ML: Lecture 4

Deep Learning I: Training Neural Networks

Panagiotis Papapetrou, PhD
Professor, Stockholm University

Syllabus

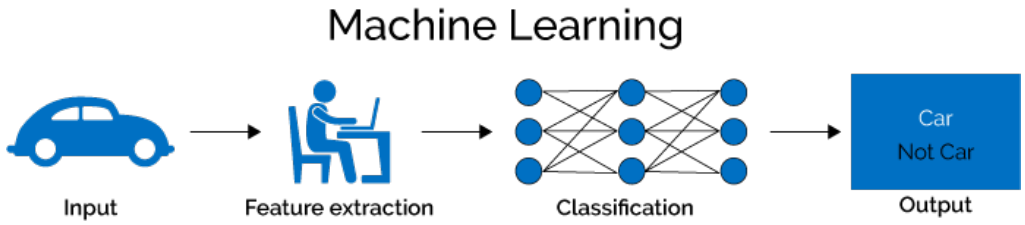
Jan 16	Introduction to machine learning
Jan 18	Regression analysis
Jan 19	Laboratory session 1: numpy and linear regression
Jan 23	Ensemble learning
Jan 25	**Deep learning I: Training neural networks**
Jan 26	Laboratory session 2: ML pipelines, ensemble learning
Jan 30	Deep learning II: Convolutional neural networks
Feb 1	Laboratory session 3: training NNs and tensorflow
Feb 6	Deep learning III: Recurrent neural networks
Feb 8	Laboratory session 4: CNNs and RNs
Feb 13	Deep learning IV: Autoencoders, transformers, and attention
Feb 20	Time series classification

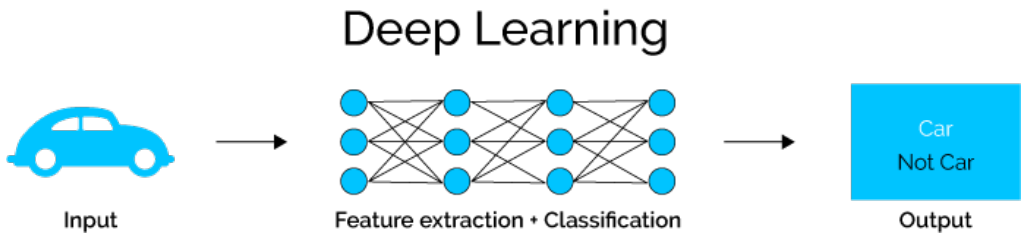
Today

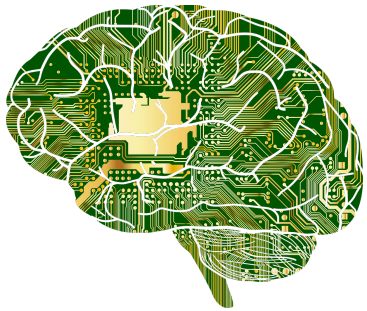
- What is deep learning?
- Commonly used activation functions
- Basic training procedures for deep learning
- Regularization techniques for NNs
- Gradient descent

Deep Learning

- A subfield of machine learning using Artificial Neural Networks for learning representations of data
- Exceptionally effective at learning hidden patterns
- Deep learning algorithms attempt to learn (multiple levels of) feature representations by using a hierarchy of multiple layers
- If you provide "tons" of information as input, it begins to understand it and respond in useful ways





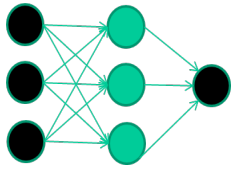


The series of layers between input & output do **feature identification and processing in a series of stages**, just as our brains do!

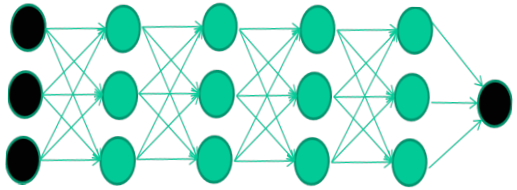
Why now?

- Multilayer neural networks have been around for 70 years
- What is actually new?

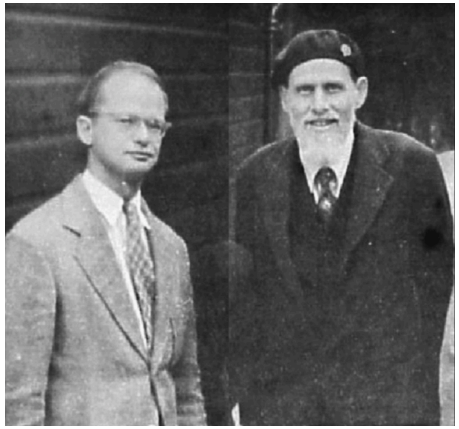
we have always had good algorithms for learning the **weights** in networks with **1 hidden layer**



but these algorithms are **not good at learning the weights** for networks with **more hidden layers**



what is new: algorithms for training multi-layer networks, the availability of massive amounts of data, computing power



McCulloch & Pitts, 1943

"A logical calculus of the ideas immanent in nervous activity"

tried to understand how the brain could produce highly complex patterns by using many basic cells that are connected

The simplest ANN (recall from DAMI)

The Perceptron

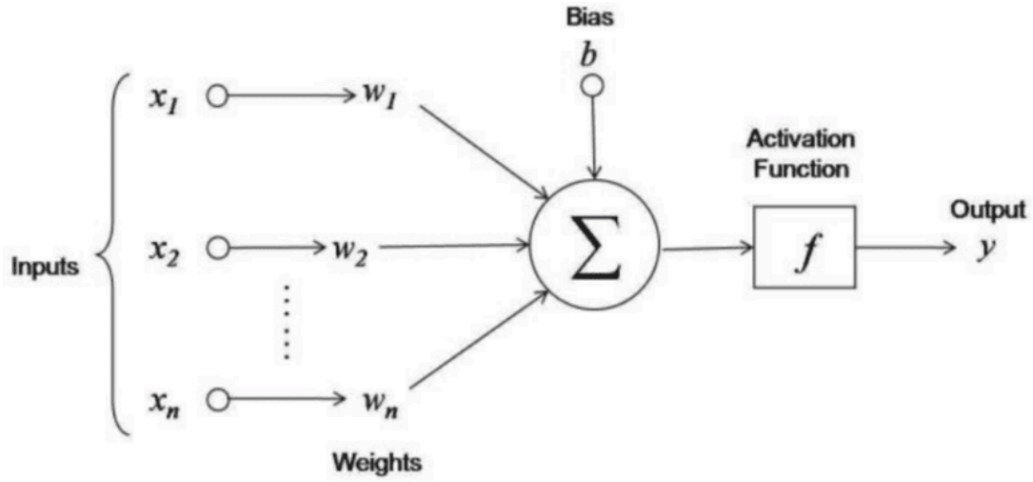
- an artificial neuron
- **input:** $X = \langle x_1, x_2, \ldots, x_n \rangle$
- **n** weights: $W = \langle w_1, w_2, w_3, \ldots, w_n \rangle$
- **summation function** (**dot product**):

  $x_1w_1 + x_2w_2 + x_3w_3 + \ldots$

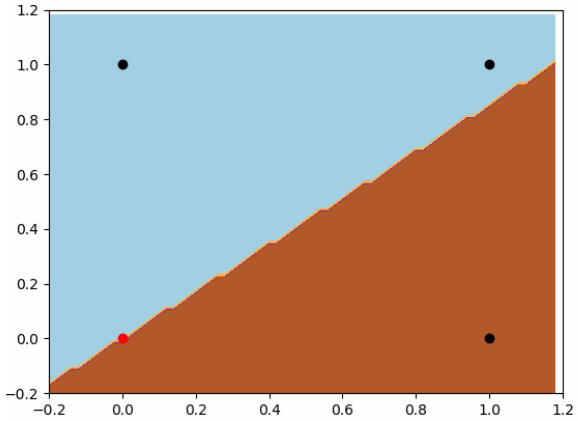
- **bias:** allows you to shift the activation function to either right or left
- activation function:

  $f(X*W) > 0$

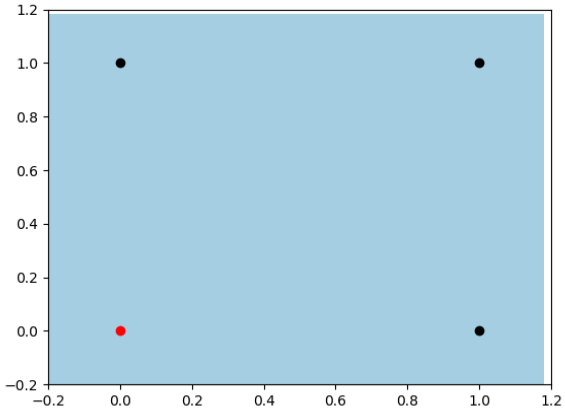
- **output:** a class label



OR problem - no bias

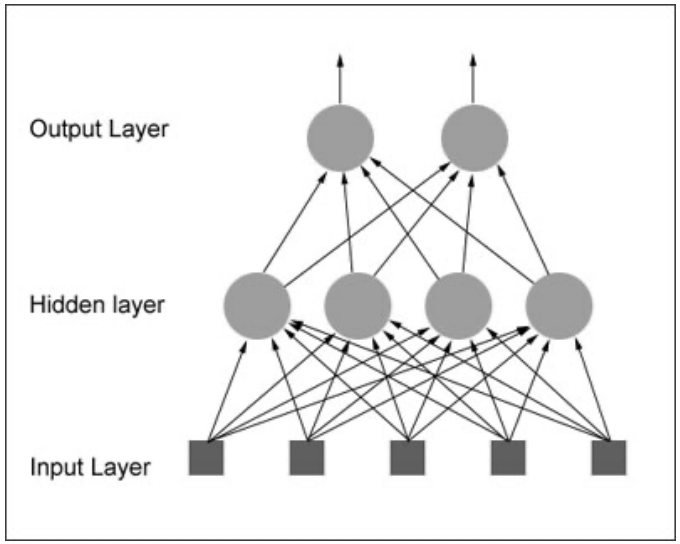


OR problem - with bias



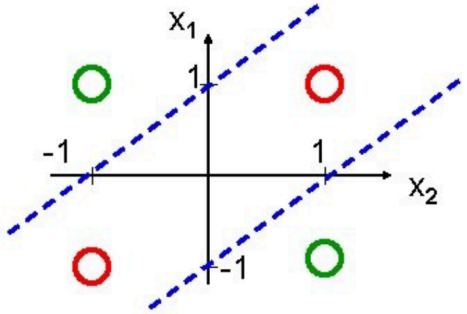
The multi-layer perceptron (MLP)

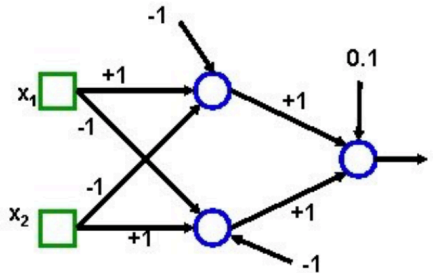
- More complex problems may require intermediate layers



