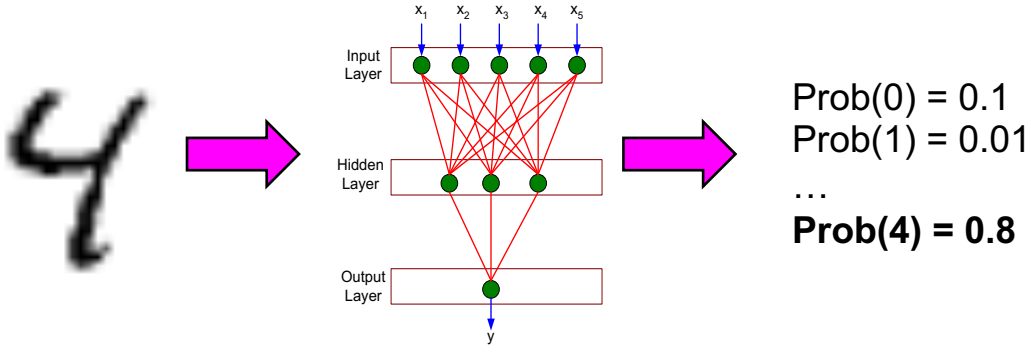
Notice that these functions were written so that they accept vectors as input and return them as output. This is called vectorization.

Parenthesis: Handwritten Digit Recognition

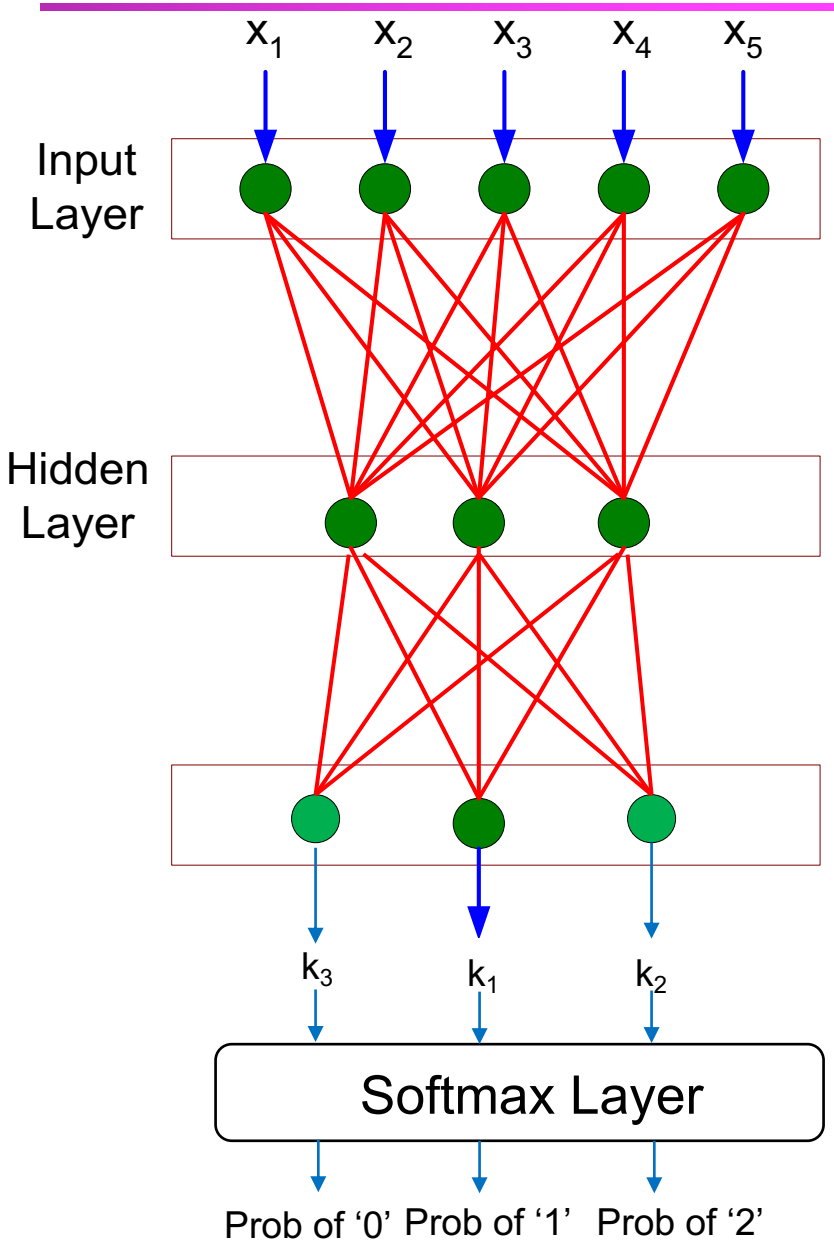
- Given a labeled training data set with handwritten digits,



Predict the actual number

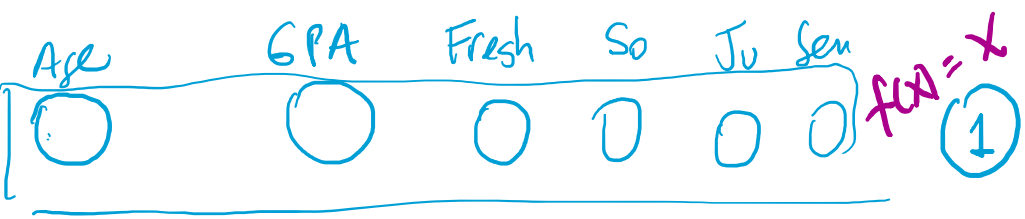


Multi-class Classification MLPs

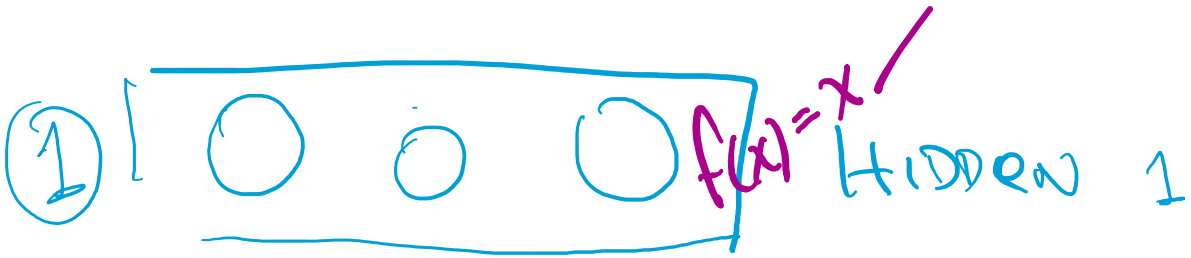


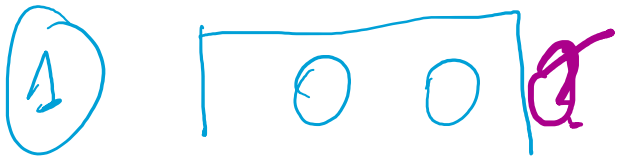
- Example problem:
  - Handwritten digit recognition. Given a picture of a handwritten digit from 0 to 9, predict which it is.
- The output can be a 10 numbers between 0 to 1, indicating the probability of each digit.
- The **output activation function** can be the **softmax** function.
- The **loss function** can be the categorical **cross entropy (categorical cross entropy)**

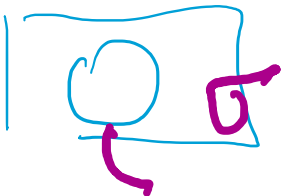
EXAMPLE)



Age	GPA	Class	Pass
12	3.1	Freshman	Y
⋮	⋮	⋮	⋮
13	3.3	Soph	?

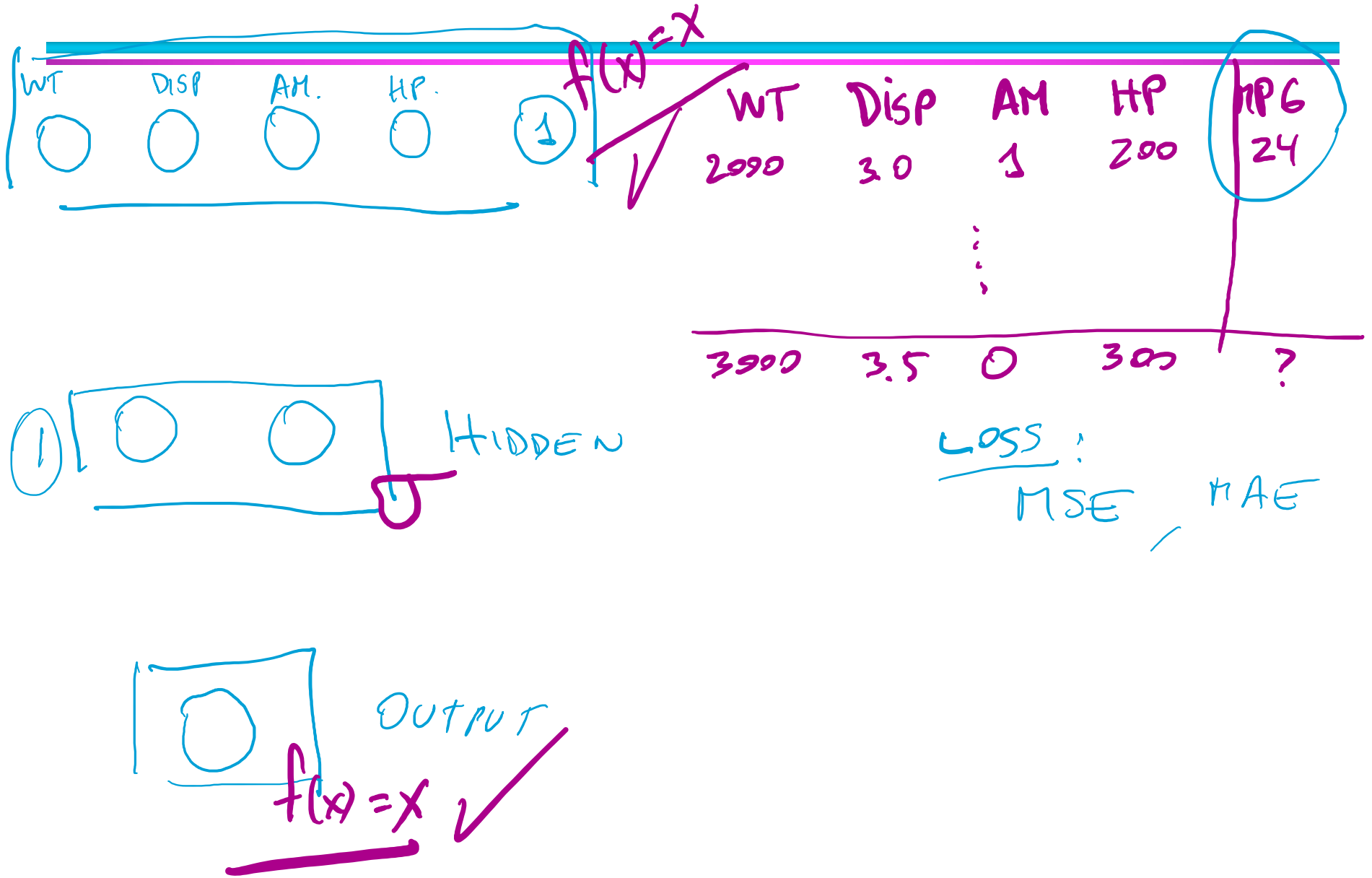






LOSS ?

BINARY CROSS ENTROPY


WT
DISP
AM.
HP.
1
f(x)=x
WT DISP AM HP MPG
2990 3.0 1 200 24
3900 3.5 0 300 ?
1
HIDDEN
LOSS :
MSE MAE
OUTPUT
f(x)=x

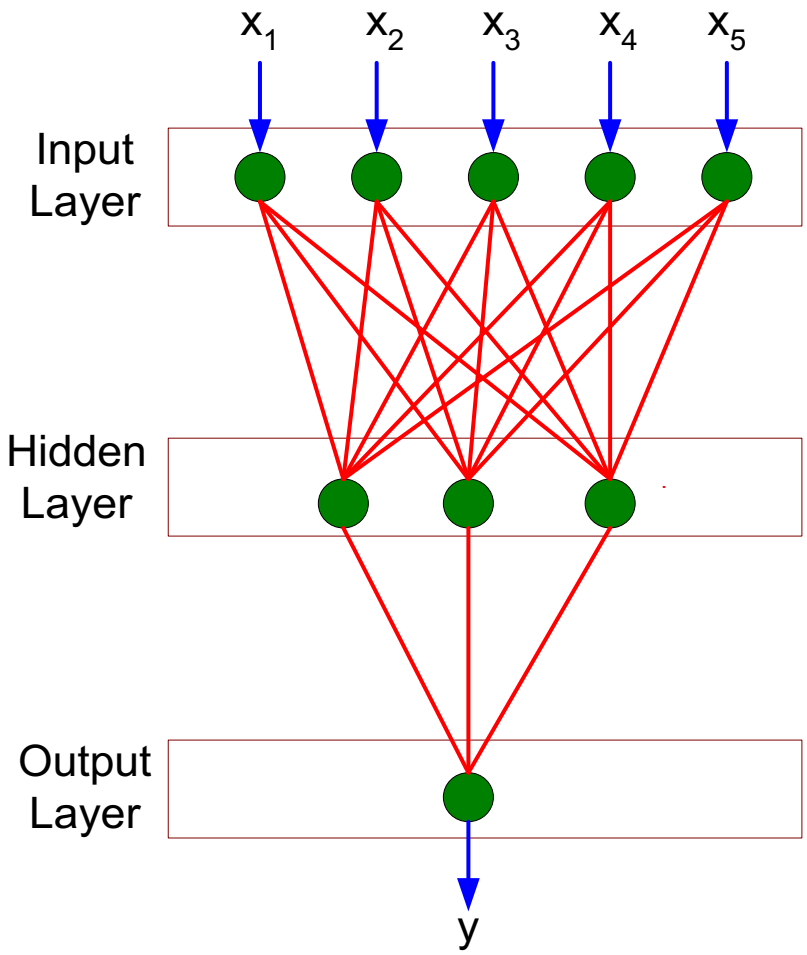
ANN Structures per Problem

Hyperparameter	Binary Classification	Multi-class Classification	Regression
Number of input neurons	One per variable	One per variable	One per variable
Number of output neurons	1	1 per class	Dimension of the output
Output activation function	Sigmoid	Softmax	None (a.k.a. linear), relu, elu, selu,
Loss function	Cross Entropy	Cross Entropy	MSE, MAE

Loss Functions vs. Performance Metrics

Loss Functions	Performance Metric
Used to train / fit the model	**Used to evaluate the model after it has been trained / fit**
Ideally if it is differentiable and well-behaved numerically (because we minimize the loss by taking its gradient in terms of the parameters of the model, as we did for linear regression)	Does not need to be differentiable or numerically well-behaved
Does not need to be easily interpretable or intuitive	Needs to be easily interpretable and intuitive
Examples: SSE (sum of the squared of the errors), MSE (mean squared error), Huber, Binary Cross Entropy, Categorical Cross Entropy, etc.	Examples: Accuracy, MAE (mean absolute error), MAPE (mean absolute percentage error), RMSE (root mean squared error), Binary Cross Entropy, Categorical Cross Entropy, Precision, Recall, AUC (area under the curve), KL Divergence, etc.

