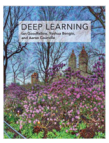
IL2230 Lecture 3
Artificial Neural Network:
From Perceptron to MLP

Prof. Zhonghai Lu
KTH Royal Institute of Technology

Main reference:
Chapter 5 Neural networks

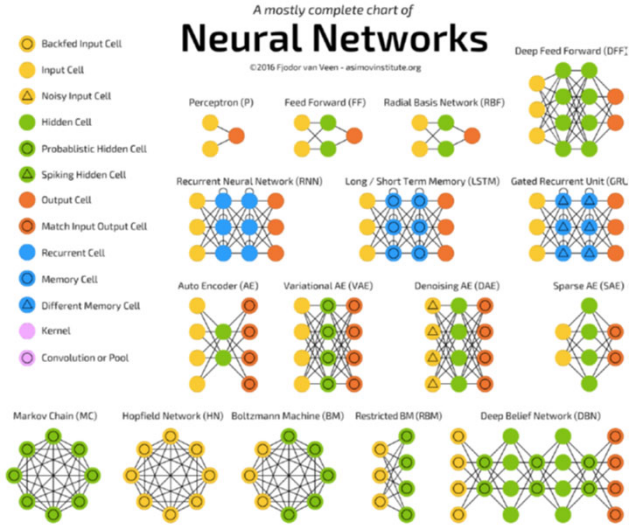


Part II. Deep Networks:
Modern Practices

Module I Deep Learning

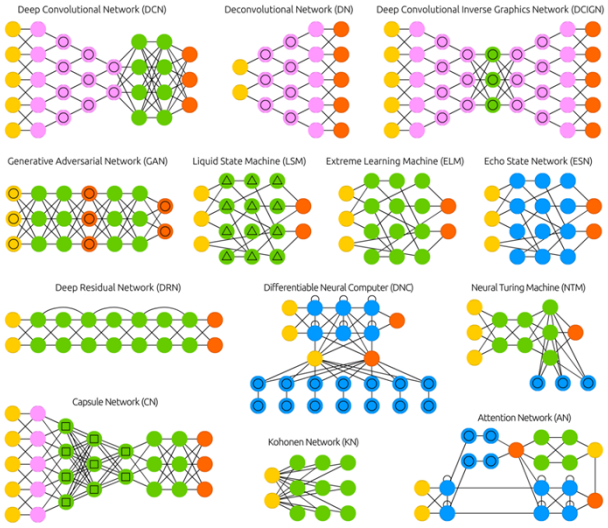
- L2 Linear and Logistic Regression
  - Linear regression, Polynomial regression
  - Logistic regression
  - Under-fitting, overfitting, regularization
- **L3 Artificial Neural Network (ANN)**
  - Perceptron: From biological neuron to artificial neuron model
  - Feedforward DNN: Multi-Layer Perceptron (MLP)
  - Learning: SGD (stochastic gradient descent) and BP (back propagation)
- L4 Convolutional Neural Network (CNN)
  - The curse of dimensionality and countermeasures
  - CNN layers
- L5 Recurrent Neural Network (RNN)
  - Basic RNN
  - Long Term Short Term Memory (LSTM)

The neural network zoo



http://www.asimovinstitute.org/neural-network-zoo/

The neural network zoo



http://www.asimovinstitute.org/neural-network-zoo/

Agenda

- Neuron and perceptron
  - Biological neuron
  - Artificial neuron
  - Perceptron
  - The XOR issue Perceptron cannot deal with XOR (Read literature)
- Multi-Layer Perceptron (MLP)
  - Error-based learning
    - Error Back-propagation algorithm
    - Stochastic gradient descent (SGD)
  - Inference
  - Counter-measures for overfitting
  - Pros & Cons
- Multiple hidden layers (Deep learning)

ANN

- The inventor of the first neurocomputer, Dr. Robert Hecht-Nielsen, defines **a neural network** as –"...*a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs*."
- The idea of ANNs is based on the belief that working principle of human brain by making the right connections, can be imitated using silicon and wires as living neurons and dendrites.