The XOR problem

x_1	x_2	x_1 xor x_2
-1	-1	**-1**
-1	1	**1**
1	-1	**1**
1	1	**-1**



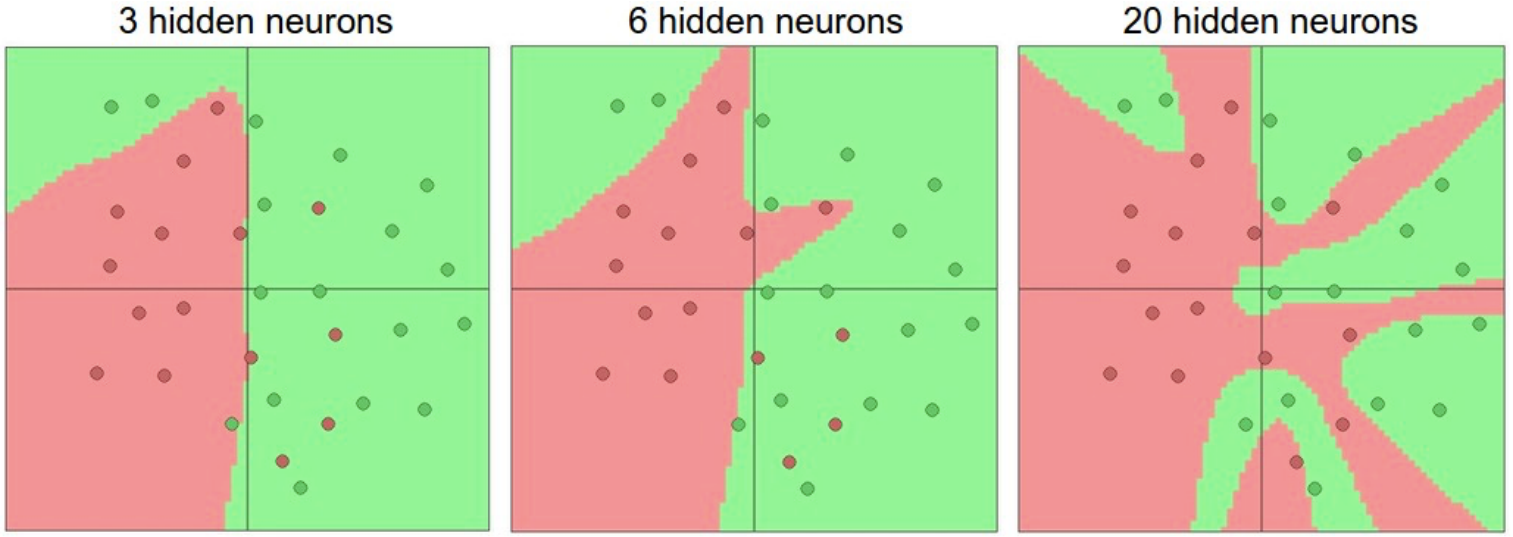


$$\varphi(v) = \begin{cases} 1 & \text{if } v > 0 \\ -1 & \text{if } v \le 0 \end{cases}$$

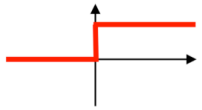
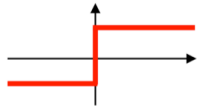
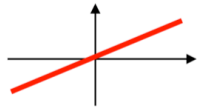
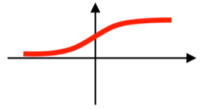
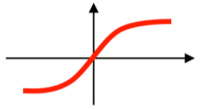
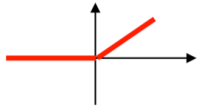
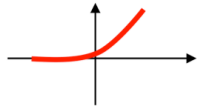
φ is the sign function

Activation function *f*

- If *f*(*x*) is non-linear, a network with 1 hidden layer can, in theory, learn *perfectly* any classification problem
- The set of weights that can produce the targets from the inputs exists
- The problem is finding the proper number of neurons and learn their weights

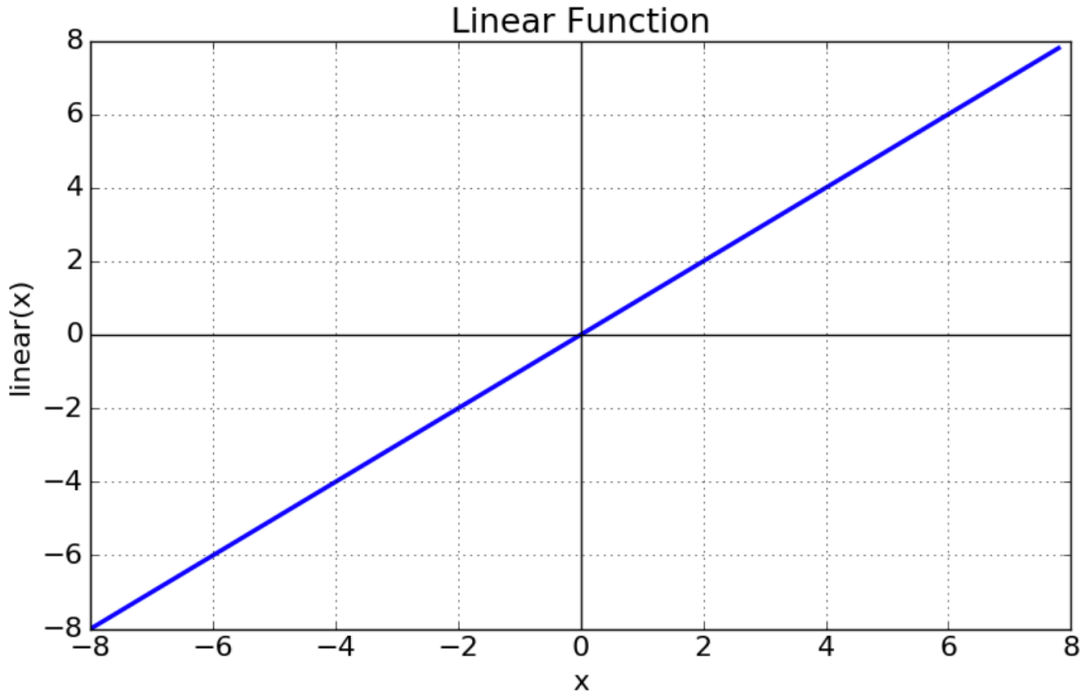


Many possible activation functions

Activation function	Equation	Example	1D Graph
Unit step (Heaviside)	$\phi(z) = \begin{cases} 0, & z < 0, \\ 0.5, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Sign (Signum)	$\phi(z) = \begin{cases} -1, & z < 0, \\ 0, & z = 0, \\ 1, & z > 0, \end{cases}$	Perceptron variant	
Linear	$\phi(z) = z$	Adaline, linear regression	
Piece-wise linear	$\phi(z) = \begin{cases} 1, & z \geq \frac{1}{2}, \\ z + \frac{1}{2}, & -\frac{1}{2} < z < \frac{1}{2}, \\ 0, & z \leq -\frac{1}{2}, \end{cases}$	Support vector machine	
Logistic (sigmoid)	$\phi(z) = \frac{1}{1 + e^{-z}}$	Logistic regression, Multi-layer NN	
Hyperbolic tangent	$\phi(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$	Multi-layer Neural Networks	
Rectifier, ReLU (Rectified Linear Unit)	$\phi(z) = max(0, z)$	Multi-layer Neural Networks	
Rectifier, softplus	$\phi(z) = \ln(1 + e^{z})$	Multi-layer Neural Networks	

Linear activation function

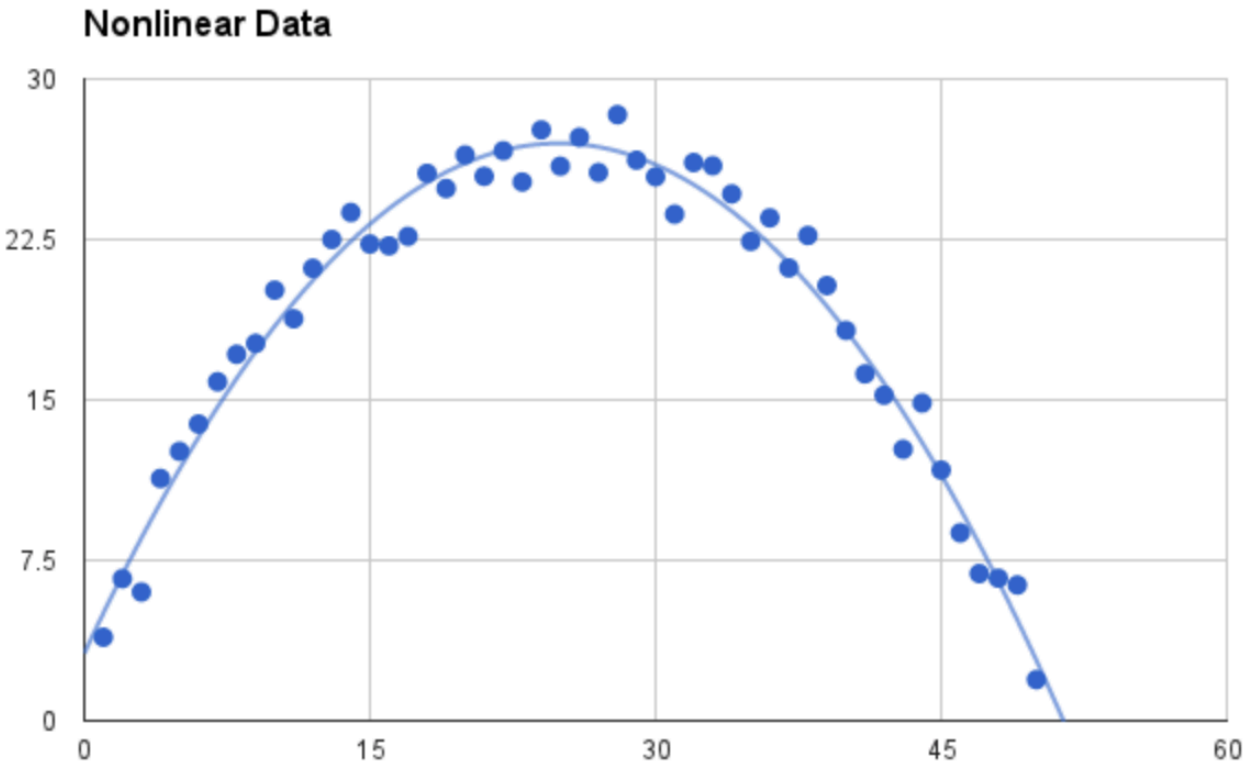
- **Equation:** f(x) = x
- **Range:** (-∞, ∞)