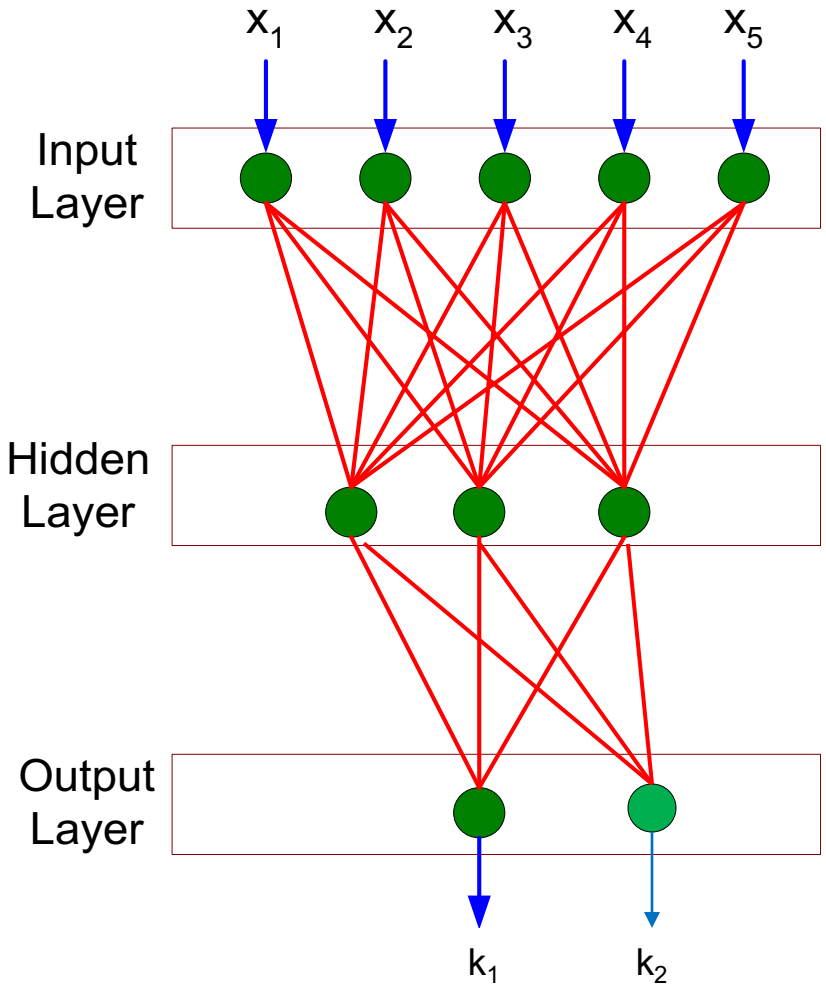
Neural Networks in R Keras: Binary Classification and Multi-Class

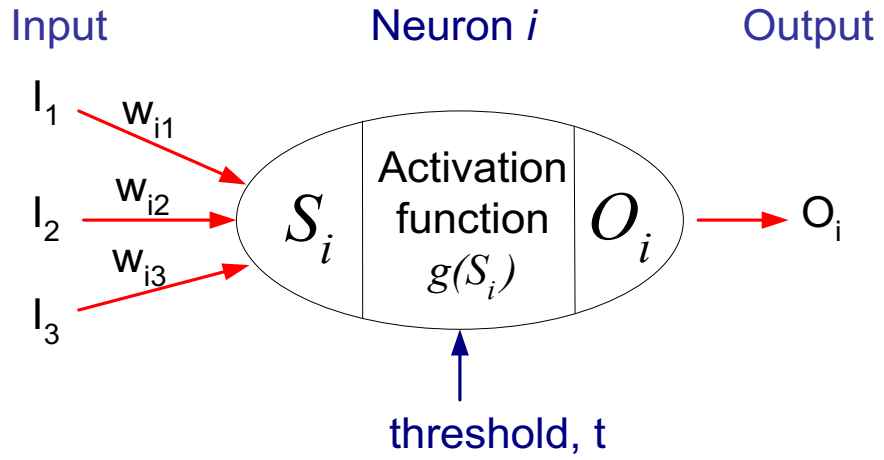



```
# Defines the structure of the NN
model <-
  keras_model_sequential() %>%
  layer_dense(units = 3,
              input_shape = 5,
              activation = 'sigmoid') %>%
  layer_dense(units = 1,
              activation = 'sigmoid')

# Specifies how training will be done
model %>% compile (
  loss = 'binary_crossentropy',
  metrics = c('accuracy'),
  optimizer = optimizer_sgd(lr = 0.01))

# Begins training
model %>% fit(
  train_x, train_y,
  epochs = 10,
  batch_size = 32,
  validation_split = 0.2
)
```

The final layer uses the '**sigmoid**' activation since we assume that we are doing binary classification. If we were doing linear regression, then either replace 'sigmoid' by 'linear', or just remove the activation parameter to the function. Write '**softmax**' instead for multi-class classification

Backpropagation for ANNs





Now the error is defined so that it takes into consideration the different outputs:

$$E(w) = \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

This is the error in the d-th example:

$$E_d = \frac{1}{2} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

Backpropagation Algorithm for ANNs

- **Input**: A training set *D* such that each element is of the form (x_d, t_d) (with t_d being the labels), λ learning rate
- **Output**: A local minimum *w* of the cost function *E(w)*

Remember the gradient descent update rule

$$w_{k+1} = w_k - \lambda \nabla E(w)$$

- Create a feed-forward ANN with n_{in} inputs, n_{hidden} hidden units and n_{out} output units
- Initialize the weights w_i with small random values
- Until convergence, do:
  - For each *(x, t)* in *D* do:
    - Input *x* to the ANN and obtain the output o_k for every unit *k*
    - For each output unit *k*, calculate the error term δ_k: $\delta_k = o_k(1 - o_k)(t_k - o_k)$
    - For each hidden unit *h*, calculate the error term δ_h:

$$\delta_h = o_h(1 - o_h) \sum_{k \in outputs} (w_{kh}\delta_k)$$

    - For each weight *ij*, do: $w_{ji} = w_{ji} + \Delta w_{ji}$ where $\Delta w_{ji} = \lambda \delta_j x_{ji}$

epoch.

Backpropragation part

For *j* in the output layer:

$$\delta_j = -\frac{\partial E_d}{\partial net_j} = (t_j - o_j)o_j(1 - o_j)$$

For *j* in the hidden layer:

$$\delta_j = o_j(1 - o_j) \sum_{k \in Downstream(j)} \delta_k w_{kj}$$

Deep Neural Networks

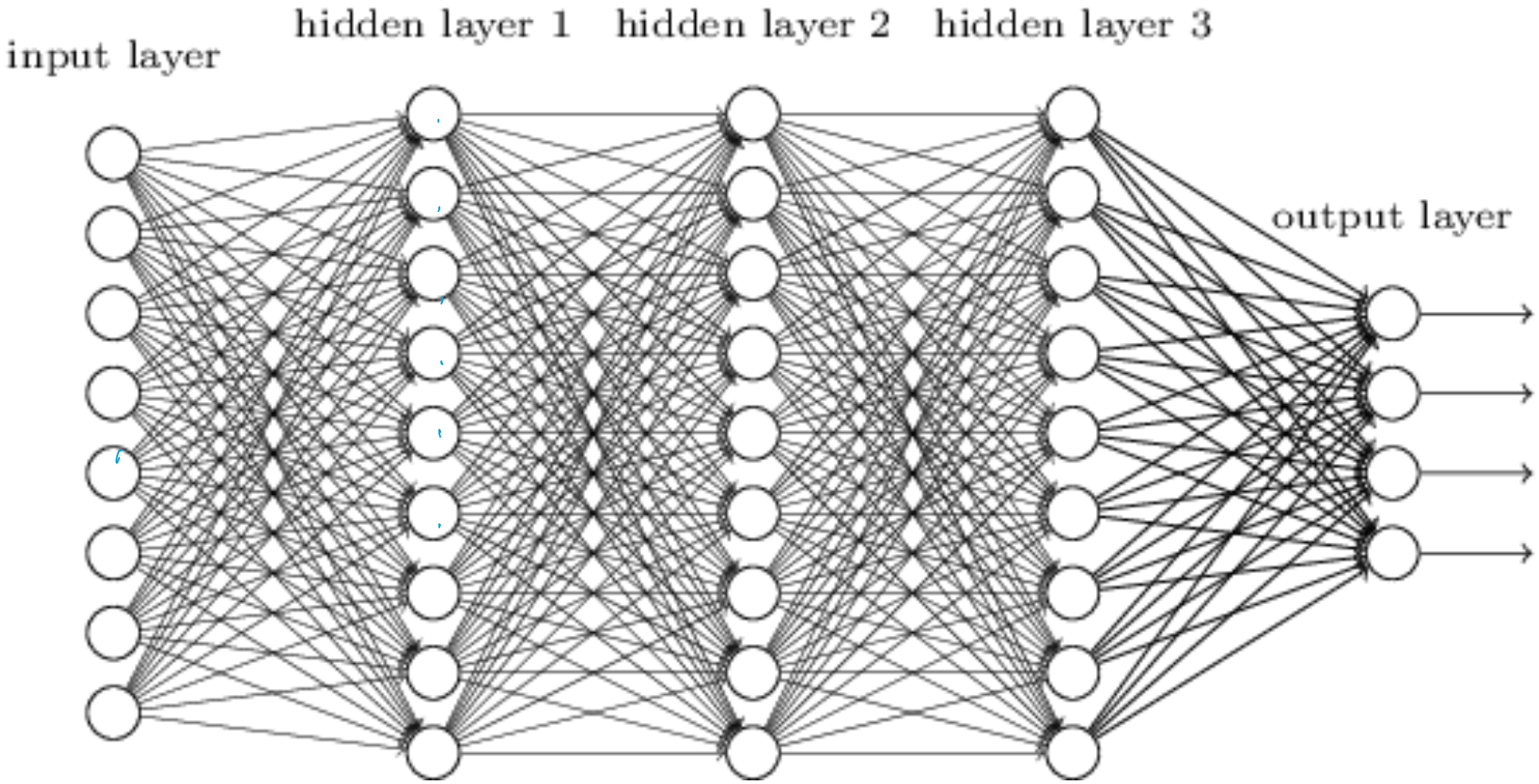


Figure from http://neuralnetworksanddeeplearning.com/chap6.html

Vanishing Gradient Problem

- Suppose a deep neural network with the structure

- The output of unit *i* is *oi* and w_i is the weight from the *i*-th unit to the
- We also use the sigmoid function and the same loss function we used in backpropagation

$$E_d = \frac{1}{2} \sum_{k \in outputs} (t_{kd} - o_{kd})^2$$

- Using the composition rule for derivatives, we can prove that

$$\frac{\partial E_d}{\partial o_t} = \sigma'(o_{t+1}) \cdot w_{t+1} \cdot \frac{\partial E_d}{\partial o_{t+1}}$$