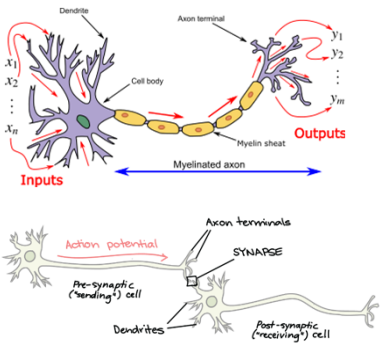
ANN and brain

- An ANN is based on a collection of connected units or nodes called **artificial neurons** which loosely model the neurons in a biological brain.
- Each connection, like the **synapses** in a biological brain, can transmit a signal from one artificial neuron to another.
- An artificial neuron that receives a signal can process it and then signal additional artificial neurons connected to it.

Source: Wikipedia

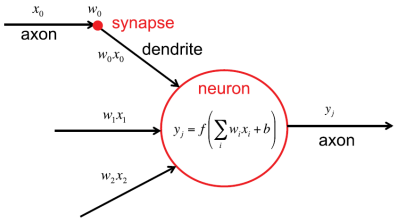
Bio-neuron structure vs. Artificial neuron model

- An artificial neuron mimics the working of a biophysical neuron, but is not a biological neuron model.
- An artificial neural network is a composition of simple elements called **artificial neurons**, which receive input, perform simple calculation, and produce output.



$$y = f\left(\sum_{i=1}^{n} x_i w_i + b\right)$$

Multiply-Accumlator (MAC)

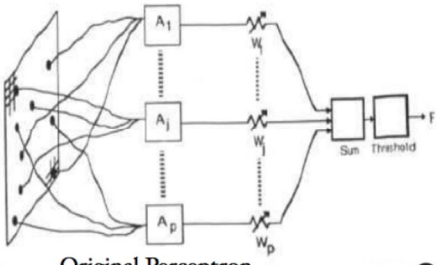


Wikipedia

Sze et al. Proceedings of IEEE. Dec. 2017

The perceptron (1962)

- Perceptron is a mathematical or conceptual model of a biological neuron.

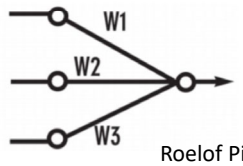


Original Perceptron

(From Perceptrons by M. L Minsky and S. Papert, 1969, Cambridge, MA: MIT Press. Copyright 1969 by MIT Press.

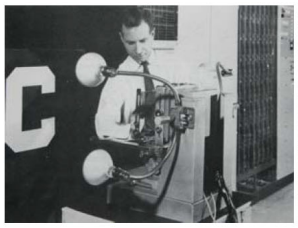
Frank Rosenblatt
(1928-1971)

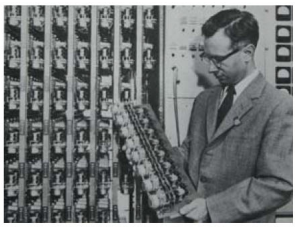
Simplified model:



Roelof Pieters, Jan 2015

The Mark 1 perceptron hardware





Left: It shows how the inputs were obtained using a simple camera system in which an input scene, in this case a printed character, was illuminated by powerful lights, and an image focused onto a **20 × 20** array of cadmium sulphide photocells, giving a primitive 400 pixel image.

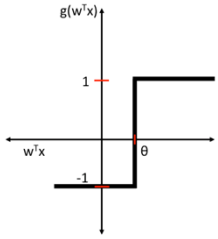
Middle: The perceptron also had a patch board, which allowed different configurations of input features to be tried. Often these were **wired up at random** to demonstrate the **ability of the perceptron to learn without the need for precise wiring**, in contrast to a modern digital computer.

Right: one of the racks of adaptive weights. Each weight was implemented using **a rotary variable resistor**, also called a potential meter, driven by an electric motor thereby allowing the value of the weight to be adjusted automatically by the learning algorithm.