Not flexible enough to capture complex relations between data variables

Non-linear activation functions

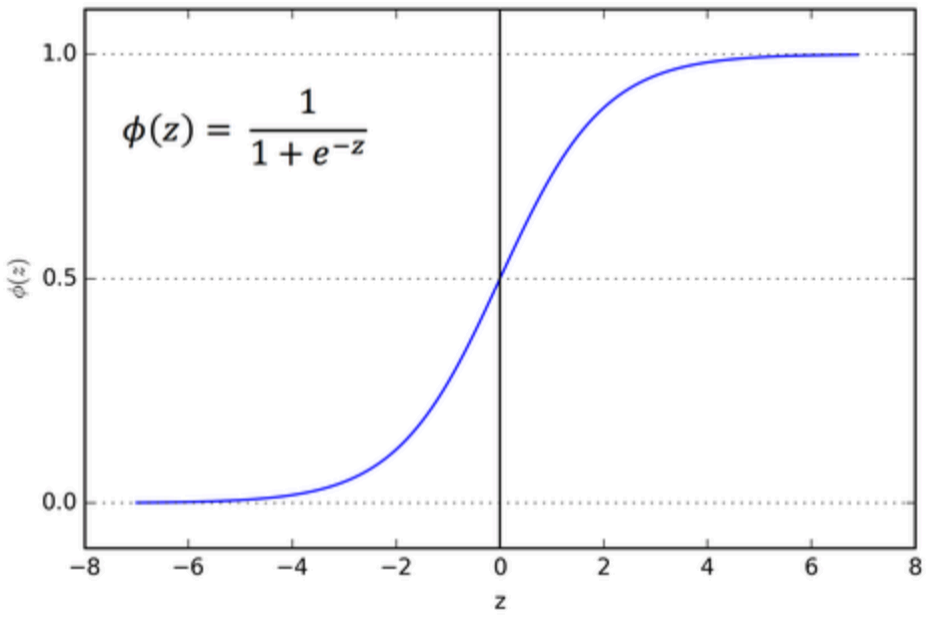
- It makes it easier for the model to generalize or adapt as data varies



Sigmoid activation function

- **Range:** (0, 1)

Convenient for models that predict probabilities



The function is:

- **differentiable:** we can find the slope of the sigmoid curve at any two points
- **monotonic**, but its derivative is not

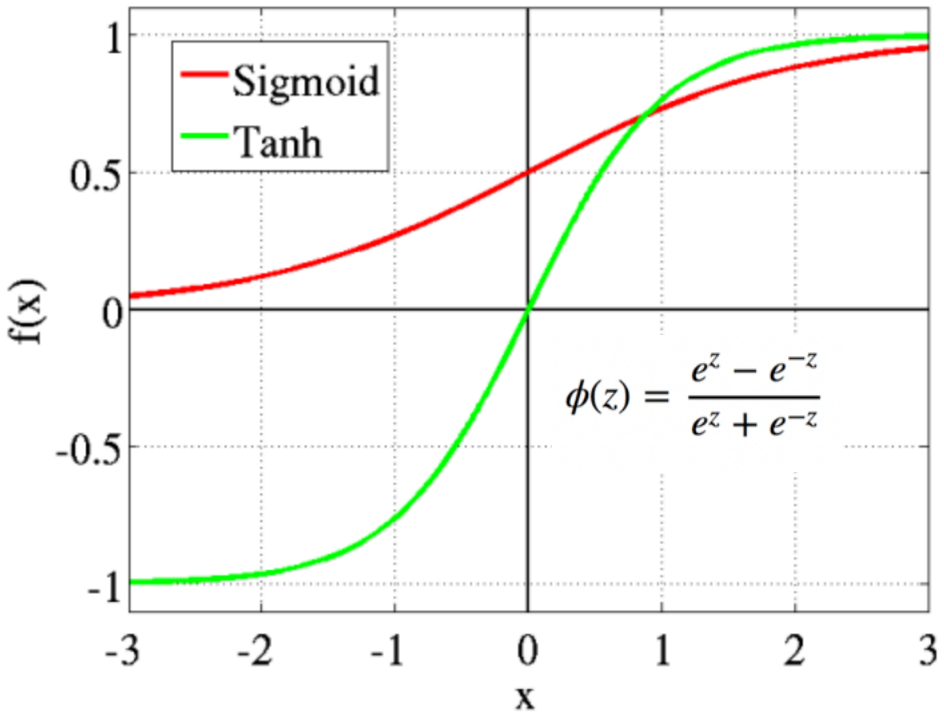
It can cause a neural network to get stuck at training time

The **softmax function** is a more generalized logistic activation function which is used for **multiclass classification**

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

The Tanh activation function

- **Range:** (-1, 1)
- Takes negative values



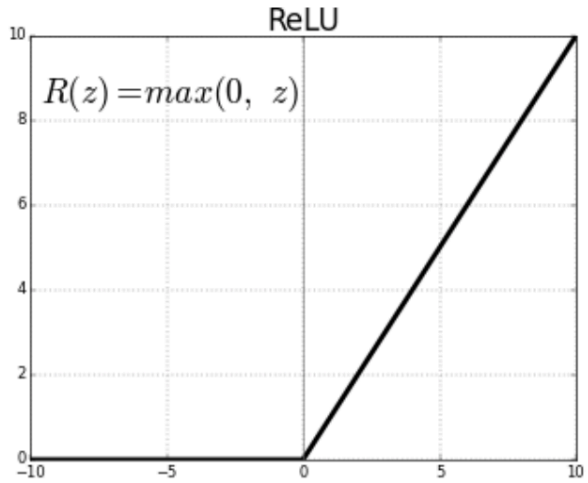
The function is:

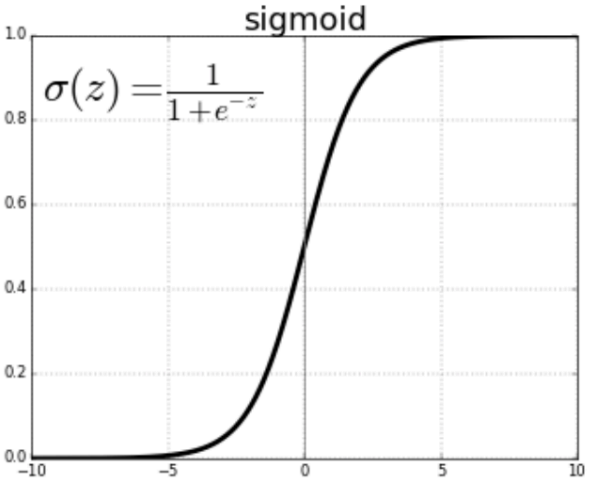
- **differentiable:** we can find the slope of the tanh curve at any two points
- **monotonic**, but its derivative is not

Negative input is mapped to **strongly negative values** and zero input is mapped to **near zero**

The ReLU activation function

- **Range:** (0, ∞)





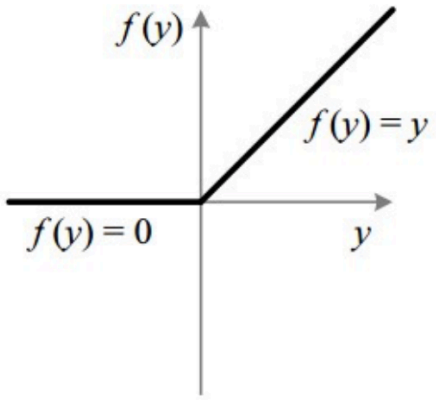
Both the function and its derivative are **monotonic**

Problem: negative input turns the value into zero immediately, hence not mapping the negative values appropriately

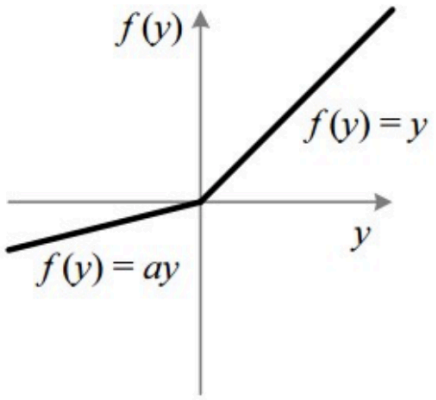
The leaky/randomized ReLU function

- **Range:** (-∞, ∞)

ReLU

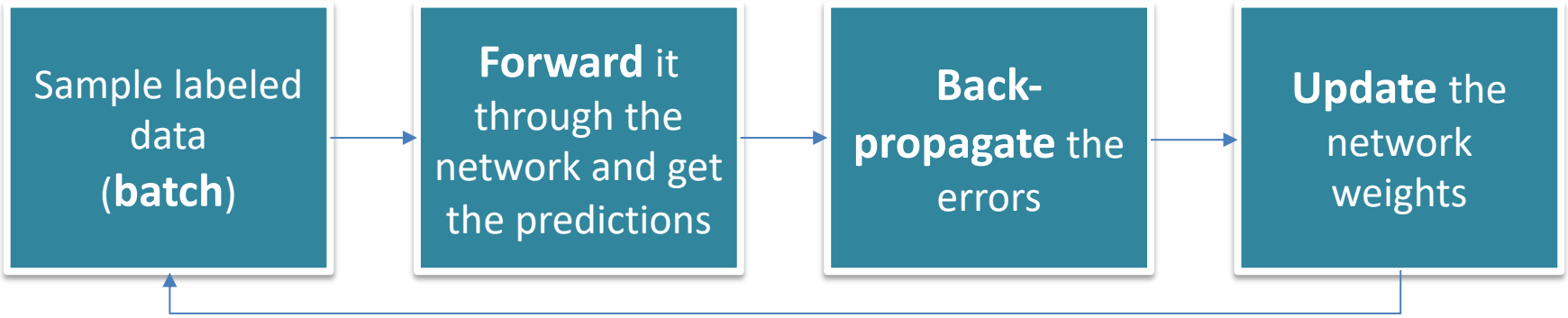


Leaky ReLU

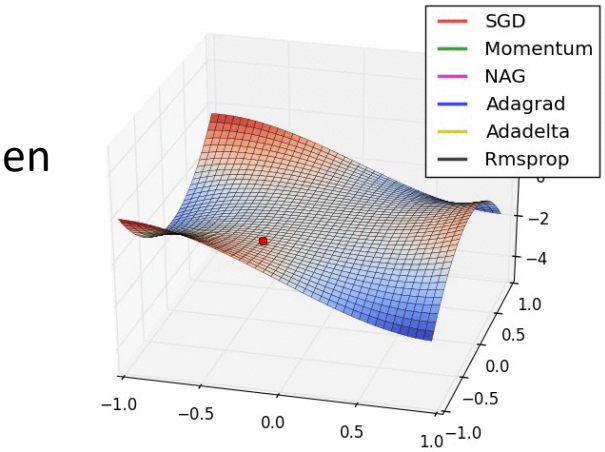


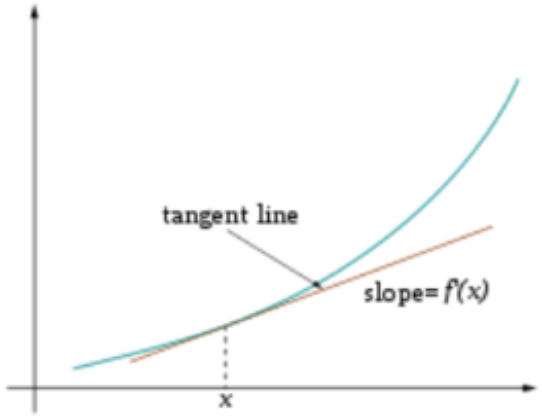
- The leak helps to increase the range of the ReLU function
- Usually **a** is 0.01 (leaky) or it may have any other value (randomized)
- Leaky ReLU functions are monotonic in nature
- Also, their derivatives also monotonic in nature