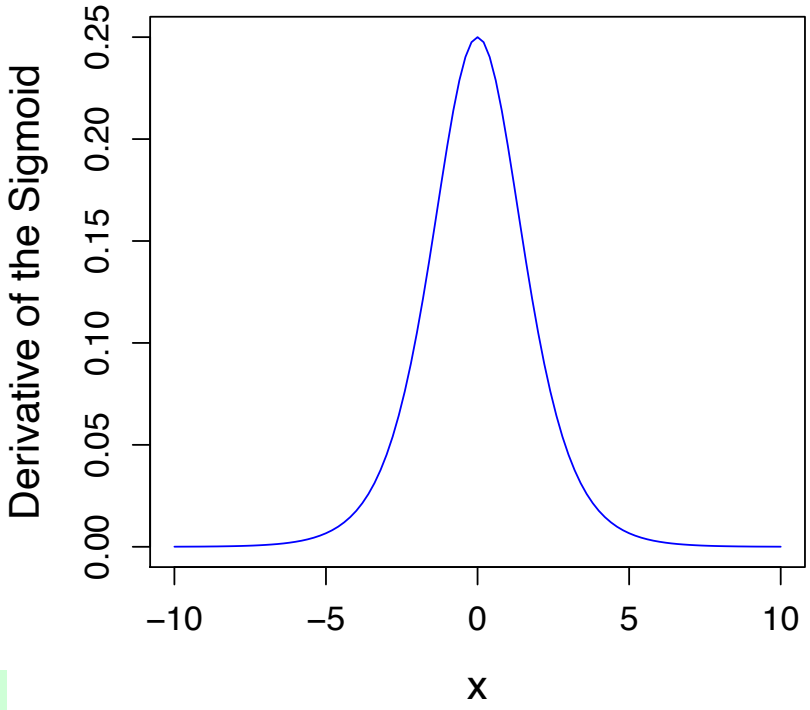
Vanishing Gradient Problem

- Suppose a deep neural network with the structure

$$\frac{\partial E_d}{\partial o_t} = \sigma'(o_{t+1}) \cdot w_{t+1} \cdot \frac{\partial E_d}{\partial o_{t+1}}$$

- **The derivative of the sigmoid is bounded above by 0.25**
- **The weights can be normalized to be less than 1**
- **This means that the gradient vanishes for the first layers:**

$$\frac{\partial E_d}{\partial o_{t-k}} = |c|^k \cdot \frac{\partial E_d}{\partial o_t}$$



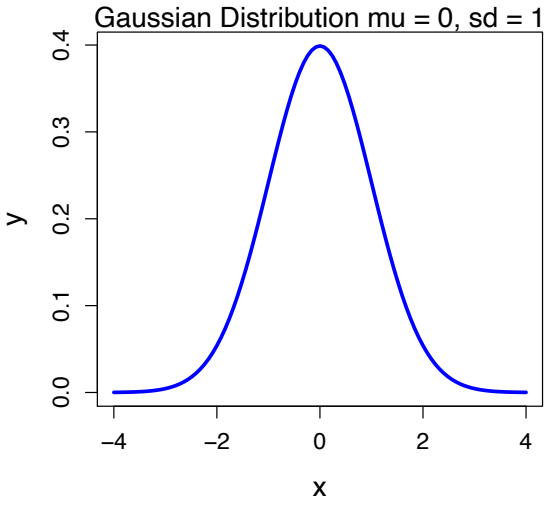
This is one of the reasons why training DNNs with simple backpropagation is very slow

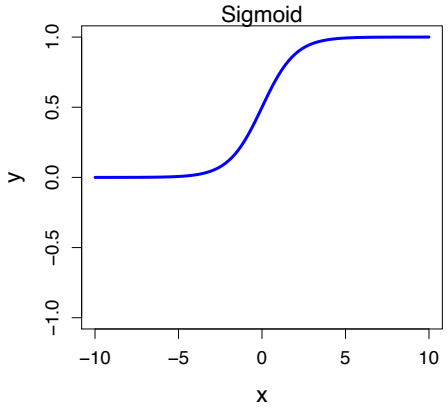
Vanishing Gradient Problem

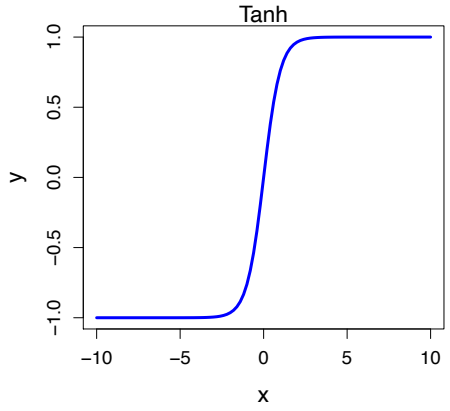
- Glorot and Bengio (2010) found that the vanishing gradient problem has other causes as well:
  - Poor initialization functions:
    - Initializing weights by sampling from a Gaussian with N(0,1)
    - Initializing weights by sampling from a uniform distribution:

$$W_{ij} \sim U\left[-\frac{1}{\sqrt{n}}, \frac{1}{\sqrt{n}}\right]$$

  - Poor activation functions
    - The predominant activation function was the sigmoid or the tanh







Glorot and Bengio, “Understanding the Difficulty of Training Deep Feedforward Neural Networks”, 2010

Dealing with Slow Training in DNNs

- Training deep neural networks can be very slow
- There are several ways to speed up DNN training:
  - Use faster optimizers
  - Use good activation functions
  - Use good weight initialization strategies
  - Use batch normalization
  - Use pre-trained networks

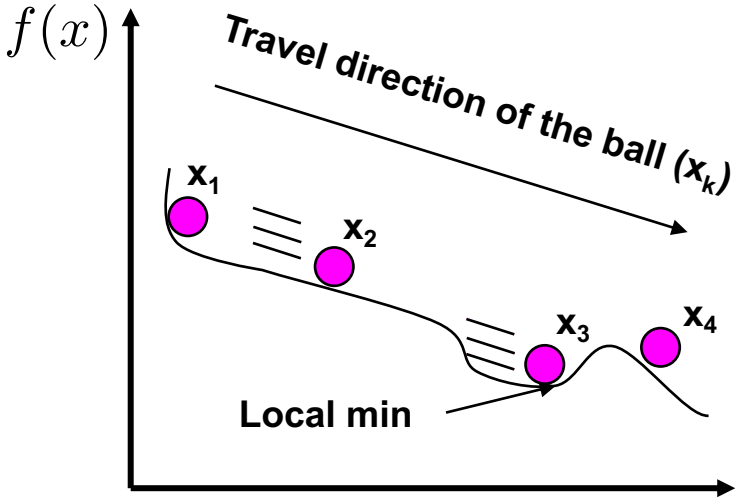
Using Faster Optimizers

- These optimizers can be faster than regular gradient descent:
  - Momentum Optimization (1964)
  - Nesterov Accelerated Gradient (1983)
  - AdaGrad (2011)
  - RMSProp (2012)
  - Adam Optimization (2015)
  - Nadam Optimization (2016)

B. Polyak, ”Some Methods of speeding up convergence of iteration methods”, 1964
Y. Nesterov, “A method for Unconstrained Convex Minimization Problem with the Rate of Convergence $O(1/k^2)$”, 1983
J. Duchi et al. “Adaptive Subgradient Methods for Online Learning and Stochastic Optimization”, 2011
Tiejmen Tieleman and Geoffrey Hinton. Coursera Class, 2012.
D. Kingma, J. Ba, “Adam: A Method for Stochastic Optimization”, 2015
T. Dozat. “Incorporating momentum optimization into Adam”, 2016

Momentum Optimization

- **Input**: $f(x) : \mathbb{R}^n \to \mathbb{R}, x_0 \in \mathbb{R}^n, \lambda \in \mathbb{R}, \beta \in (0, 1)$ (Initial guess; Learning rate; Momentum Parameter)
- **Output**: A local minimum of f



Momentum gives the "bowling ball" speed so that it does not get stuck in local minima

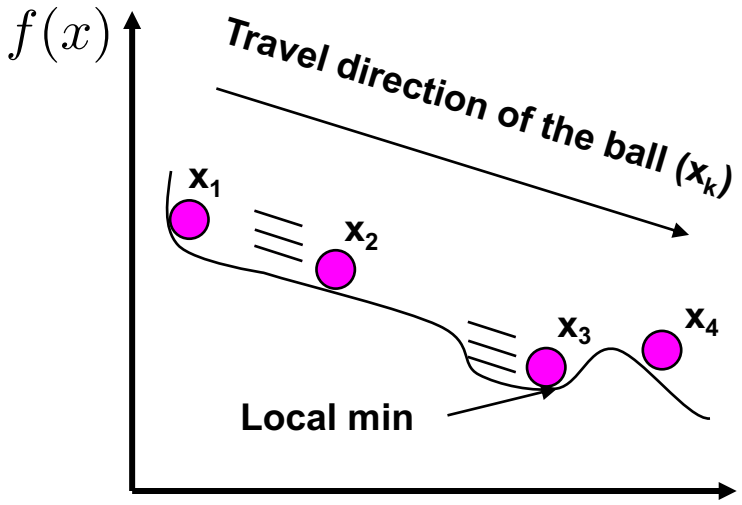
- Incorporate old gradient values in the computation of the next step
- Helps avoid getting stuck in local minima

Figure adapted from Charu C. Aggarwal. Neural Networks and Deep Learning. Springer 2018

Momentum Optimization

- **Input**: $f(x) : \mathbb{R}^n \to \mathbb{R}, x_0 \in \mathbb{R}^n, \lambda \in \mathbb{R}, \beta \in (0,1)$ (Initial guess; Learning rate; Momentum Parameter)
- **Output**: A local minimum of f

- For k = 1, 2, …:
  - Compute the gradient $\nabla_{x_k}(f)$
  - Update the momentum
    $$m_{k+1} = \beta m_k - \lambda \nabla_{x_k}(f)$$
  - Take the step
    $$x_{k+1} = x_k + m_k$$
  - Test for convergence. If yes, then exit.



Momentum gives the "bowling ball" speed so that it does not get stuck in local minima

Figure adapted from Charu C. Aggarwal. Neural Networks and Deep Learning. Springer 2018

Momentum Optimization

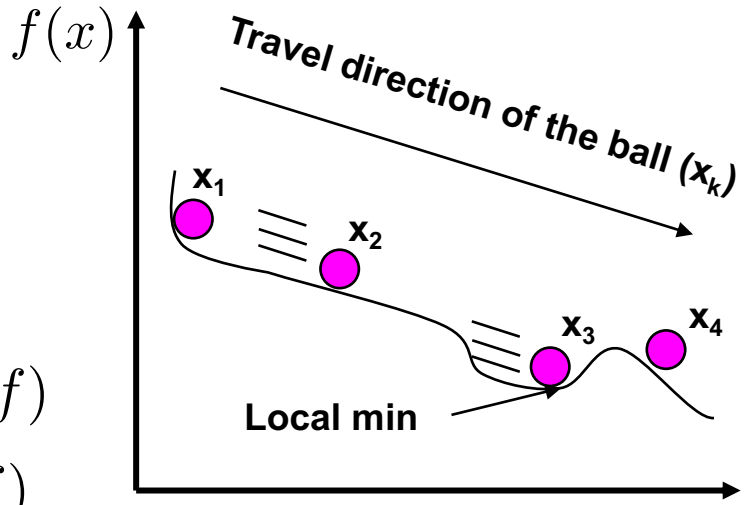
Input: $f(x) : \mathbb{R}^n \to \mathbb{R}, x_0 \in \mathbb{R}^n, \lambda \in \mathbb{R}, \beta \in (0, 1)$

Initial guess | Learning rate | Momentum Parameter

Output: A local minimum of f

$$m_{k+1} = \beta m_k - \lambda \nabla_{x_k}(f)$$
$$x_{k+1} = x_k + m_k$$

Momentum Optimization Update Rules



Momentum gives the "bowling ball" speed so that it does not get stuck in local minima

$$\begin{aligned} m_{k+1} &= \beta m_k - \lambda \nabla_{x_k}(f) \\ &= \beta(\beta m_{k-1} - \lambda \nabla_{x_{k-1}}(f)) - \lambda \nabla_{x_k}(f) \\ &= \beta^2 m_{k-1} - \beta\lambda \nabla_{x_{k-1}}(f) - \lambda \nabla_{x_k}(f) \\ &= \cdots \\ &= \beta^k m_1 - \beta\lambda \nabla_{x_{k-1}}(f) - \lambda \nabla_{x_k}(f) \\ &= \beta^k m_1 - \beta^{k-1}\lambda \nabla_{x_1}(f) - \beta^{k-2}\lambda \nabla_{x_2}(f) + \cdots - \lambda \nabla_{x_k}(f) \end{aligned}$$

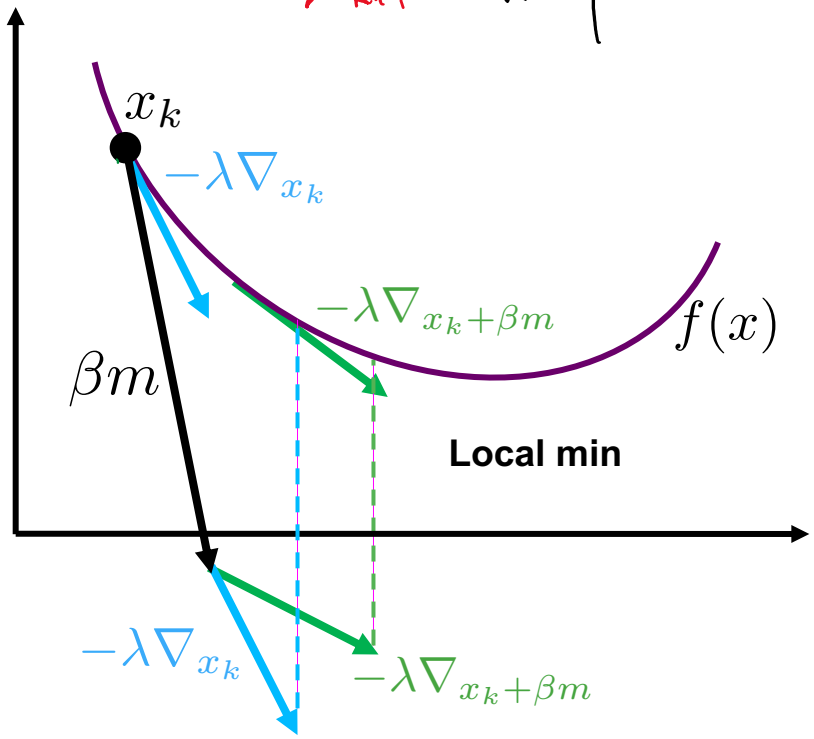
Figure adapted from Charu C. Aggarwal. Neural Networks and Deep Learning. Springer 2018

Nesterov Optimization

- **Input**: $f(x) : \mathbb{R}^n \to \mathbb{R}, x_0 \in \mathbb{R}^n, \lambda \in \mathbb{R}, \beta \in (0, 1)$ (Initial guess; Learning rate; Momentum Parameter)
- **Output**: A local minimum of f

$$x_{k+1} = x_k + \beta m - \lambda \nabla_{x_k}$$

$$x_{k+1} = x_k + \beta m - \lambda \nabla_{x_k + \beta m}$$



- Also called Nesterov Accelerated Gradient (NAG)
- Almost always faster than Momentum
- Computes the gradient at a position slightly ahead of the current point:

$$\nabla_{x_k + \beta m_k}(f)$$

Figure adapted from Aurelien Geron. Hands on machine Learning.