The perceptron (1962)

- The perceptron is an artificial neuron model.
- A nonlinear function is essential to model arbitrarily complex functionality which is not achievable by linear combinations of any depth.
- The nonlinear activation function $f(\cdot)$ was given by a step function.

$$f(a) = \begin{cases} +1, & a \geqslant 0 \\ -1, & a < 0 \end{cases}$$



Unit step function.

Activation function

- The function f is non-linear and is called the Activation Function, also transfer function.
  - Purpose: the activation function is to introduce non-linearity into the output of a neuron.
  - Importance: because most real world data is non-linear and we want neurons to learn these non-linear representations.

Why not the step function?

- Since the learning strategy requires computation of the gradient of the error function at each iteration step, we must guarantee the **continuity and differentiability of the error function**.
- Obviously we have to use a kind of activation function other than **the step function** used in perceptrons. because the composite function produced by interconnected perceptrons is discontinuous, and therefore the error function too.
- One of the more popular activation functions for backpropagation networks is the **sigmoid**.

 R. Rojas: Neural Networks, Springer-Verlag, Berlin, 1996

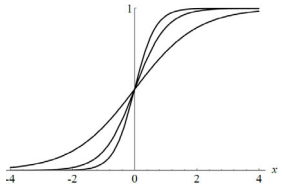
Logistic Sigmoid function

- The term "Sigmoid" means **S** shaped. It is called a 'squashing function' as it maps the whole real axis into a finite interval.
  - Tanh() is also S shaped.
- Logistic Sigmoid function: The constant **c** can be selected arbitrarily and its reciprocal **1/c** is called the temperature parameter in stochastic neural networks.

$$s_c(x) = \frac{1}{1 + \mathrm{e}^{-cx}}$$

- Higher values of c bring the shape of the sigmoid closer to that of the step function and in the limit $c \to \infty$, it converges to a step function at the origin.

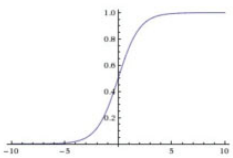
$$\frac{d}{dx} s(x) = \frac{\mathrm{e}^{-x}}{(1 + \mathrm{e}^{-x})^2} = s(x)(1 - s(x))$$



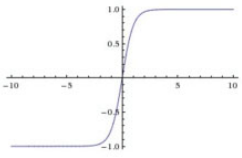
Three sigmoids (for $c = 1$, $c = 2$ and $c = 3$)

Common activation functions

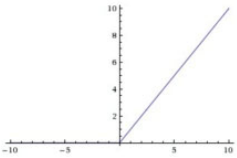
- Activation function



Sigmoid



tanh



ReLU

σ(x) = 1 / (1 + exp(−x))

Sigmoid()

tanh(x) = 2σ(2x) − 1

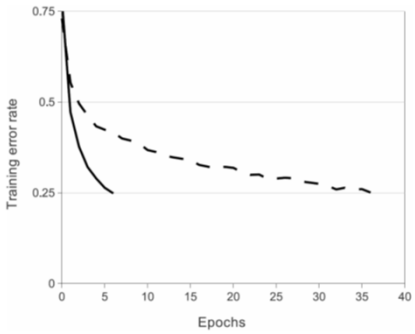
$$tanhx = \frac{sinhx}{coshx} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

f(x) = max(0, x)

ReLU stands for
Rectified Linear Unit

Why ReLU, not sigmoid or tanh?

- The standard way to model a neuron's output f as a function of its input x is with $f(x) = \tanh(x)$ or $f(x) = (1 + e^{-x})^{-1}$
- In terms of training time with gradient descent, these saturating nonlinearities are much slower than the non-saturating nonlinearity ReLu f(x) = max(0; x).



- A four-layer convolutional neural network with **ReLUs (solid line)** reaches a 25% training error rate on CIFAR-10 six times faster than an equivalent network with **tanh neurons (dashed line).**
- The learning rates for each network were chosen independently to make training as fast as possible.
- No regularization of any kind was employed. The magnitude of the effect varies with network architecture, but networks with ReLUs consistently learn several times faster than equivalents with saturating neurons.