Training Neural Nets



Optimize (min or max) **objective/cost function** $J(\theta)$
Generate **error signal** that measures the difference between the predictions and the target values

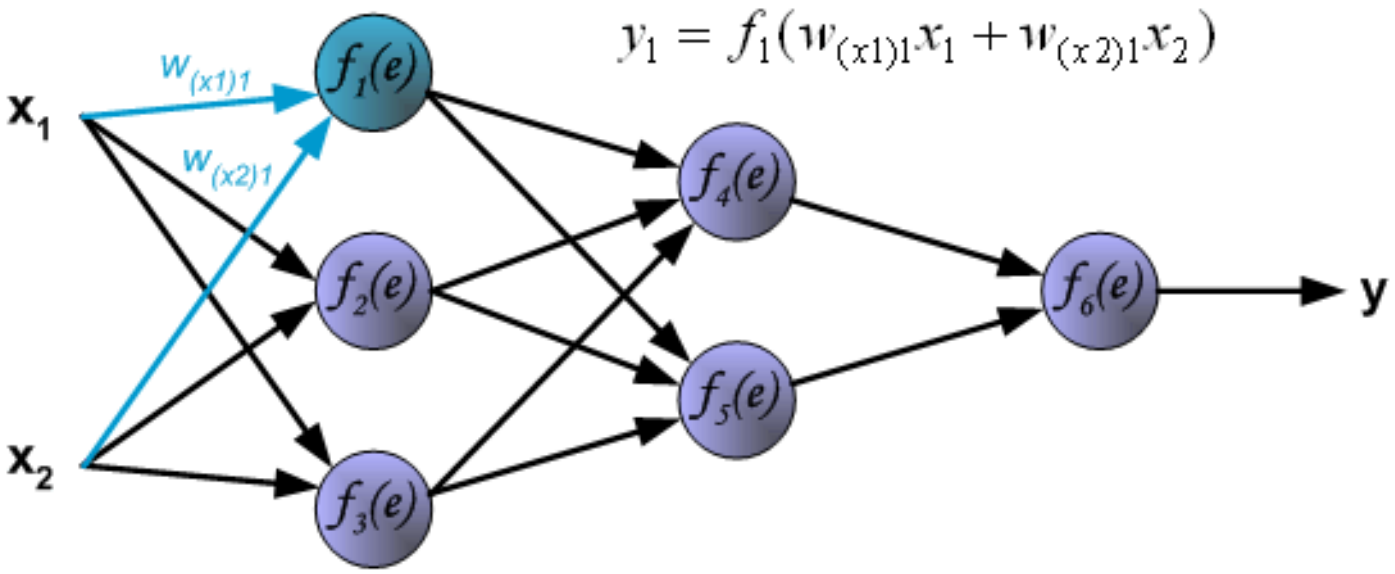




Use error signal to change the **weights** and get more accurate predictions
Subtracting a fraction of the **gradient** moves you towards the **(local) minimum of the cost function**
Epoch: one pass over the training set

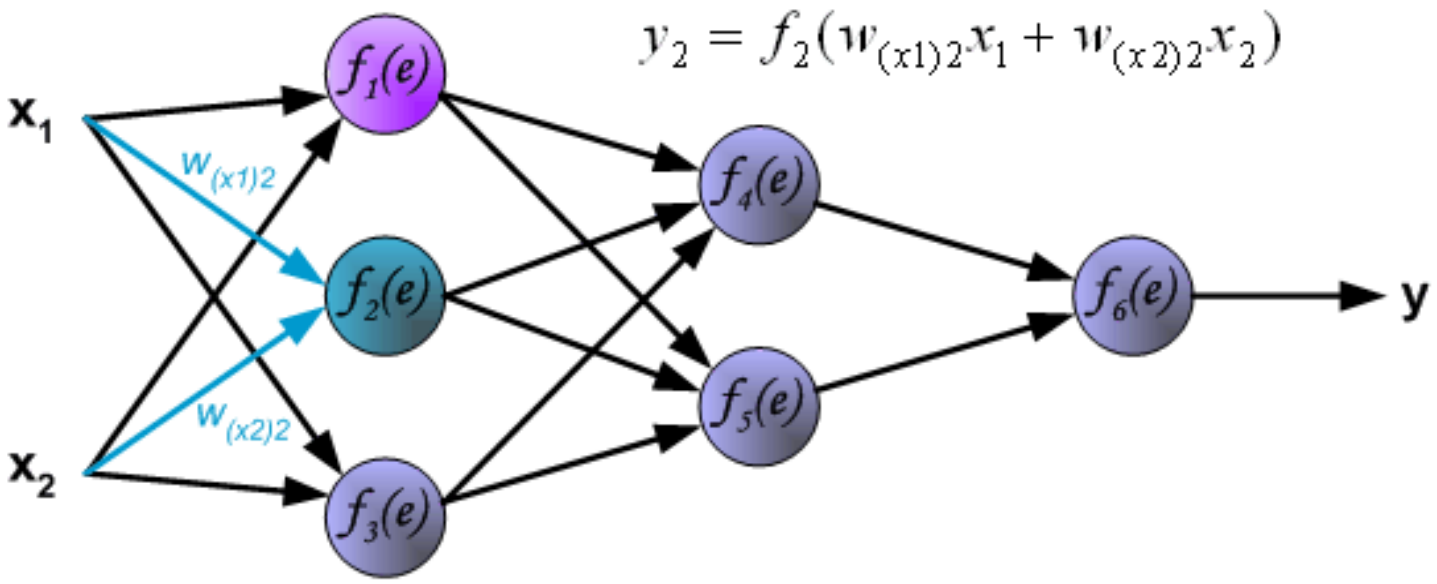
Learning Algorithm: forward propagation

- The “signal” is propagating through the network
- $w_{(xm)n}$: weights of connections between network input x_m and neuron n in input layer
- y_n: represents output signal of neuron n



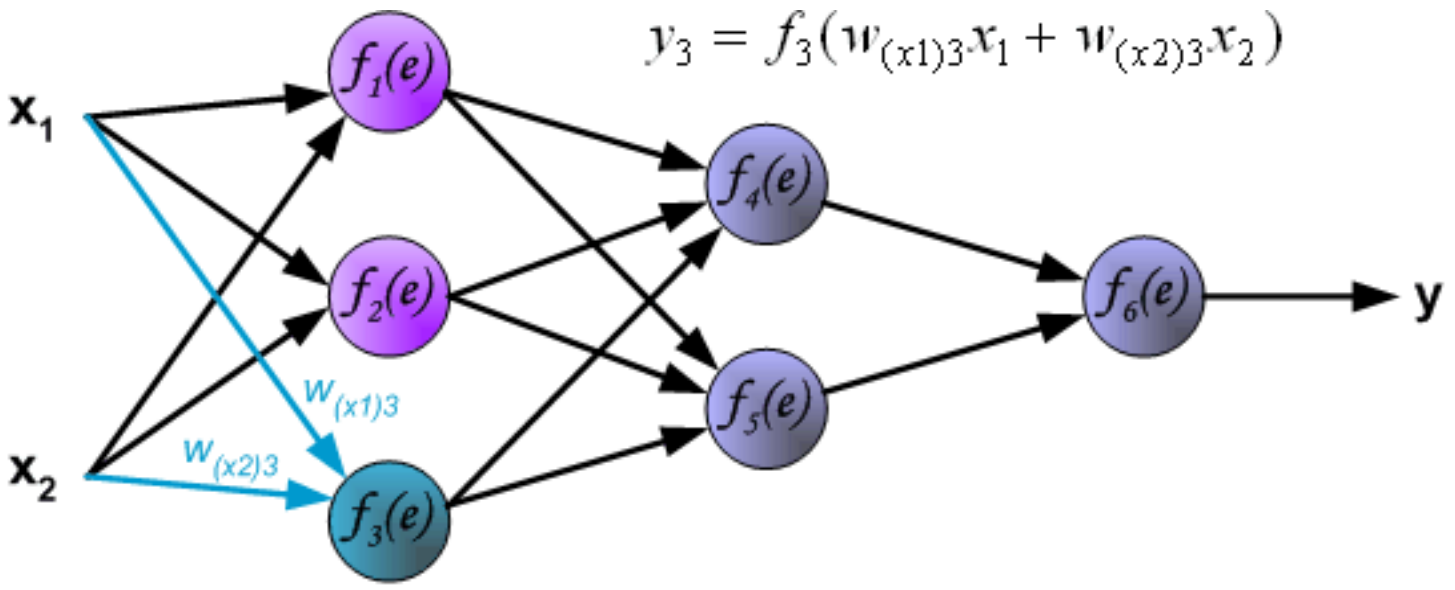
Learning Algorithm: forward propagation

- The “signal” is propagating through the network
- $w_{(xm)n}$: weights of connections between network input x_m and neuron n in input layer
- y_n: represents output signal of neuron n



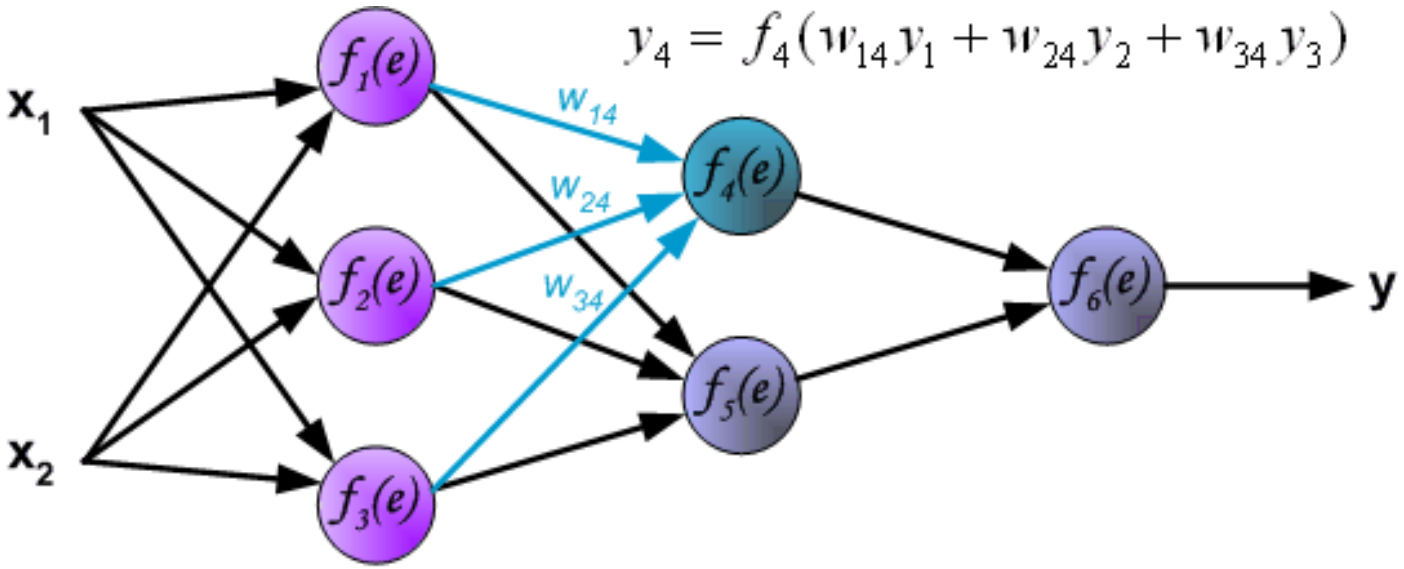
Learning Algorithm: forward propagation

- The “signal” is propagating through the network
- $w_{(xm)n}$: weights of connections between network input x_m and neuron n in input layer
- y_n: represents output signal of neuron n