Nesterov Optimization

- **Input**: $f(x) : \mathbb{R}^n \to \mathbb{R}, x_0 \in \mathbb{R}^n, \lambda \in \mathbb{R}, \beta \in (0,1)$ (Initial guess; Learning rate; Momentum Parameter)
- **Output**: A local minimum of f

- For k = 1, 2, …:
  - Compute the gradient $\nabla_{x_k+\beta m_k}(f)$
  - Update the momentum

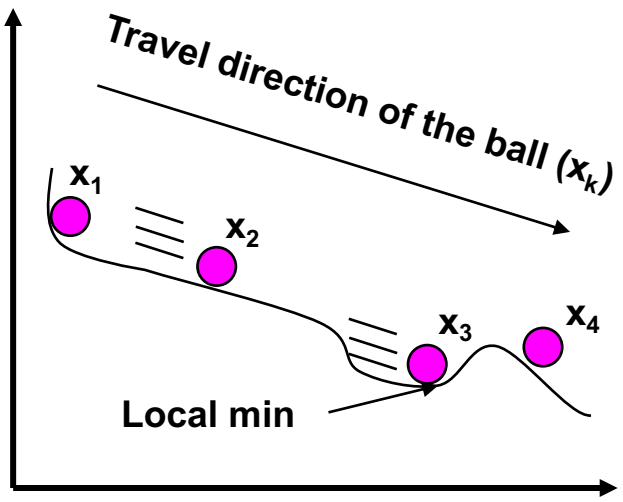
$$m_{k+1} = \beta m_k - \lambda \nabla_{x_k+\beta m_k}(f)$$

Only difference

  - Take the step

$$x_{k+1} = x_k + m_k$$

  - Test for convergence. If yes, then exit.



Momentum gives the "bowling ball" speed so that it does not get stuck in local minima

Figure adapted from Charu C. Aggarwal. Neural Networks and Deep Learning. Springer 2018

Optimizers in R Keras

Stochastic Gradient Descent

```r
model %>% compile(
    loss='binary_crossentropy',
    optimizer = optimizer_sgd(lr = 0.01),
    metrics='accuracy'
)
```

```r
model %>% compile(
    loss='binary_crossentropy',
    optimizer = 'sgd',
    metrics='accuracy'
)
```

Momentum Optimization

```r
model %>% compile(
    loss='binary_crossentropy',
    optimizer = optimizer_sgd(lr = 0.01, momentum = 0.2),
    metrics='accuracy'
)
```

Nesterov Optimization

```r
model %>% compile(
    loss='binary_crossentropy',
    optimizer = optimizer_sgd(lr = 0.01, momentum = 0.2, nesterov = TRUE),
    metrics='accuracy'
)
```

Optimizers in R Keras (Cont'd)

Adam Optimization

```r
model %>% compile(
    loss='binary_crossentropy',
    optimizer = optimizer_adam(lr = 0.001),
    metrics='accuracy'
)
```

```r
model %>% compile(
    loss='binary_crossentropy',
    optimizer = 'adam',
    metrics='accuracy'
)
```

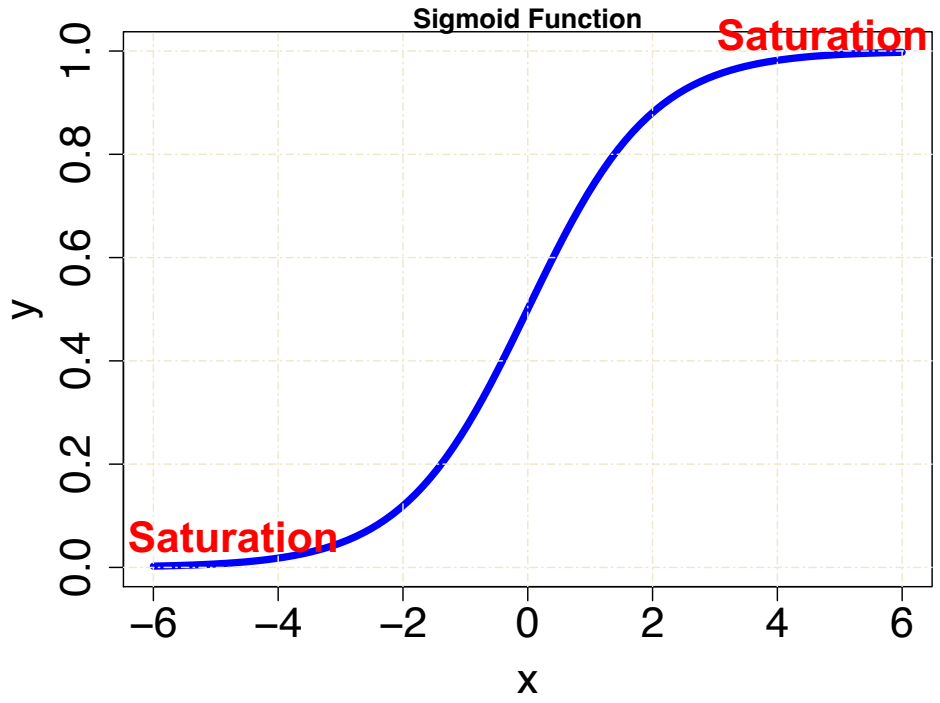
There are other optimizers, and their syntax is similar: rmsprop, nadam, adamax, etc.

Dealing with Slow Training in DNNs

- Training deep neural networks can be very slow
- There are several ways to speed up DNN training:
  - Use faster optimizers
  - Use good activation functions
  - Use good weight initialization strategies
  - Use batch normalization
  - Use pre-trained networks

Other Activation Functions

- Glorot and Bengio noticed in 2010 that part of the problem of the vanishing gradient happened because the sigmoid is a poor activation function for deep neural networks.

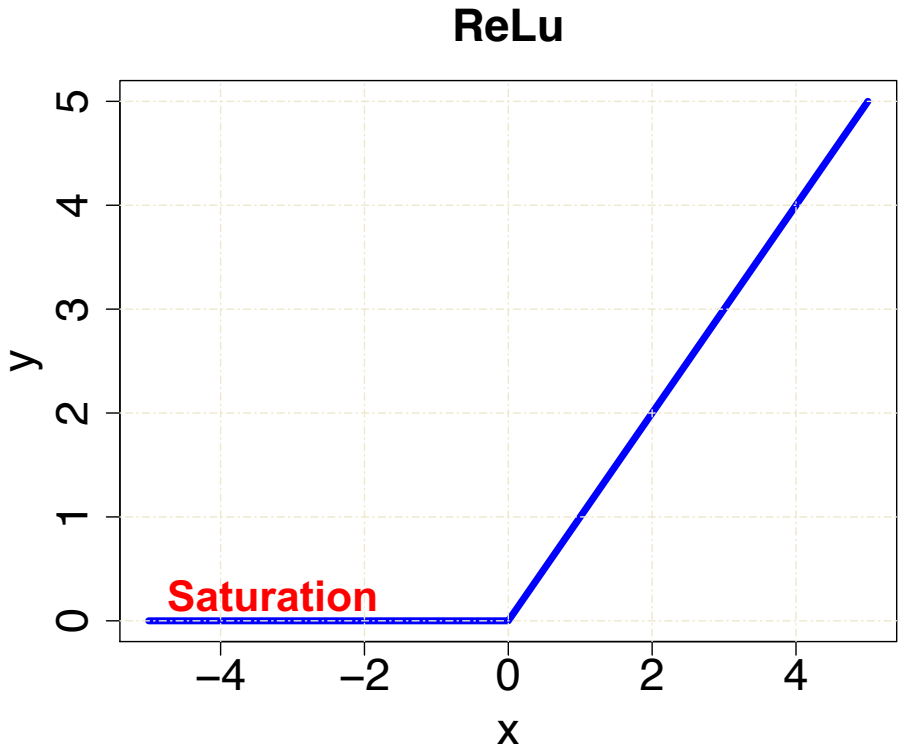


The derivative of the sigmoid becomes 0 for large absolute values of x

Other Activation Functions

- The ReLU function is one such that its derivative is not 0 for large inputs of x.
- However, it does saturate (0 derivative) for x less than or equal to 0.

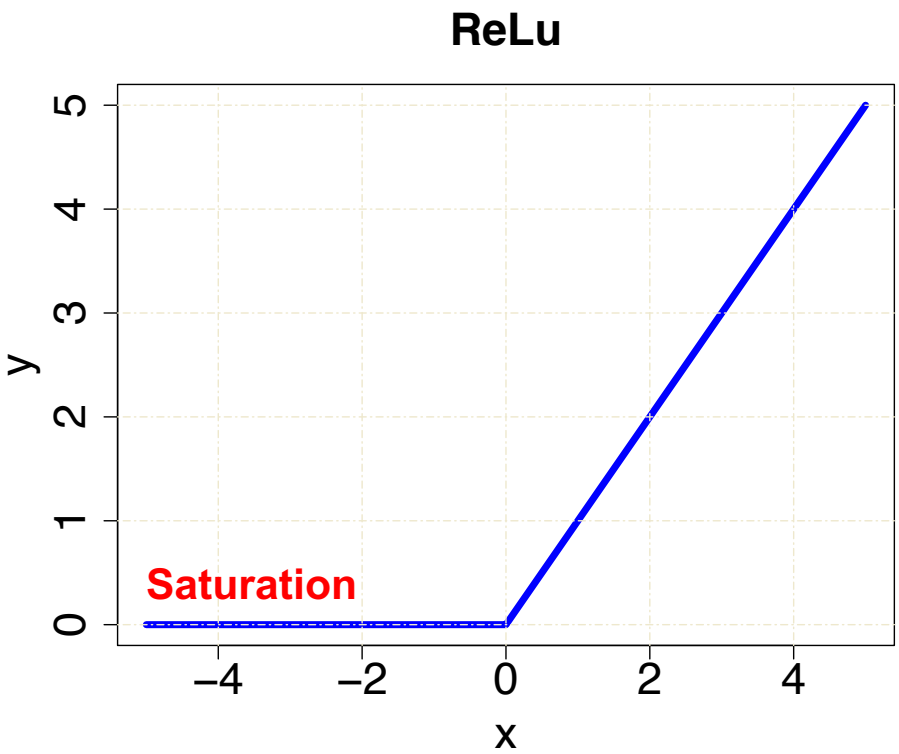
$$ReLU(x) = \begin{cases} x & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

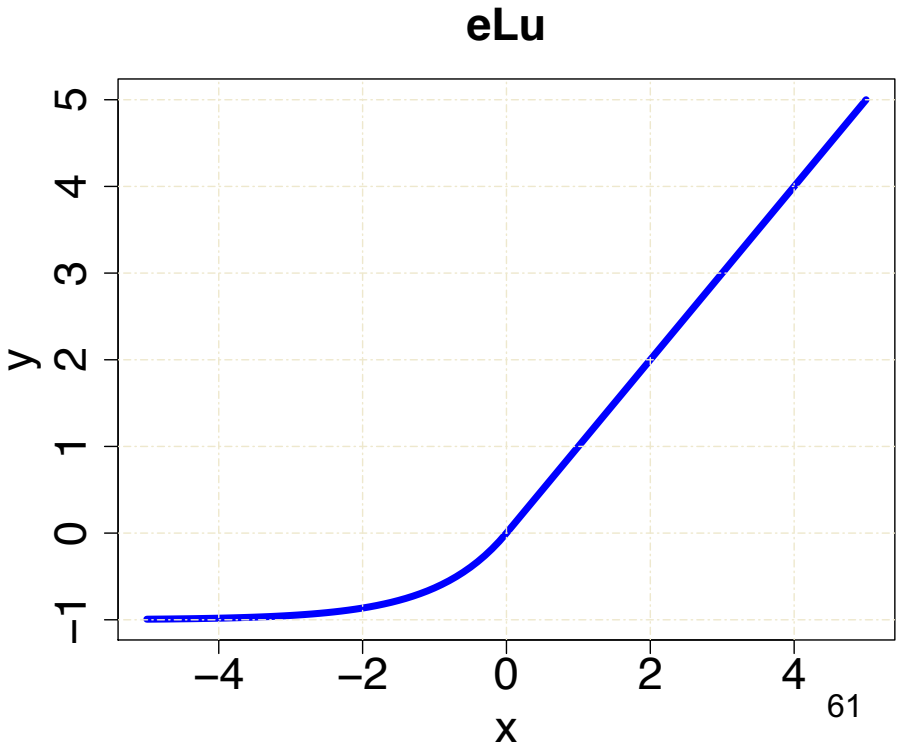


Other Activation Functions

- The Elu function does not saturate, but it is more expensive to compute,

$$ELU_\alpha(x) = \begin{cases} \alpha\left(e^x - 1\right) & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$$