One epoch means that the entire dataset is used for training the neural network once.

 Alex Krizhevsky et al. "ImageNet Classification with Deep Convolutional Neural Networks".

Activation functions

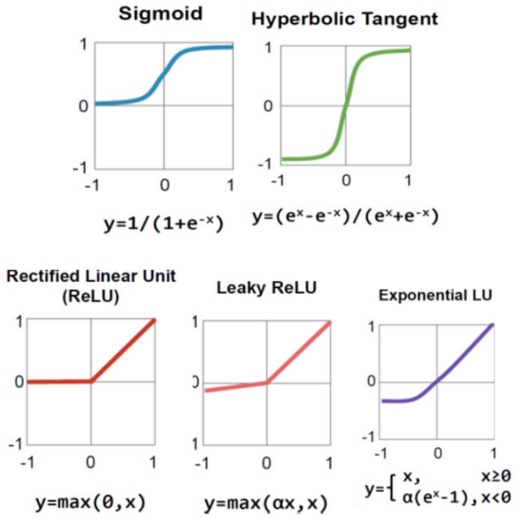


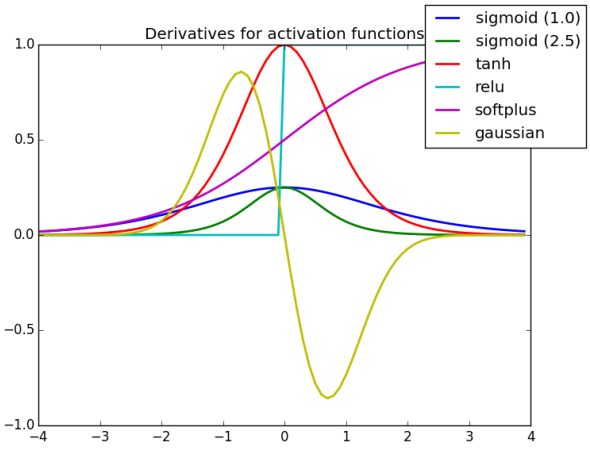
Image Source: Caffe Tutorial

Derivatives of activation functions

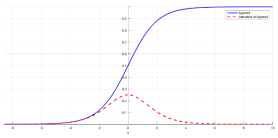
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1-f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6

Derivatives of activation functions



$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$



https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e

The Perceptron story

From Bishop's PRML book.

Frank Rosenblatt
1928–1969

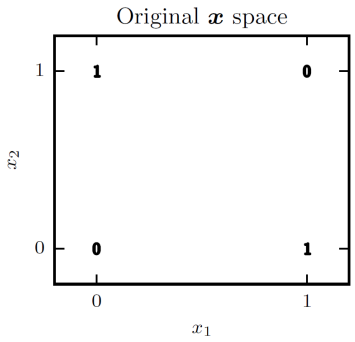
Rosenblatt's perceptron played an important role in the history of machine learning. Initially, Rosenblatt simulated the perceptron on an IBM 704 computer at Cornell in 1957, but by the early 1960s he had built special-purpose hardware that provided a direct, parallel implementation of perceptron learning. Many of his ideas were encapsulated in "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms" published in 1962. Rosenblatt's work was criticized by Marvin Minksy, whose objections were published in the book "Perceptrons", co-authored with Seymour Papert. This book was widely misinterpreted at the time as showing that neural networks were fatally flawed and could only learn solutions for linearly separable problems. In fact, it only proved such limitations in the case of single-layer networks such as the perceptron and merely conjectured (incorrectly) that they applied to more general network models. Unfortunately, however, this book contributed to the substantial decline in research funding for neural computing, a situation that was not reversed until the mid-1980s. Today, there are many hundreds, if not thousands, of applications of neural networks in widespread use, with examples in areas such as handwriting recognition and information retrieval being used routinely by millions of people.

- Frank Rosenblatt, "Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms", 1962.
- Marvin Minsky, Seymour Papert. "Perceptrons: An Introduction to Computational Geometry". MIT Presss, 1969.