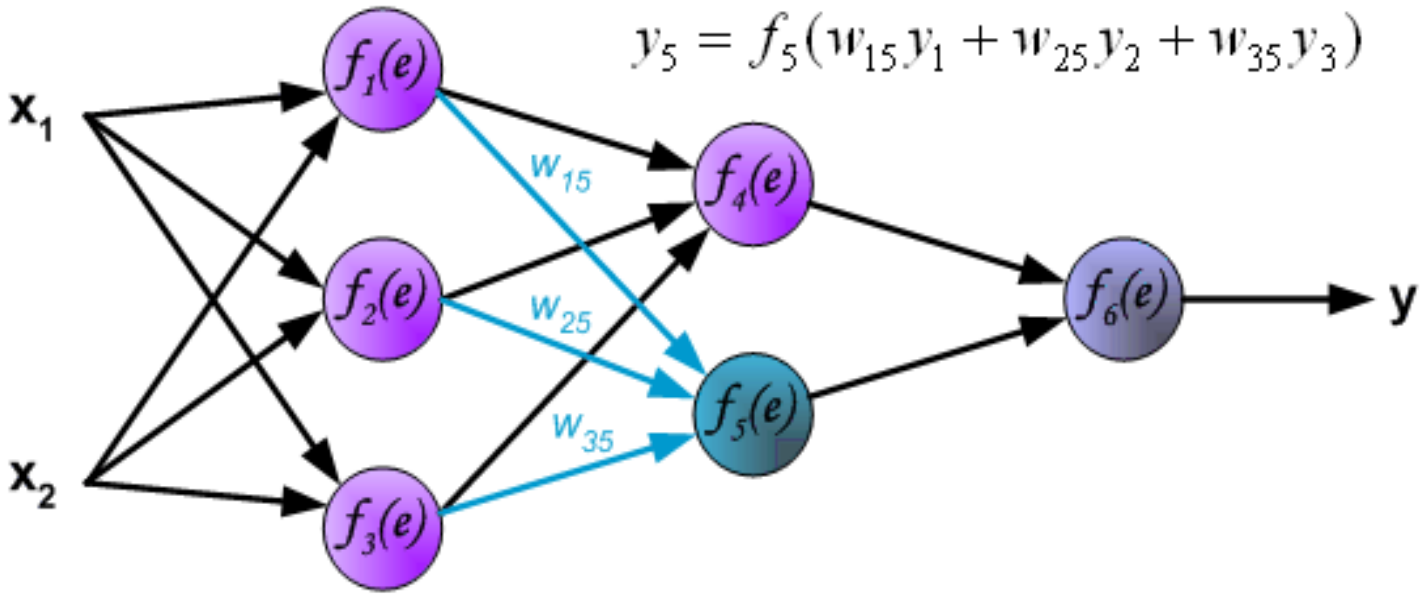
Learning Algorithm: forward propagation

- Propagation of signals through the hidden layer
- w_{mn}: weights of connections between output of neuron ***m*** and input of neuron ***n*** in the next layer



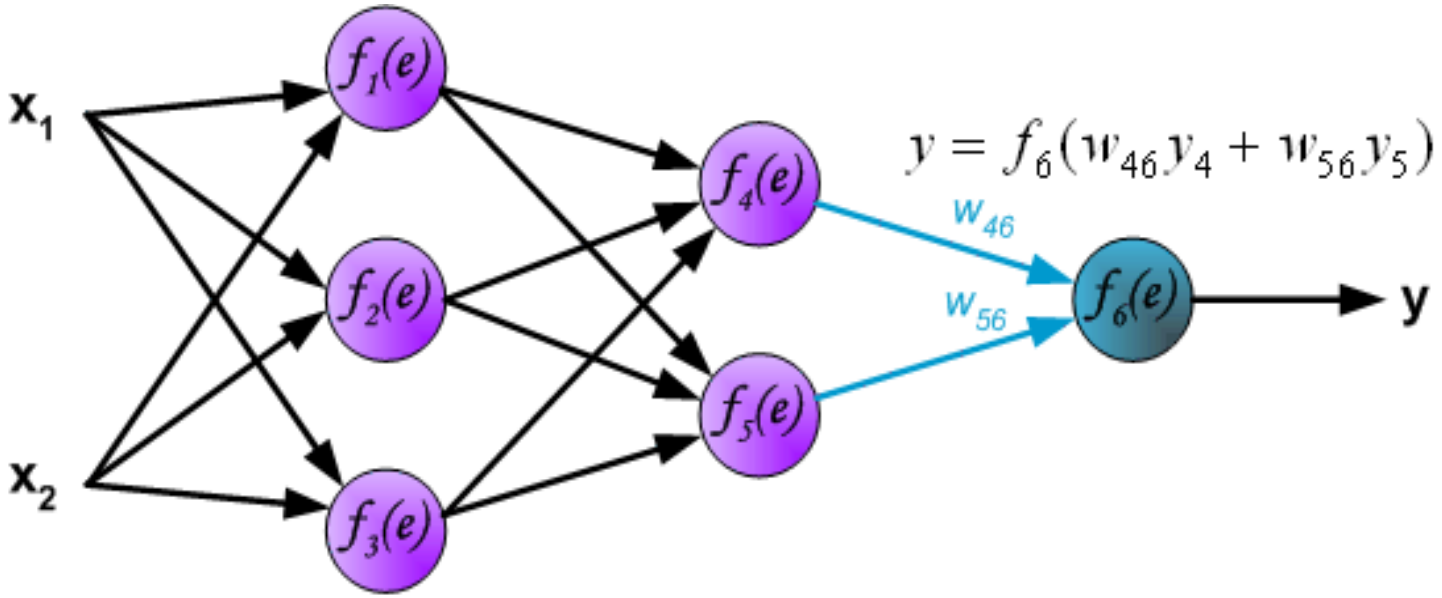
Learning Algorithm: forward propagation

- Propagation of signals through the hidden layer
- w_{mn}: weights of connections between output of neuron ***m*** and input of neuron ***n*** in the next layer



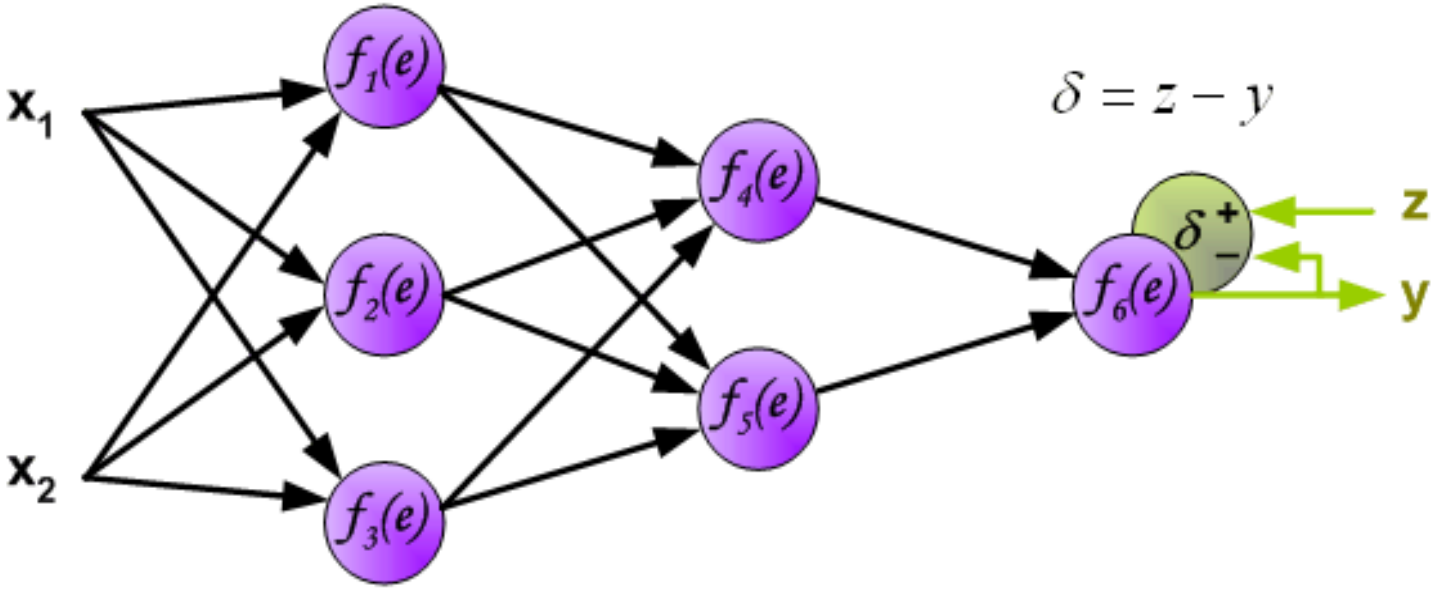
Learning Algorithm: forward propagation

- Propagation of signals through the hidden layer
- w_{mn}: weights of connections between output of neuron ***m*** and input of neuron ***n*** in the next layer