Other Activation Functions

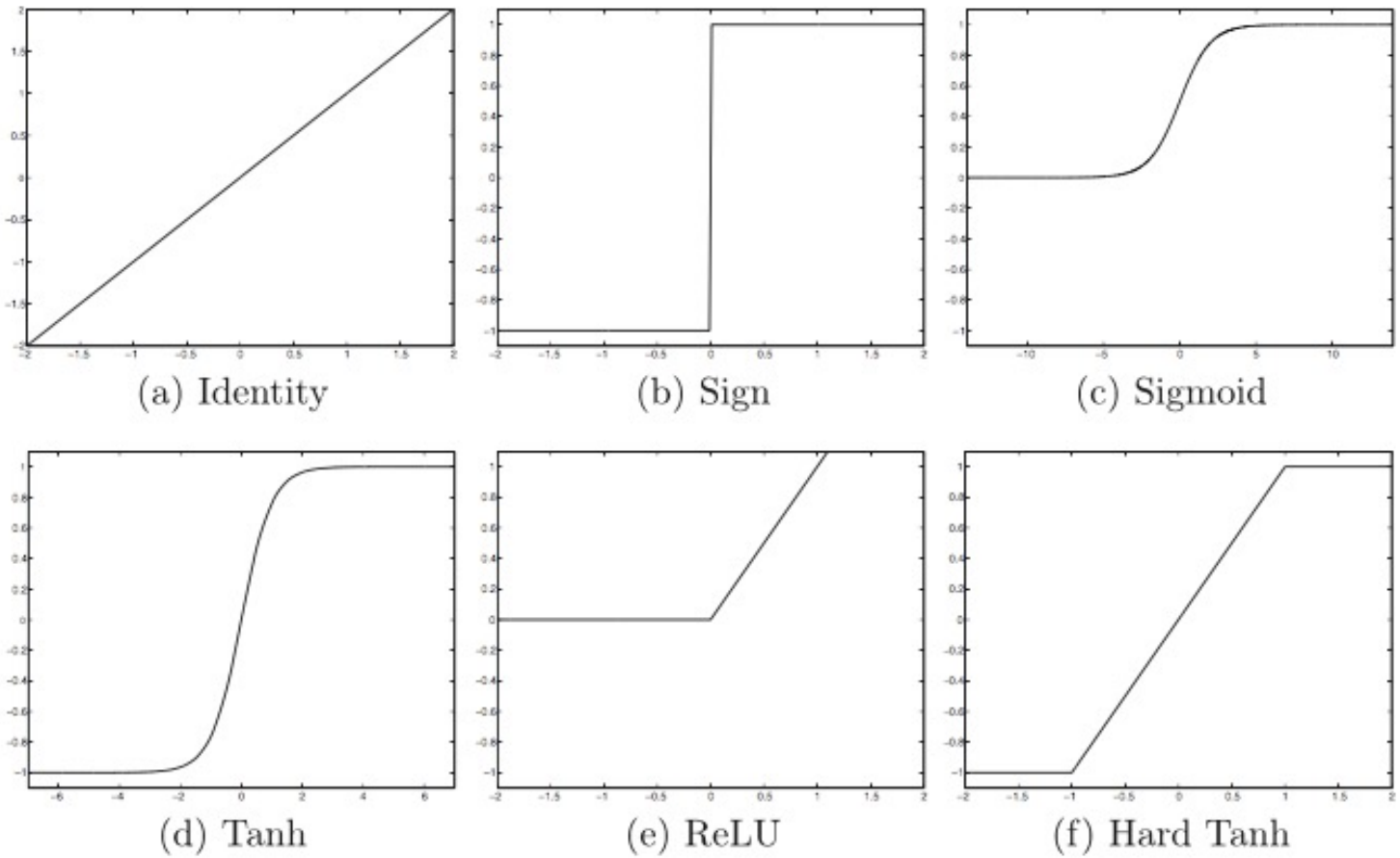
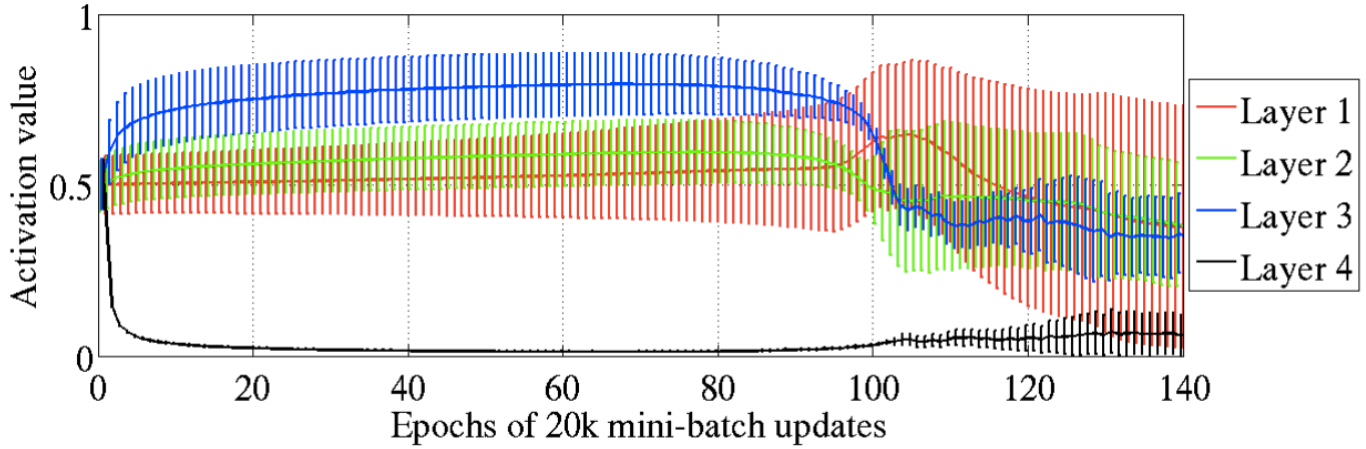


Figure from C. Aggarwal. Neural Networks and Deep Learning, 2018

Dealing with Slow Training in DNNs

- Training deep neural networks can be very slow
- There are several ways to speed up DNN training:
  - Use faster optimizers
  - Use good activation functions
  - Use good weight initialization strategies
  - Use batch normalization
  - Use pre-trained networks

Weight Initialization



- **Layer 1 is the first hidden layer (leftmost if from left to right), and layer 4 is the last hidden layer**
- Using a DNN with the sigmoid function and random initialization U(0,1) on the Shapeset-3 x 2 dataset training with stochastic backpropagation on mini-batches of size 10
- The figures show the the mean and standard deviations of the outputs of the neurons at different hidden layers
- The variance of the output is in general much bigger than the variance of the inputs

Figure from Glorot and Bengio, “Understanding the Difficulty of Training Deep Feedforward Neural Networks”, 2010

Weight Initialization

Initialization	Best for these Activation Functions	Sample from the following distribution
Glorot and Bengio (2010)	Linear, tanh, logistic, softmax	$N\left(\mu = 0, \sigma^2 = \frac{2}{n_{\text{inputs}} + n_{\text{outputs}}}\right)$
He (2015)	ReLu, eLu	$N\left(\mu = 0, \sigma^2 = \frac{4}{n_{\text{inputs}}}\right)$
LeCun (1990s)	SELU	$N\left(\mu = 0, \sigma^2 = \frac{1}{n_{\text{inputs}}}\right)$

Here n_{inputs} is the number of inputs of each layer, and $n_{outputs}$ is the number of outputs, so the standard deviation is layer dependent

Glorot and Bengio, “Understanding the Difficulty of Training Deep Feedforward Neural Networks”, 2010
He, “Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification”, 2015

Weight Initialization in Keras

```
# Defines the structure of the NN
model <-
  keras_model_sequential() %>%
  layer_flatten(input_shape = c(28,28)) %>%
  layer_dense(units = 200,
              activation = 'relu',
              kernel_initializer = initializer_he_normal()) %>%
  layer_dense(units = 100,
              activation = 'relu',
              kernel_initializer = initializer_he_normal()) %>%
  layer_dense(units = 10, activation = 'softmax')

# Specifies how training will be done
model %>% compile (
  loss = 'binary_crossentropy',
  metrics = c('accuracy'),
  optimizer = 'adam'
)
# Begins training
model %>% fit(
  train_x, train_y,
  epochs = 10,
  batch_size = 32,
  validation_split = 0.2
)
```

Dealing with Slow Training in DNNs

- Training deep neural networks can be very slow
- There are several ways to speed up DNN training:
  - Use faster optimizers
  - Use good activation functions
  - Use good weight initialization strategies
  - Use batch normalization
  - Use pre-trained networks

Batch Normalization

- Good activation functions combined with good initialization strategies help deal with the vanishing gradient only at the beginning of training.
- To deal with this problem during training, we can use **batch normalization**.
- During training, when a layer receives a mini-batch as input, BN normalizes this mini-batch by subtracting its mean and dividing by its standard deviation. Then it scales and shifts this mean.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Epsilon is a small number (e.g. 10^{-6}) to avoid dividing by 0

Figure from Ioffe and Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariance Shift", ICML 2015

Batch Normalization in Keras

```r
model <-
  keras_model_sequential() %>%
  layer_flatten(input_shape = 64) %>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_batch_normalization() %>%
  layer_dense(units = 4, activation = 'softmax')
```

```r
summary(model)
```

```
Model: "sequential_13"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_13 (Flatten)         (None, 64)                0
_________________________________________________________________
dense_27 (Dense)             (None, 32)                2080
_________________________________________________________________
batch_normalization_10 (BatchNormal (None, 32)         128
_________________________________________________________________
dense_28 (Dense)             (None, 4)                 132
=================================================================
Total params: 2,340
Trainable params: 2,276
Non-trainable params: 64
_________________________________________________________________
```

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β

Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

- Notice that batch normalization adds more parameters to the model.
- It adds 4 parameters (mu, sigma, gamma and beta) per unit in a layer (32 * 4 = 128).
- Out of these 4 parameters per unit, only 2 are trainable (gamma and beta). That is the reason for the 64 non-trainable parameters.

Dealing with Slow Training in DNNs

- Training deep neural networks can be very slow
- There are several ways to speed up DNN training:
  - Use faster optimizers
  - Use good activation functions
  - Use good weight initialization strategies
  - Use batch normalization
  - Use pre-trained networks

Pre-trained Networks

- There are many networks available for download that have been already pre-trained for certain tasks and that you can re-use for other similar purposes:
  - LeNet
  - AlexNet
  - GoogLeNet
  - Inception

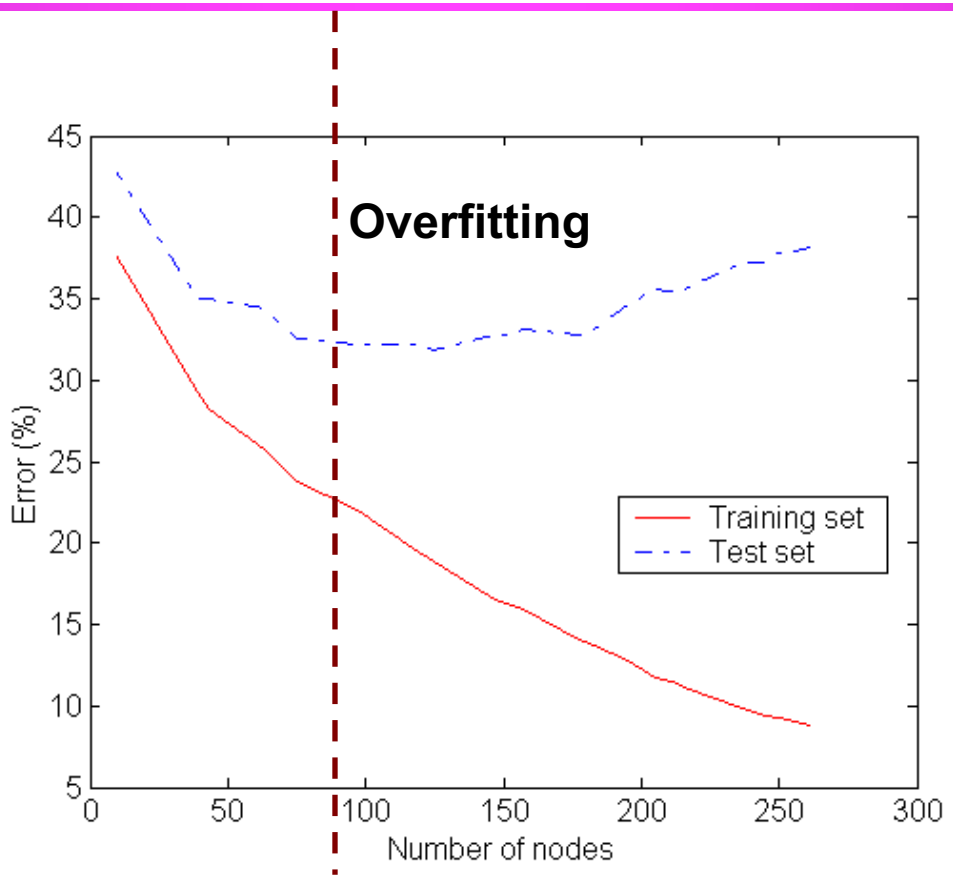
https://www.mathworks.com/videos/deep-learning-in-11-lines-of-matlab-code-1481229977318.html