Minsky (MIT) received the ACM Turing Award 1969.

The XOR issue

- The perceptron cannot solve the XOR problem, which is linearly not separable.



1 OR

There exists no line which can separate the outpu

MLP for the XOR problem

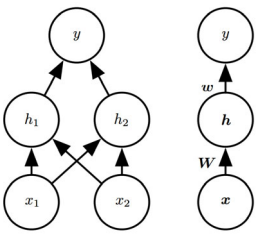
- What about adding one hidden layer?

The hidden layer:

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^\top \max\{0, \boldsymbol{W}^\top \boldsymbol{x} + \boldsymbol{c}\} + b.$$

Bias

ReLu



Biases omitted

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

Y

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

$$\boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix},$$

$$\boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

b=[0]

Calculation steps

- Batch processing: input vector => input matrix

x1 x2

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

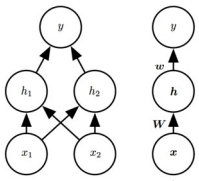
$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \quad \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \quad \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix},$$

1. Multiply the weight matrix W

$$\boldsymbol{XW} = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

2. Add the bias vector c

$$\begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$



Biases omitted

3. ReLU transformation

h1 h2

$$\begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

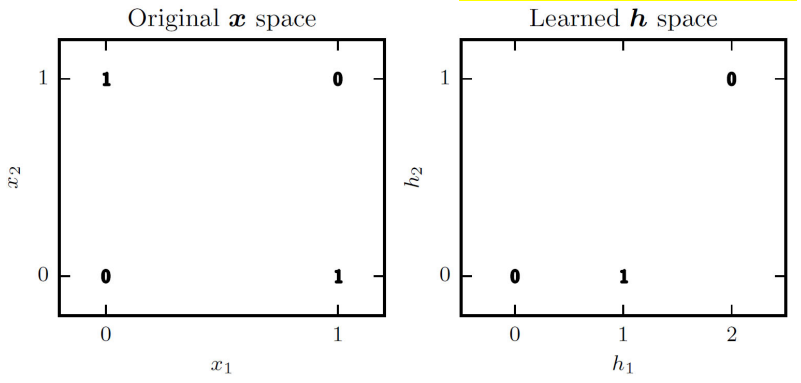
4. Multiply weight vector w

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Why MLP can do this?

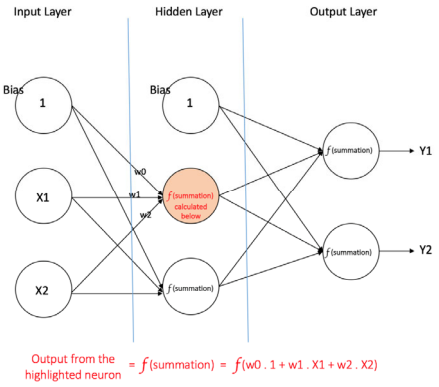
- An illustration

Created a new space, the hidden layer h space



Multi-Layer Perceptron (MLP)

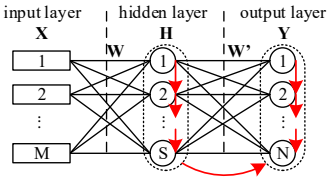
- A Multi-Layer Perceptron (MLP) contains one or more hidden layers, apart from one input and one output layer.
  - The input layer has no weights associated. It should not be considered as a network layer. For convenience, many people still call it input layer.