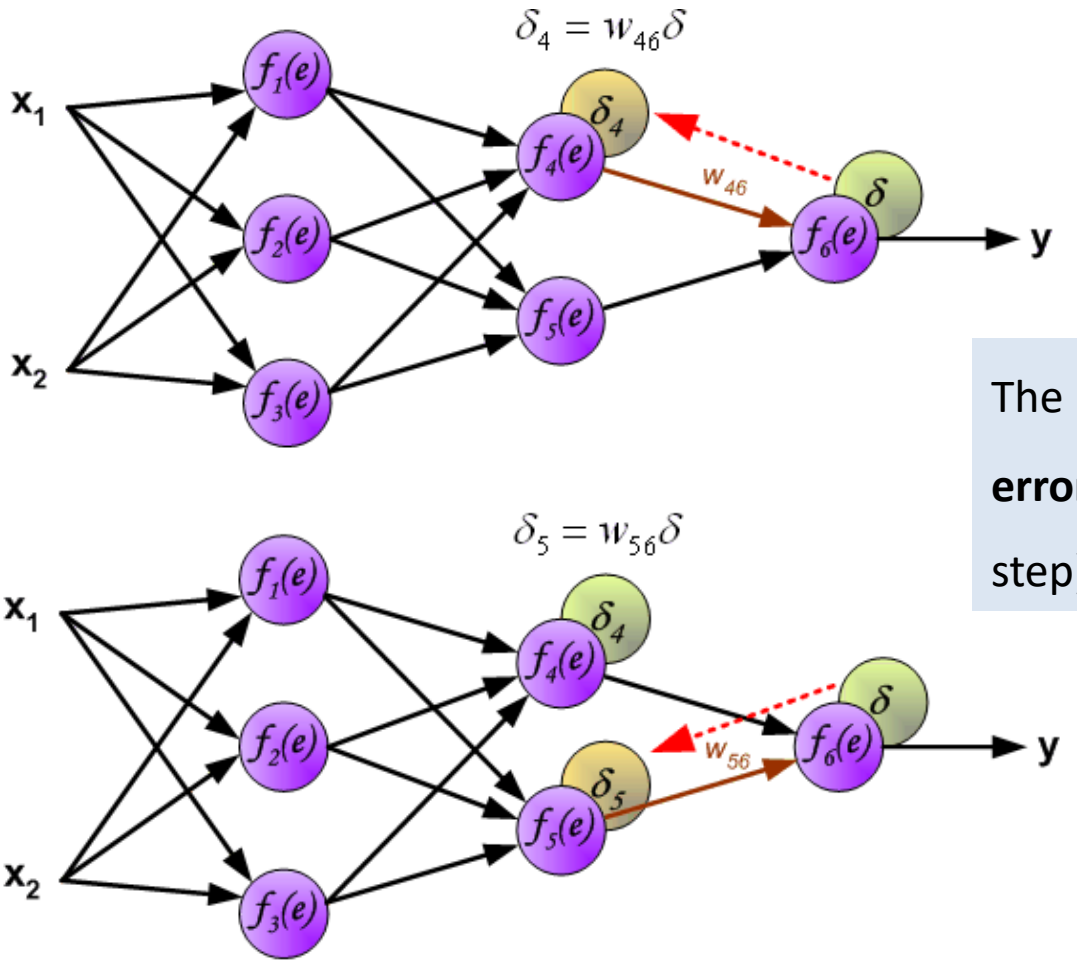


Learning Algorithm: forward propagation

- The output signal of the network ***y*** is compared with the desired output value ***z*** (the target), which is found in the training data set
- The difference is called error signal δ of output layer neuron

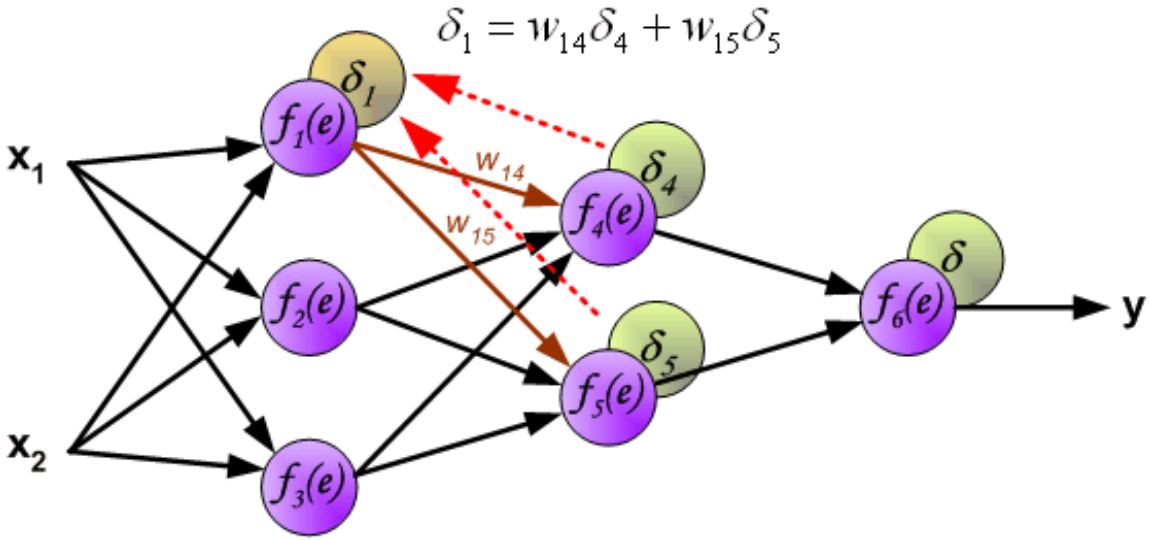


Learning Algorithm: backpropagation



The idea is to propagate the **error** ***δ*** (computed in single teaching step) back to all neurons

Learning Algorithm: backpropagation

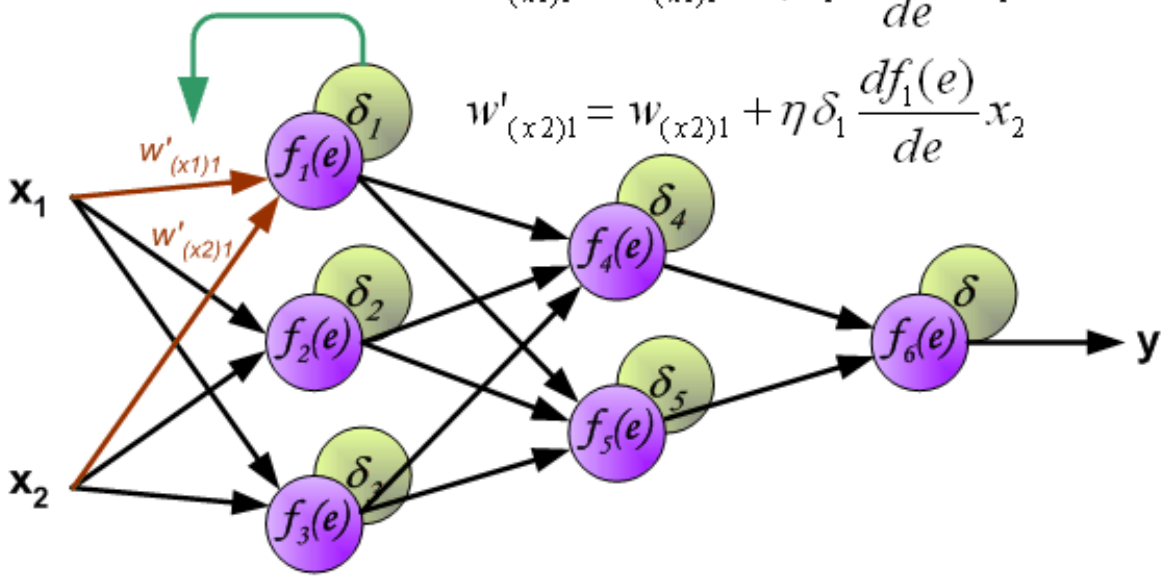


- The weight coefficients w_{mn} used to propagate errors back are equal to this used during computing output value
- Only the direction of data flow is changed (signals are propagated from output to inputs one after the other)
- This technique is used for all network layers
- If propagated errors came from many neurons, they are added

Learning Algorithm: backpropagation

$$w'_{(x1)1} = w_{(x1)1} + \eta \delta_1 \frac{df_1(e)}{de} x_1$$

$$w'_{(x2)1} = w_{(x2)1} + \eta \delta_1 \frac{df_1(e)}{de} x_2$$

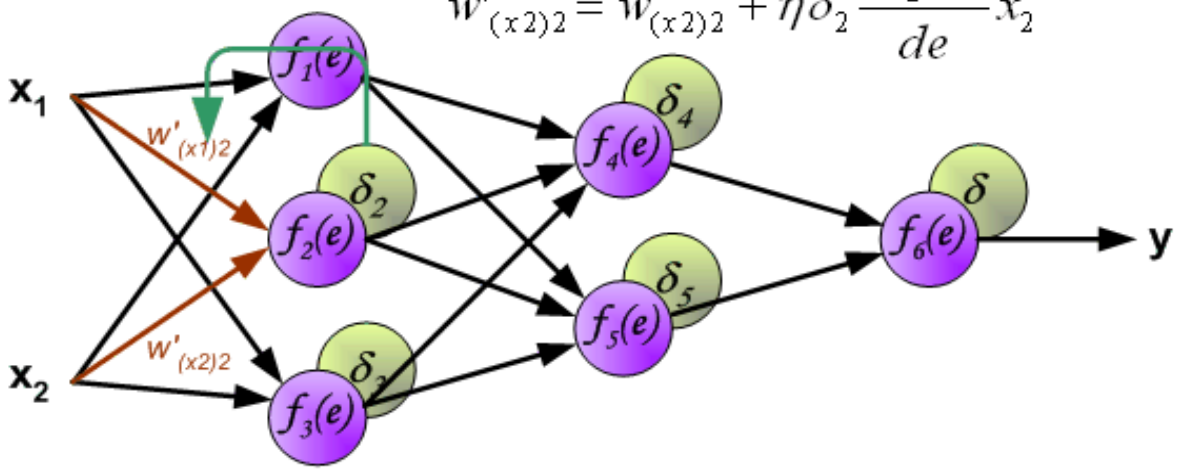


- When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified
- *df(e)/de* is the derivative of the neuron activation function (weights are modified)
- ***η*** is the learning rate (default 0.01): smaller ***η*** requires more epochs

Learning Algorithm: backpropagation

$$w'_{(x1)2} = w_{(x1)2} + \eta \delta_2 \frac{df_2(e)}{de} x_1$$

$$w'_{(x2)2} = w_{(x2)2} + \eta \delta_2 \frac{df_2(e)}{de} x_2$$

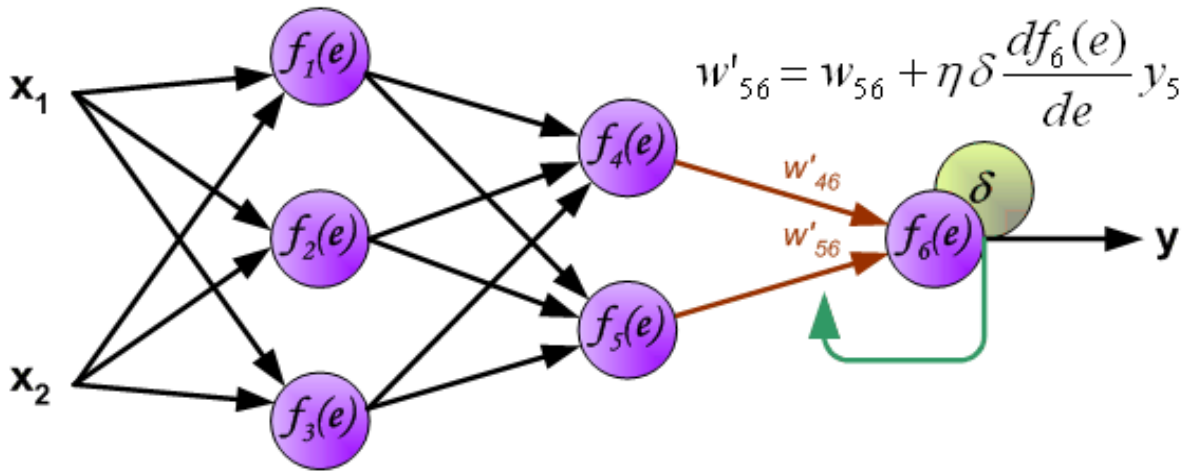


- When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified
- *df(e)/de* is the derivative of the neuron activation function (weights are modified)
- ***η*** is the learning rate (default 0.01): smaller ***η*** requires more epochs

Learning Algorithm: backpropagation

$$w'_{46} = w_{46} + \eta \delta \frac{df_6(e)}{de} y_4$$

$$w'_{56} = w_{56} + \eta \delta \frac{df_6(e)}{de} y_5$$



- When the error signal for each neuron is computed, the weights coefficients of each neuron input node may be modified
- *df(e)/de* is the derivative of the neuron activation function (weights are modified)
- ***η*** is the learning rate (default 0.01): smaller ***η*** requires more epochs

Cost functions

- The main goal of training a neural network:
  - finding the **parameters θ** that significantly reduce a **cost function J(θ)**
- **J(θ):** Typically includes a **performance measure** evaluated on the **entire training set** as well as additional *regularization terms*
- Taken over the data-generating distribution p_{data}

$$J^*(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim p_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

Why not use this one?