Caffe's model zoo: https://github.com/BVLC/caffe/wiki/Model-Zoo
Tensorflow's model Garden: https://github.com/tensorflow/models

DNNs and Regularization

- Regularization is one of the mechanisms to deal with overfitting in any predictive model.
- Some regularization mechanisms for DNNs are:
  - Early stopping
  - L_1 or L_2 regularization
  - Dropout regularization

Early Stopping



Stop training when you see no progress on the validation set for a number of epochs (patience)

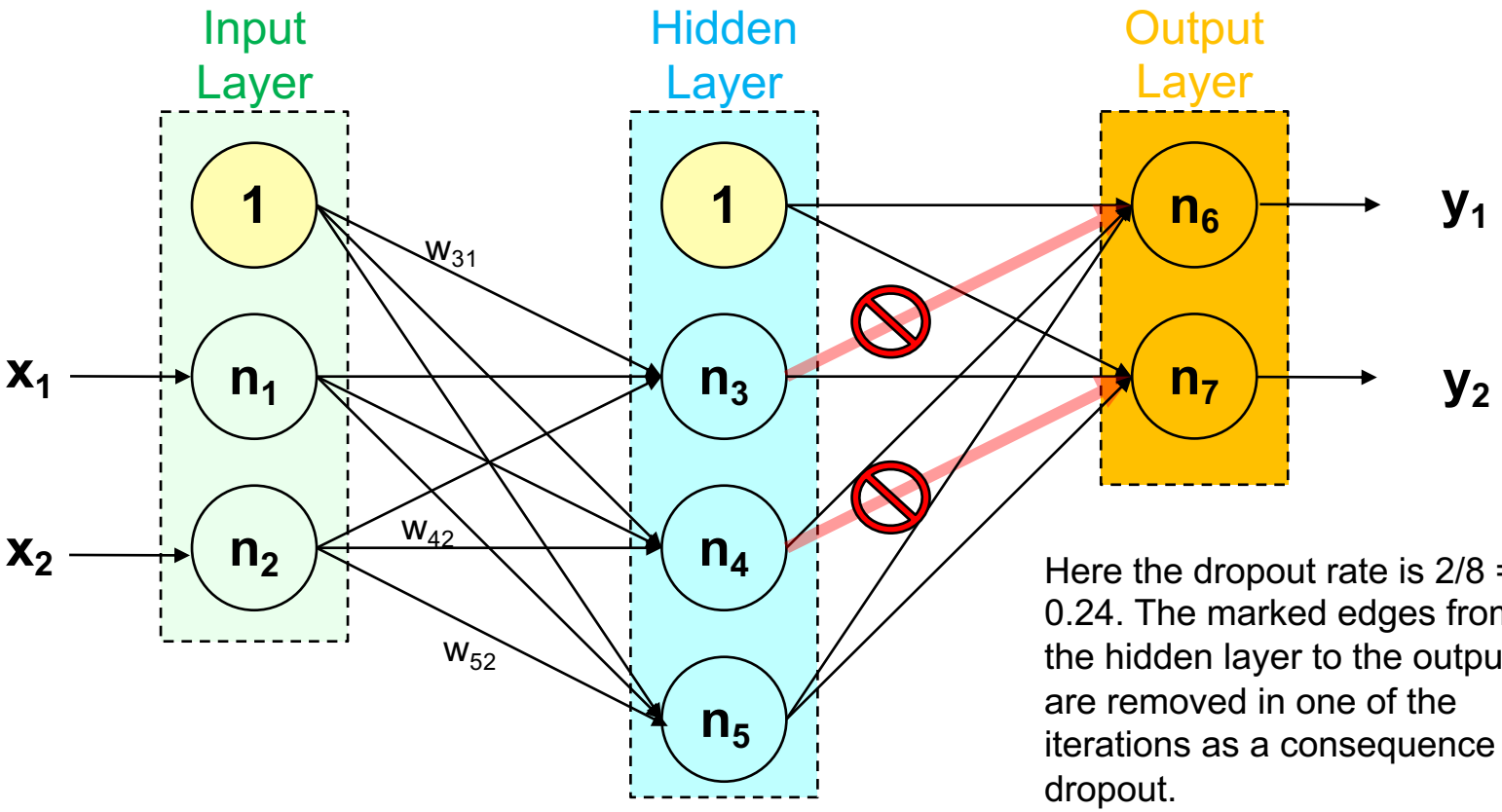
Early Stopping in R Keras

```
# Defines the structure of the NN
model <-
  keras_model_sequential() %>%
  layer_flatten(input_shape = 64) %>%
  layer_dense(units = 32, activation = 'relu') %>%
  layer_dense(units = 1, activation = 'softmax')

# Specifies how training will be done
model %>% compile(
    loss='binary_crossentropy',
    optimizer = 'sgd',
    metrics='accuracy')

# Begins training
model %>% fit(data, labels, callbacks = list(
  callback_early_stopping(monitor = "val_loss", patience = 1)
))
```

In this example, if there is no progress in the validation loss for patience = 1 epoch, it stops training.

Dropout Regularization (Hinton, 2012)

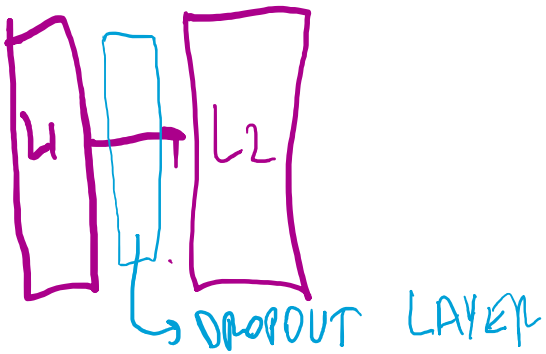
- At every **training step (not testing)**, each neuron has a probability *p* of being dropped out temporarily, but may come back in the next step



Here the dropout rate is 2/8 = 0.24. The marked edges from the hidden layer to the output are removed in one of the iterations as a consequence of dropout.

Dropout Regularization (Cont'd)

- Just for visualizing the effects of dropout, we can simulate that we have a layer output that we dropout.
- Keras normalizes the output by dividing by 1/(1-rate)




```{r}
library(reticulate)
use_python("~/venv")
```

```{r}
library(keras)
library(tensorflow)
```

Assume that the output of a layer is:

```{r}
layer1.output <- c(3, 6, 2, -1.6, 10, 11.0, 12, 4)
```

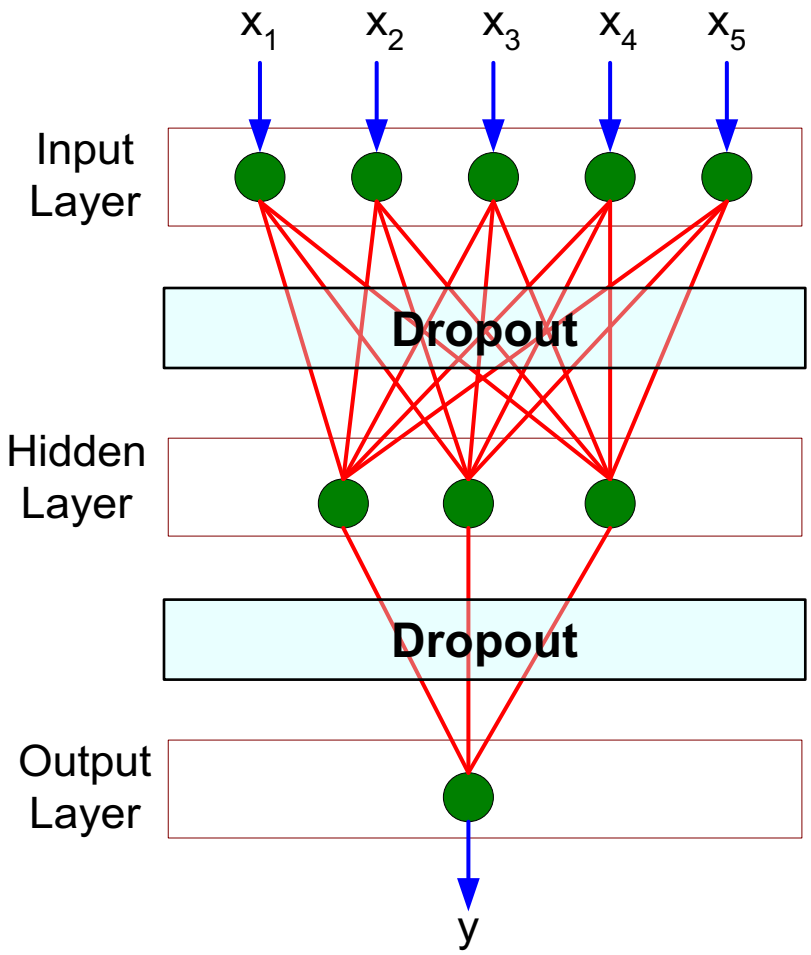
then, after doing dropout:

```{r}
after.dropout <- tf$nn$dropout(layer1.output, rate = 0.5)
after.dropout$numpy()
```

```
[1]  6 12  0  0 20 22 24  8
```

Note: If you use Keras, you do not call the dropout function like in this slide. This example is only to demonstrate behavior of this function. To add a dropout layer in Keras, you must use the layer_dropout function instead.

Dropout Regularization in R Keras

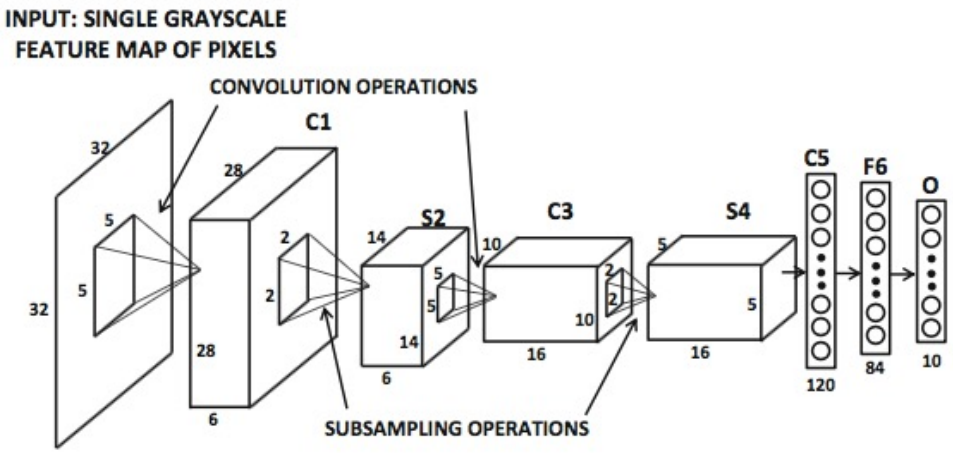


- One can add a dropout layer in between any two other layers.

```
# Defines the structure of the NN
model <-
  keras_model_sequential() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 3,
              input_shape = 5,
              activation = 'sigmoid')
%>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1,
              activation = 'sigmoid')
```

Other Deep NN Architectures

- There are several other Deep Neural Network Architectures:
  - Convolutional Neural Networks (CNN)
  - Autoencoders
  - Recurrent Neural Networks (RNN)



LeNet-5: A Convolutional Neural Network

Figure adapted from Charu C. Aggarwal. Neural Networks and Deep Learning. Springer 2018