An example of 2-layer neural network

An MLP(M, S, N) in various forms

- Mathematical representation
- Graphical representation
- Software implementation



$$\vec{H} = f(W_{S\times M} \times \vec{X} + \vec{b})$$
$$\vec{Y} = f(W'_{N\times S} \times \vec{H} + \vec{b'})$$

```
hidden layer   for (int j = 0; j < S; j++)          //loop S1
                 for (int i = 0; i<M; i++)          //loop M
                   H[j] += W[j,i]*X[i];
output layer   for (int k = 0;k < N; k++)          //loop N
                 for(int j = 0; j < S; j++)         //loop S2
                   Y[k] += W'[k,j]*H[j];
```

Basic operations: Matrix-Vector multiplication
Batch processing: Matrix-Matrix multiplication

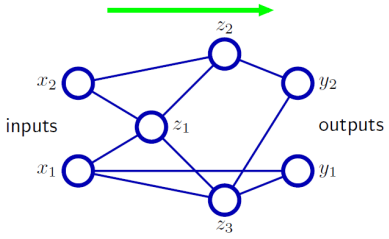
Sequential layer-after-layer processing

- Hardware implementation (Lab 3A) (Refer to the paper).

Shenggang Chen and Zhonghai Lu. "Hardware Acceleration of Multi-layer Perceptron Based on Inter-layer Optimization". International Conference on Computer Design (ICCD), November 2019.

Generalization of the feedforward network architecture

- Go deeper: Adding more hidden layers
- To include skip-layer connections

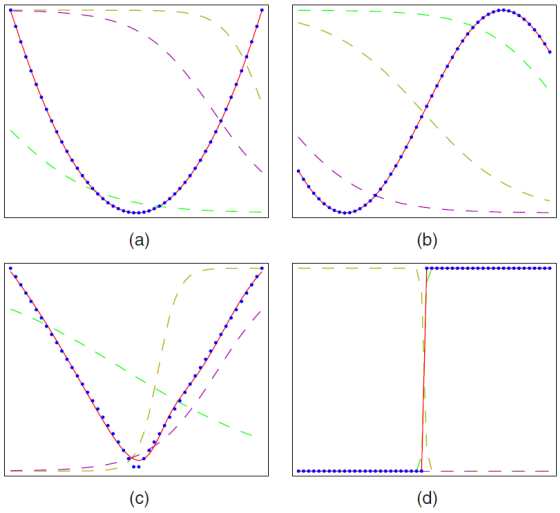


Example of a neural network having a general feed-forward topology.
Note that each hidden and output unit has an associated bias (omitted for clarity).

Capability of MLP

- Neural networks are said to be *universal function approximators*.
- For example, a two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided the network has a sufficiently large number of hidden units.

An illustrative example



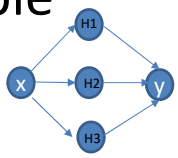


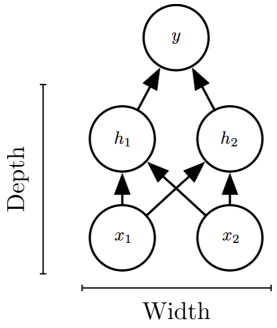
Figure 5.3 Illustration of the capability of a multilayer perceptron to approximate four different functions comprising (a) $f(x) = x^2$, (b) $f(x) = \sin(x)$, (c), $f(x) = |x|$, and (d) $f(x) = H(x)$ where $H(x)$ is the Heaviside step function. In each case, $N = 50$ data points, shown as blue dots, have been sampled uniformly in x over the interval $(-1, 1)$ and the corresponding values of $f(x)$ evaluated. These data points are then used to train a two-layer network having 3 hidden units with 'tanh' activation functions and linear output units. The resulting network functions are shown by the red curves, and the outputs of the three hidden units are shown by the three dashed curves.

Universal Approximator Theorem

- By the theorem, MLP with one hidden layer is enough to *represent* (*not learn*) an approximation of any function to an arbitrary degree of accuracy.
- Even if MLP is able to represent an arbitrary function, learning can fail for two reasons.
  - The optimization algorithm may fail to find the value of the parameters for the desired function.
  - The training algorithm might choose the wrong function as a result of overfitting.