If we knew the actual data distribution, this would be a standard optimization problem!

Now it is a machine learning optimization problem: we only have training data ☺

A typical cost function J

- Written as an <u>average over the training set</u>

$$J(\boldsymbol{\theta}) = \mathbb{E}_{(\boldsymbol{x},\mathrm{y})\sim\hat{p}_{\mathrm{data}}} L(f(\boldsymbol{x};\boldsymbol{\theta}), y)$$

- *L:* per example loss function
- *f:* is the predicted output for input x
- defined over the empirical distribution, i.e., the training set

$$\mathbb{E}_{\boldsymbol{x},\mathrm{y}\sim\hat{p}_{\mathrm{data}}(\boldsymbol{x},y)}[L(f(\boldsymbol{x};\boldsymbol{\theta}), y)] = \frac{1}{m}\sum_{i=1}^{m} L(f(\boldsymbol{x}^{(i)};\boldsymbol{\theta}), y^{(i)})$$

Empirical risk minimization: training process for minimizing the average training error

Problems: overfitting + differentiation

- **Problem 1:**
  - gradient descent: basis for most modern optimization algorithms
  - does not work for 0-1 loss functions: the derivatives cannot be computed
  - **solution:** surrogate loss functions
- **Problem 2:**
  - empirical risk minimization is prone to overfitting ☹
  - models with high capacity can simply memorize the training set
  - **solution:** regularization

Loss functions

- **Regression loss functions:**
  - MSE
  - MAE
  - Huber loss
- **Classification loss functions:**
  - 0-1 loss
  - Binary Cross-Entropy
  - Categorical Cross-Entropy

Regression loss functions

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

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y^{(i)} - \hat{y}^{(i)}|$$

Huber loss:

- If the absolute difference between the actual and predicted value is less than or equal to a threshold value, δ, then MSE is applied
- if the error is sufficiently large, then MAE is applied

$$Huber\ Loss = \begin{cases} \frac{1}{n}\sum_{i=1}^{n}(y^{(i)} - \hat{y}^{(i)})^2 & |y^{(i)} - \hat{y}^{(i)}| \leq \delta \\ \frac{1}{n}\sum_{i=1}^{n}\delta(|y^{(i)} - \hat{y}^{(i)}| - \frac{1}{2}\delta) & |y^{(i)} - \hat{y}^{(i)}| > \delta \end{cases}$$

0-1 loss

$$\mathbf{0-1\ Loss} = -\frac{1}{n}\sum_{i=1}^{n}\delta_{(\hat{y}_i \neq y_i)} \qquad \delta_{(\hat{y}_i \neq y_i)} = \begin{cases} 0, & if\ \hat{y}_i \neq y_i \\ 1, & \text{otherwise} \end{cases}$$

Practically counts for how many class labels the predictions are correct

- The problem with the 0-1 loss function is that it is not differentiable
- Hence it cannot be optimized efficiently

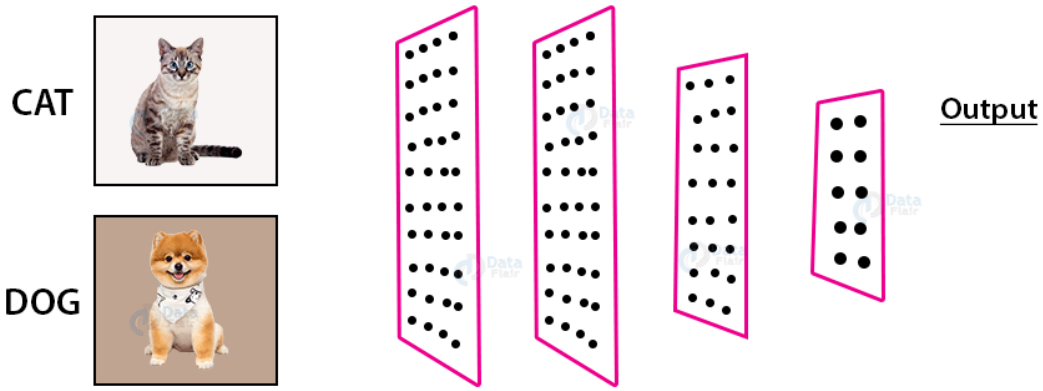
Solution: consider alternative functions that can be optimized, also referred to as *surrogate loss functions*

Binary cross entropy

$$\boldsymbol{BCE\ Loss} = -\frac{1}{n}\sum_{i=1}^{n}(y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i))$$

Classification neural networks output a vector of probabilities $\hat{y}_i$

- $\hat{y}_i$: the predicted probability that the class label is 1
- $1 - \hat{y}_i$: the predicted probability that the class label is 0



Outcome:

- 1 (there is a cat)
- 0 (there is no cat)

Case 1: The NN is 80% confident that the first image contains a cat

Case 2: The NN is 10% confident that the second image contains a cat

$$\boldsymbol{BCE\ Loss_1} = -(1 \log(0.8) + (1 - 1) \log(1 - 0.8)) = 0.32$$

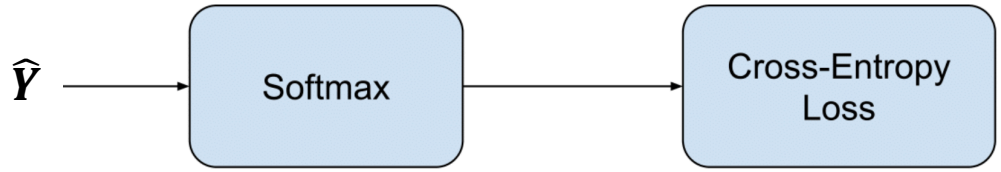
$$\boldsymbol{BCE\ Loss_2} = -(0 \log(0.1) + (1 - 0) \log(1 - 0.1)) = 0.15$$

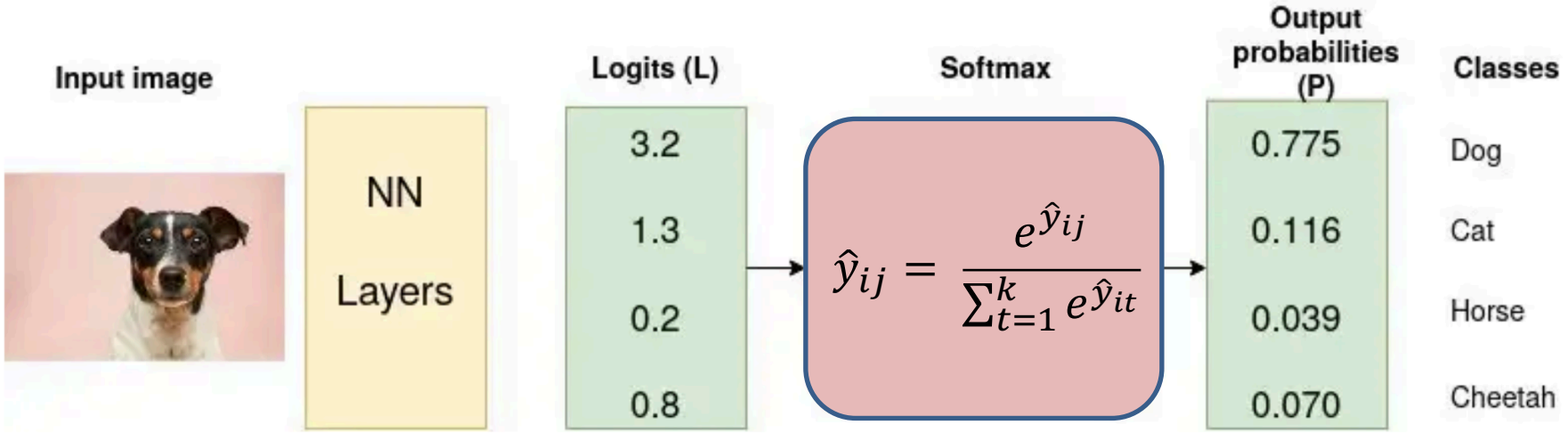
Categorical cross entropy

$$\boldsymbol{CCE\ Loss} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{k} y_{ij} \cdot \log\left(\hat{y}_{ij}\right)$$

Classification neural networks output a vector of probabilities $\hat{y}_{ij}$

- $\hat{\boldsymbol{y}}_{\boldsymbol{ij}}$: the predicted probability that *example i* is of *class label* is *j*
- Also referred to as ***softmax loss***





Regularization

Regularization: L2 or L1

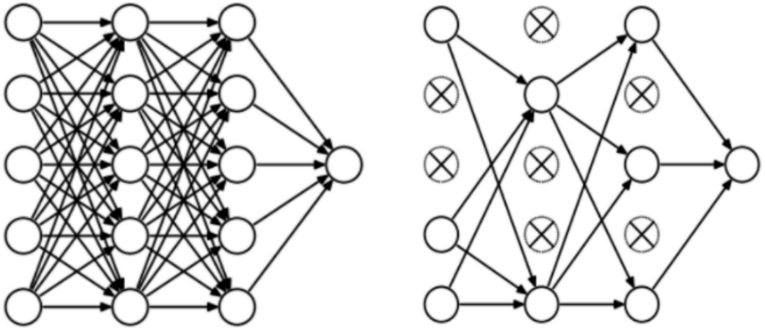
L2 regularization (or weight decay)