Convolutions

- Receives as input an n x n matrix I (representing a grayscale image, e.g.) and a filter F of size m x m:

$$(F * I)(x, y) = \sum_{i=-m}^{m} \sum_{j=-m}^{m} F(i, j) I(x - i, y - j)$$

Matrix

-3	1	7	5	-5
2	-1	8	4	1
8	1	-4	7	3
3	1	-2	1	4

*

Convolution kernel

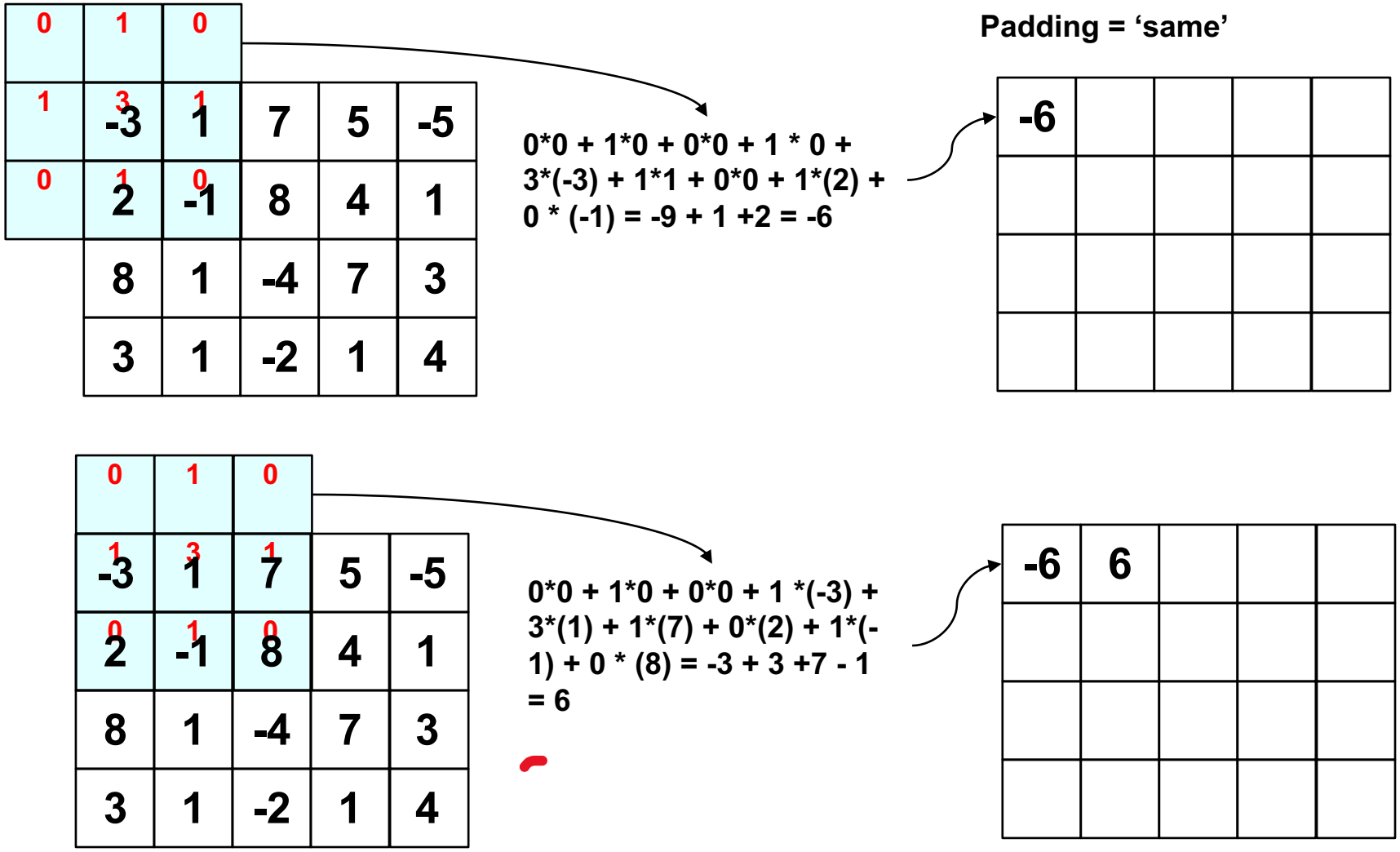
0	1	0
1	3	1
0	1	0

=

-6	6	35	21	-9
10	9	30	33	5
30	7	2	25	21
18	5	-8	12	16

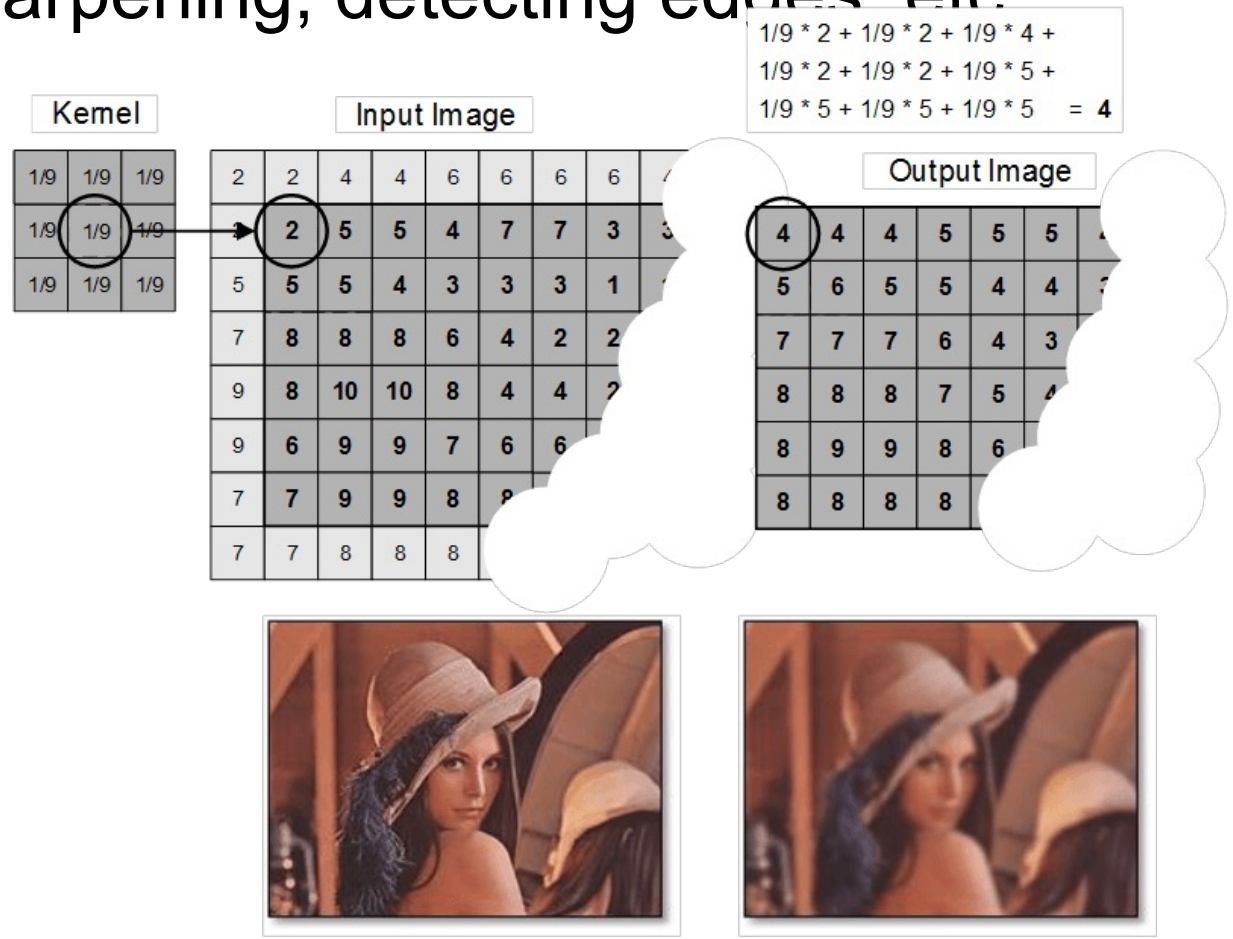
Convolution Example

- Here we assume that we padded the matrix with 0s



Convolutions Applications

- Useful for all kinds of image processing tasks like blurring, sharpening, detecting edges, etc.



The example on the right shows the application of a "simple blur.

Figure from https://www.imgtec.com/blog/heterogeneous-compute-case-study-image-convolution-filtering/

Convolutions Applications (Cont'd)

- Useful for all kinds of image processing tasks like blurring, sharpening, detecting edges, etc.

-1	-1	-1
2	2	2
-1	-1	-1

-1	2	-1
-1	2	-1
-1	2	-1

Kernels for detecting horizontal and vertical lines

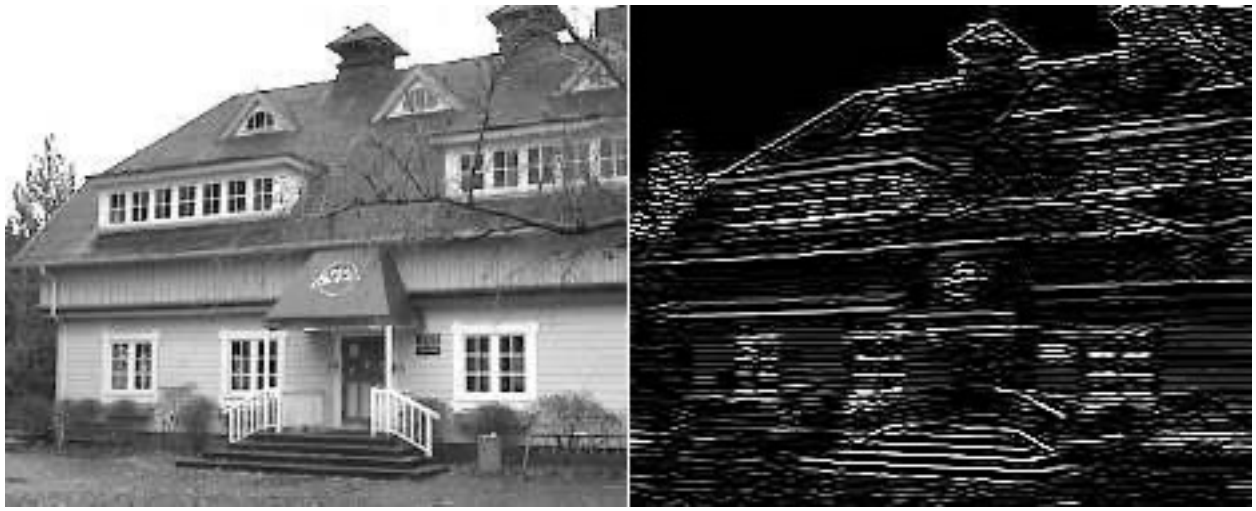


Figure adapted from https://aishack.in/tutorials/image-convolution-examples//

Pooling Example

- A pooling layer is used to reduce the size of another layer by “sampling”

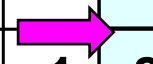
-3	1	7	5
2	-1	8	4
8	1	-4	7
3	1	-2	1

MAX Pool

3	8
8	7

MAX Pool

-3	1	7	5
2	-1	8	4
8	1	-4	7
3	1	-2	1

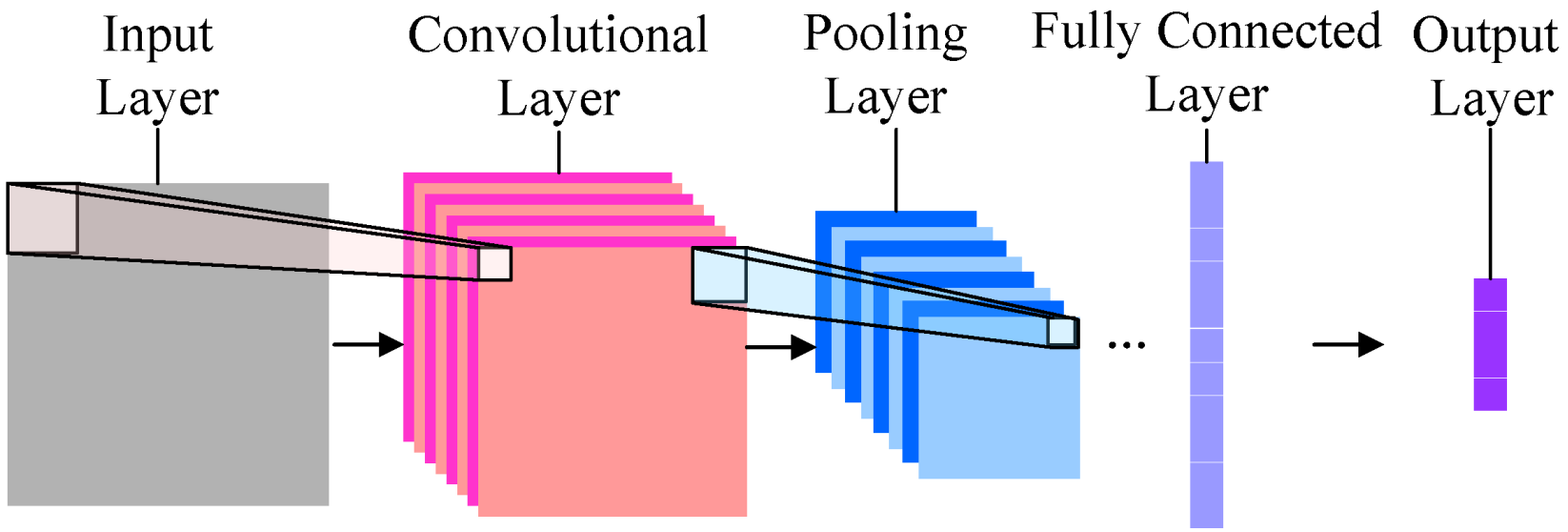


MAX Pool

3	8

MAX Pool

Convolutional Neural Networks (CNNs)



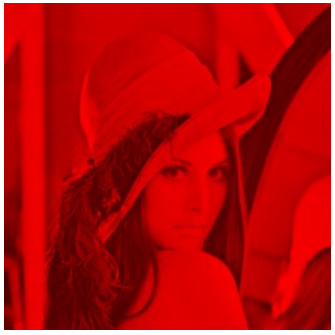
Parenthesis about Image Channels

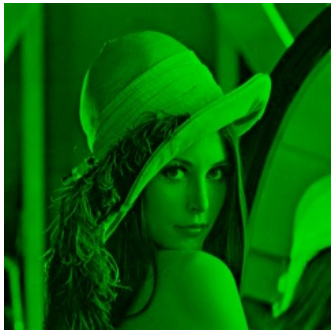
- Most images that we deal with everyday have 3 channels: Red, Green, and Blue (RGB)