- regularization term that penalizes big weights, added to the objective
- weight decay value determines how dominant regularization is during gradient computation
- big weight decay coefficient → *big penalty for big weights*

$$J_{reg}(\theta) = J(\theta) + \frac{\lambda}{2}\sum_{k}\theta_k^2$$

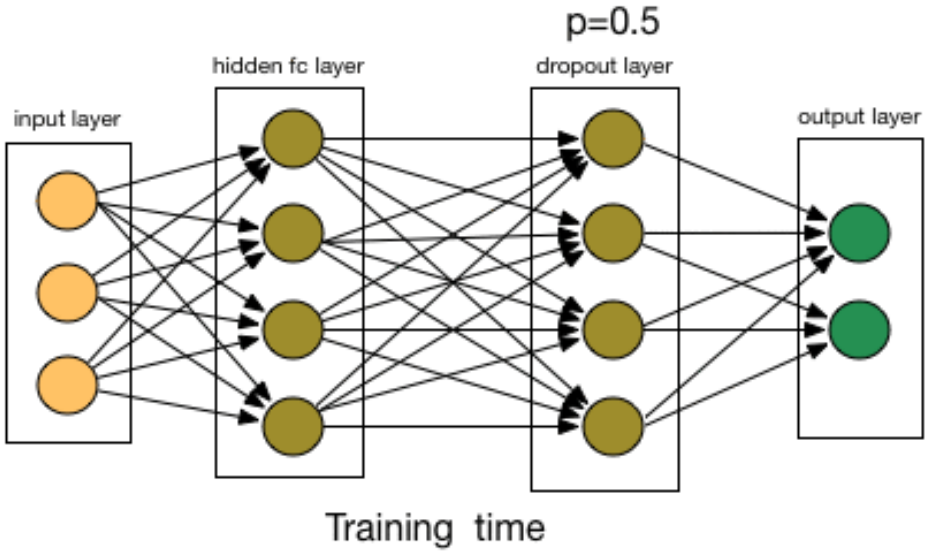
- **How about L1?**
  - not prefered as it will end up compressing the network
- **Why dividing by 2?**
  - to make the derivative computation easier

Regularization: dropout



Dropout

- randomly drop units (along with their connections) during training (at each iteration)
- each unit retained with fixed probability ***p***, independent of other units
- hyper-parameter ***p*** to be chosen (tuned)

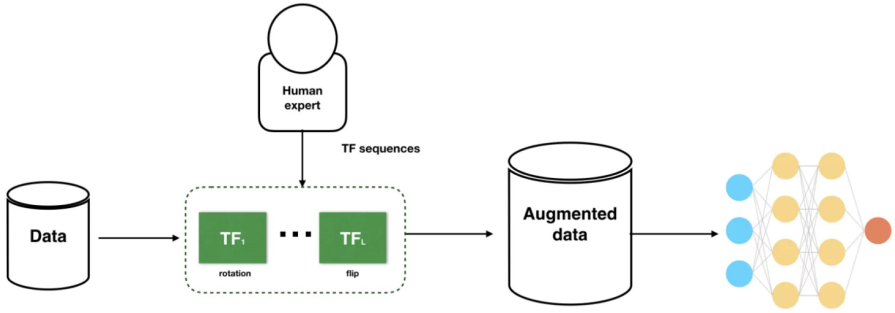


- **It can also be thought of as an ensemble technique**
- Ensemble models usually perform better than a single model as they capture more randomness
- Similarly, dropout also performs better than a normal neural network model
- Dropout can be applied to both the hidden layers as well as the input layers

Regularization: data augmentation

Data augmentation

- Increase the size of the training data when possible
- All a matter of defining data augmentation operations



- May result in improved predictive performance
- Can reduce cost of collecting and labeling data
- Enables rare event prediction
- Prevents data privacy issues

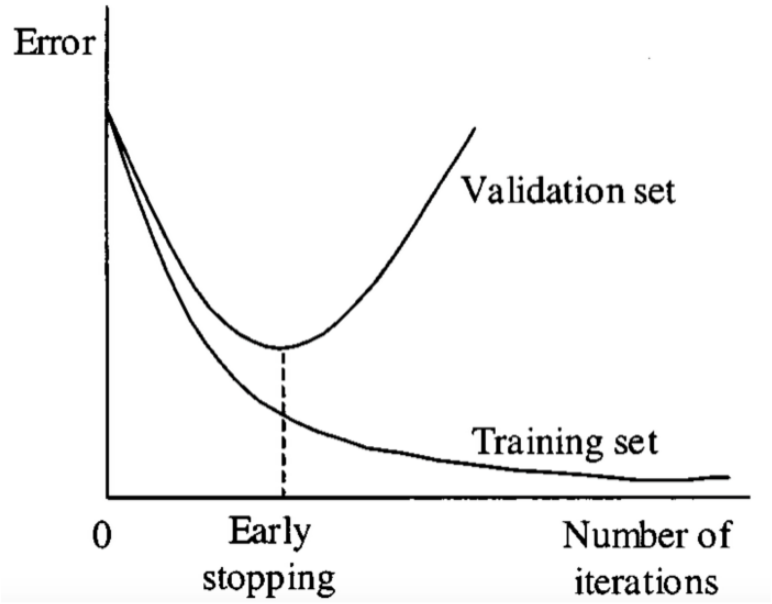
What about text?

- Easy Data Augmentation (EDA) operations: synonym replacement, word insertion, swap, and deletion
- Back translation: re-translating text from the target language back to its original language
- Contextualized word embeddings

Regularization: early-stopping

Early-stopping

- use validation error to decide when to stop training
- stop when the monitored quantity has not improved after ***n*** subsequent epochs
- ***n*** is called *patience*

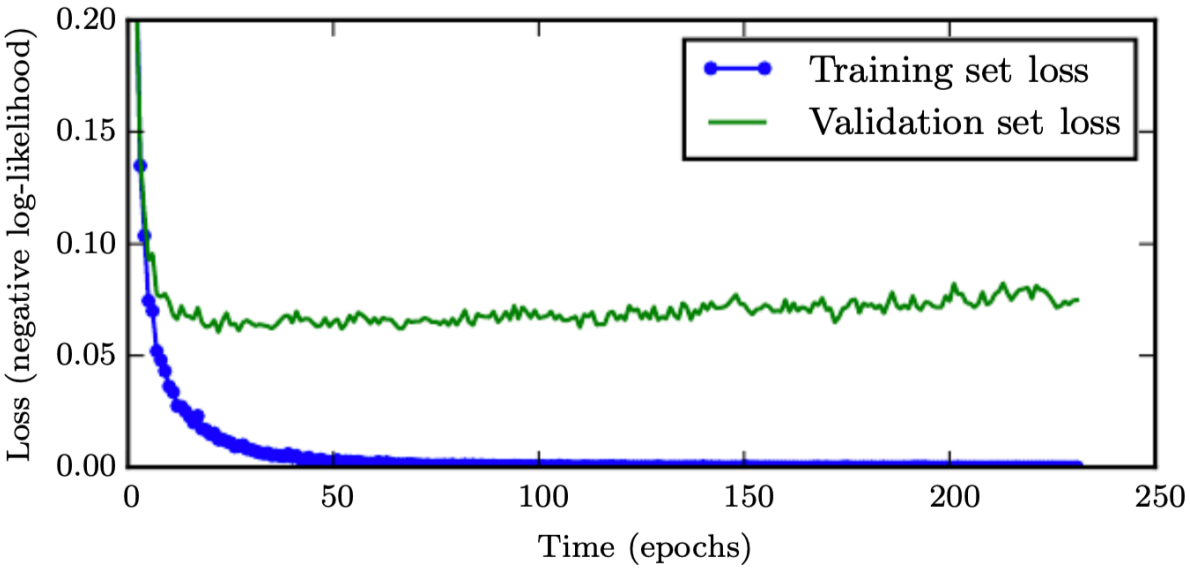


In essence…

- the current best model parameters are stored and updated during training
- when parameter updates no longer yield an improvement (after *n* iterations) the training is stopped and the last best parameters is used
- it works as a regularizer by restricting the optimization procedure to a smaller volume of parameter space

Trainning error down to zero

- **Typical situation:** loss converges after a set of epochs



- It can be the case that the training error drops to 0 while validation error may keep improving
- How so?

Validation error too high

- Training error zero
- Validation error remains high
- **Overfitting:**
  - model is too complex: reduce layer(s) and hidden nodes
  - apply early stopping and dropout
  - set a hard limit on trainning performance (eg 1-2%)
  - increase training examples
  - use regularization to penalize weights if they get too large

Stochastic gradient descent

Algorithm 8.1 Stochastic gradient descent (SGD) update

Require: Learning rate schedule $\epsilon_1, \epsilon_2, \ldots$

Require: Initial parameter $\boldsymbol{\theta}$

$k \leftarrow 1$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon_k \hat{\boldsymbol{g}}$

$k \leftarrow k + 1$

end while

$\boldsymbol{\varepsilon_0}$: higher than the learning rate that yields the best performance after the first, e.g., 100 iterations

- monitor the first several iterations
- use a learning rate that is higher than the best-performing learning rate at this time
- SGD can have high variance…why?

Stochastic gradient descent (SGD)

- Obtain an unbiased estimate of the gradient by taking the average gradient on a mini-batch of m examples drawn i.i.d from the data-generating distribution
- Learning rate $\boldsymbol{\varepsilon_k}$ is variable until the total gradient becomes small
- Decay learning rate until iteration τ
- τ: set to the number of iterations required to make a few hundred passes through the training set
- $\varepsilon_\tau = 0.01\ \varepsilon_0$

$$\sum_{k=1}^{\infty} \epsilon_k = \infty, \quad \text{and} \quad \sum_{k=1}^{\infty} \epsilon_k^2 < \infty.$$

$$\epsilon_k = (1-\alpha)\epsilon_0 + \alpha\epsilon_\tau \qquad \alpha = \frac{k}{\tau}$$

After *iteration* τ, it is common to leave $\boldsymbol{\varepsilon}$ constant

SGD with momentum

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α

Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$

while stopping criterion not met **do**

Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.

Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$.

Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$.

Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$.

end while

The larger α is relative to ε, the more the previous gradients affect the current direction

- The momentum algorithm accumulates an exponentially decaying moving average of past gradients
- Stochastic gradient descent with momentum remembers the weight update at each iteration, and determines the next update as a linear combination of the gradient and the previous update

Initialization of weights

- If two hidden units with the same activation function are connected to the same inputs, then these units must have different initial parameters
- If they have the same initial parameters, a learning algorithm applied to a deterministic cost function will constantly update both units in the same way
- **Solution:** initialize each unit to compute a different function from all the other units
- **Weights:** Gaussian or uniform distribution; the choice does not seem to matter much
- **Heuristic:** initialize the weights of a fully connected layer with m inputs and n outputs by sampling each weight from:

$$U\left(-\frac{1}{\sqrt{m}}, \frac{1}{\sqrt{m}}\right) \quad \text{or} \quad U\left(-\sqrt{\frac{6}{m+n}}, \sqrt{\frac{6}{m+n}}\right)$$

Coming up next…

Jan 16	Introduction to machine learning
Jan 18	Regression analysis
Jan 19	Laboratory session 1: numpy and linear regression
Jan 23	Ensemble learning
Jan 25	Deep learning I: Training neural networks
Jan 26	**Laboratory session 2: ML pipelines, ensemble learning**
Jan 30	**Deep learning II: Convolutional neural networks**
Feb 1	Laboratory session 3: training NNs and tensorflow
Feb 6	Deep learning III: Recurrent neural networks
Feb 8	Laboratory session 4: CNNs and RNs
Feb 13	Deep learning IV: Autoencoders, transformers, and attention
Feb 20	Time series classification

TODOs

- **Reading:**
  - The Deep Learning Book: Chapters 6, 8
- **Homework 1**
  - out: Jan 26
  - due: Feb 5