Red

Green

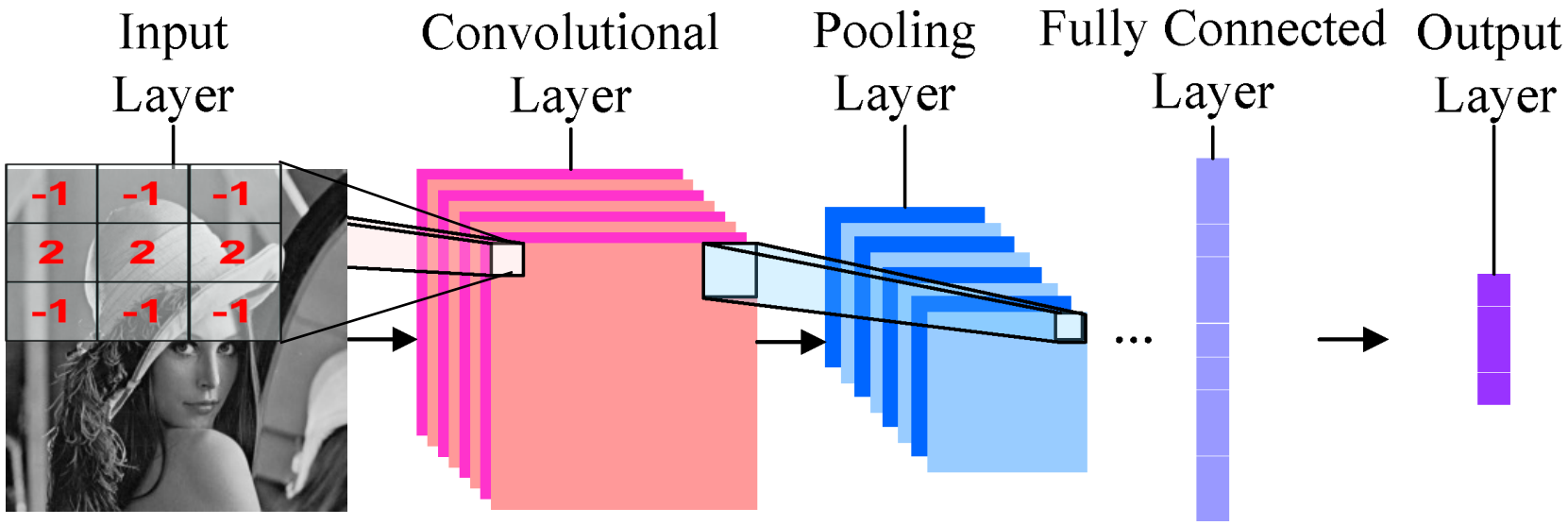
Blue



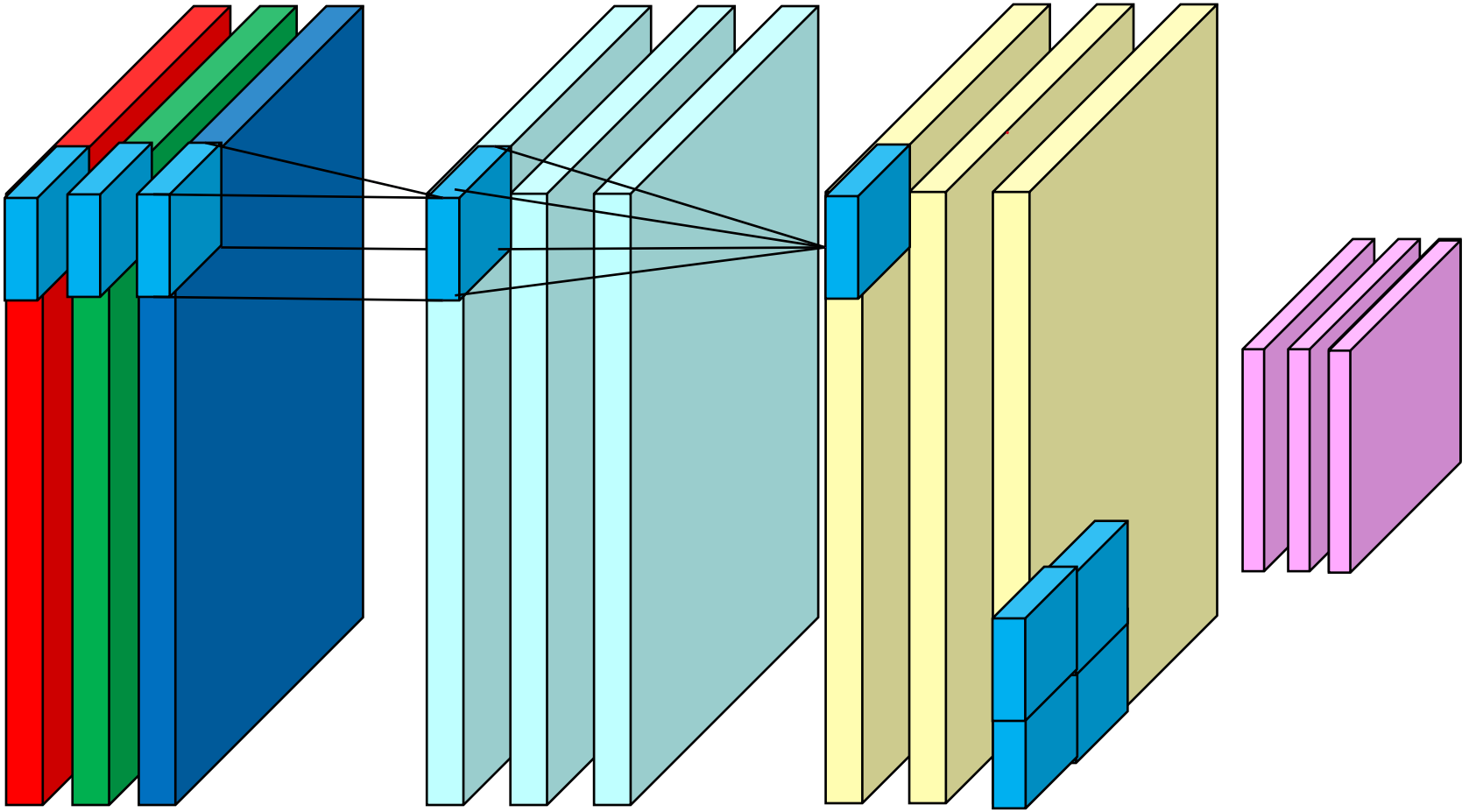


Convolutional Neural Networks (CNNs)

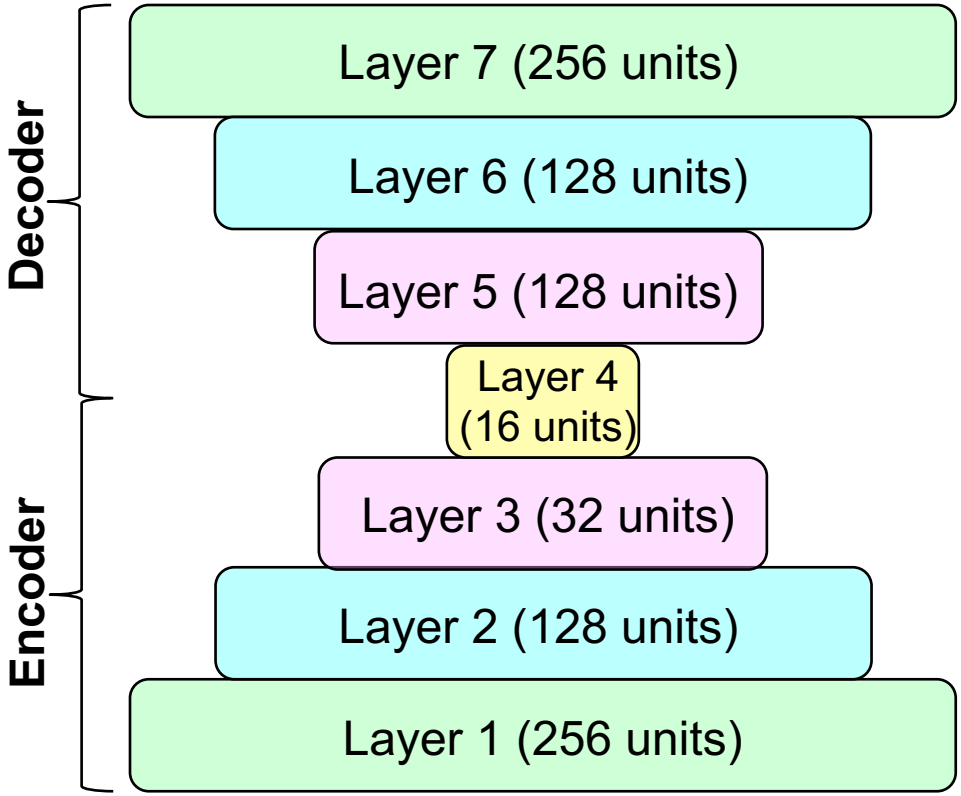
- A convolutional layer is made up of multiple feature maps.
- Each feature map in level l learns a filter corresponding to each feature map in level l-1



Convolutional Neural Networks (CNNs)



Autoencoders



Example of an autoencoder architecture. Autoencoders need not have 7 layers, their layers need not have the same units as in this example.

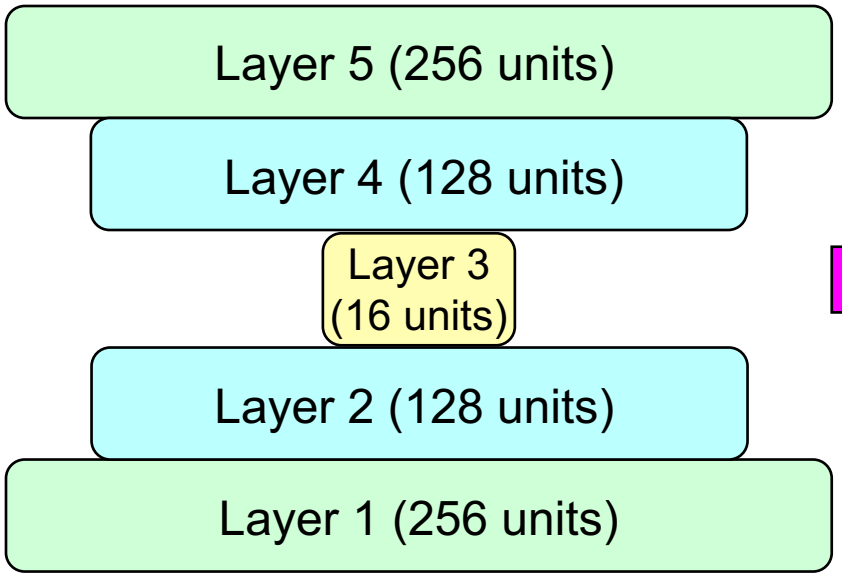
- **Autoencoders** are a neural network architecture composed of two groups of layers:
  - Encoder
  - Decoder
- The layers have a decreasing number of units as we advance from the input layer towards the middle.
- The layers have an increasing number of units from the middle as we advance towards the output.

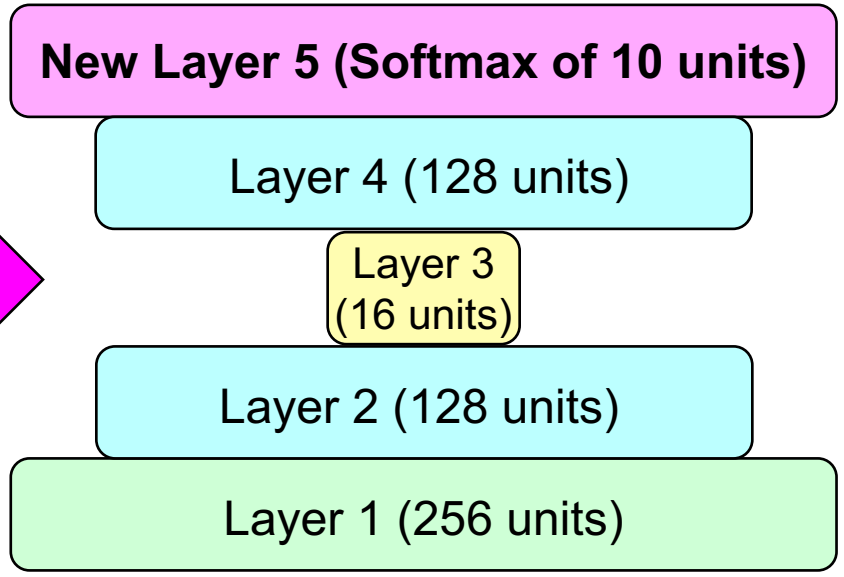
Unsupervised Pre-training

- Applied when there are few labeled data for training and many unlabeled data for training.
- Unsupervised pre-training works in two stages:
  1. Fit an autoencoder to the unlabeled data to predict the unlabeled data itself
  2. Then replace only the last layer with something appropriate for the real task at hand
  3. Fit the resulting neural network to the labeled data only.

Unsupervised Pre-training

Pr(0), Pr(1), …, Pr(9)





2 8

Deep Learning: Number of Parameters

- Deep neural networks have an enourmous number of parameters.
- You need large datasets to train these without overfitting

Neural Network	Number of Parameters (approx.)
LeNet-5	61,000
GoogleNet (2014)	7,000,000
ResNet-50 (2016)	23,000,000
GPT-3 (2020)	175,000,000,000 = $1{,}75 \times 10^{11}$
GPT-4 (?)	100,000,000,000,000 = 1×10^{14}