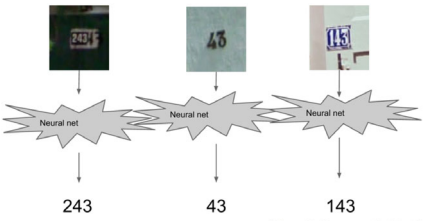
Universal Approximator Theorem

- If so, still why deeper?
  - Shallow net may need (exponentially) larger width
  - Shallow net may overfit more

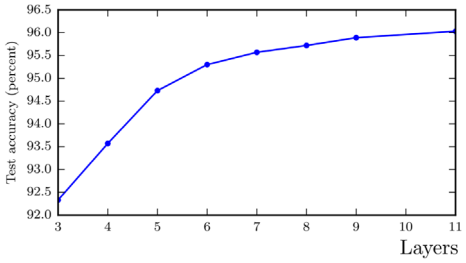


Opportunity by increasing depth

- Street view address number transcription



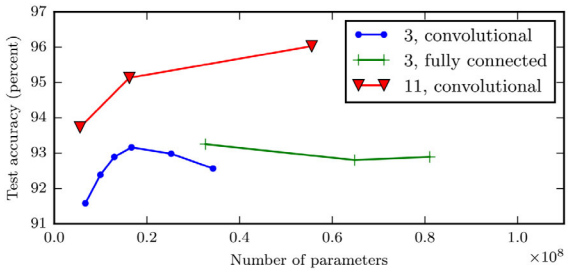
(Goodfellow et al, 2014)



- Effect of depth. Empirical results showing that deeper networks generalize better when used to transcribe multi-digit numbers from photographs of addresses.
- The test set accuracy consistently increases with increasing depth.
- Figure on the next page for a control experiment shows that other increases to the model size do not yield the same effect.

Large, shallow models overfit more

- Effect of number of parameters. Deeper models tend to perform better.



This is not merely because the model is larger.
- Increasing the number of parameters in layers of convolutional networks without increasing their depth is not nearly as effective at increasing test set performance.
- Shallow models overfit at around 20 million parameters while deep ones can benefit from having over 60 million.

This suggests that using a deep model expresses a useful preference over the space of functions the model can learn. Specifically, it expresses a belief that the function should consist of many simpler functions composed together.
- This could result either in learning a representation that is composed in terms of simpler representations (e.g., corners defined in terms of edges) or in learning a program with sequentially dependent steps (e.g., first locate a set of objects, then segment them from each other, then recognize them)

Selection of output neuron's activation function and error function

- There is a natural choice of both output unit activation function and matching error function, according to the type of problem being solved.
  - For regression, we use linear outputs (identity function) and a sum-of-squares error.
  - For binary classification, we can use a single logistic sigmoid output, or alternatively we can use a network with two outputs having a softmax output activation function.
  - For multiclass classification we use softmax outputs with the corresponding multiclass cross-entropy error function.

Problem	Output neuron activation function	Error (cost, loss) function
Regression	Identity function (linear outputs)	Sum of squares Residual* Sum of Squares
Binary classification (2 classes)	Single neuron: Logistic sigmoid Two neurons: Softmax	Binary cross-entropy
Multiclass classification*	Softmax	Multiclass cross-entropy

*The binary target variables $t_k \in \{0, 1\}$, $k \in [1, K]$ have a 1-of-K **one-hot coding** scheme indicating the class. E.g. if 4 classes, [0, 0, 0, 1], [1, 0, 0, 0] ….

The cross-entropy error function

- Consider the standard multiclass classification problem in which each input is assigned to one of *K* mutually exclusive classes.
- The binary target variables

$t_k \in \{0, 1\}$ have a 1-of-K coding scheme indicating the class, and the network outputs are interpreted as $y_k(\mathbf{x}, \mathbf{w}) = p(t_k = 1|\mathbf{x})$, leading to the following error function

$$E(\mathbf{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} t_{kn} \ln y_k(\mathbf{x}_n, \mathbf{w}).$$

- For binary classification with a single sigmoid output, the binary cross-entropy error function is

$$E(\mathbf{w}) = -\sum_{n=1}^{N} \{t_n \ln y_n + (1 - t_n) \ln(1 - y_n)\}$$

See details in Bishop's book Section 5.2. Network Training

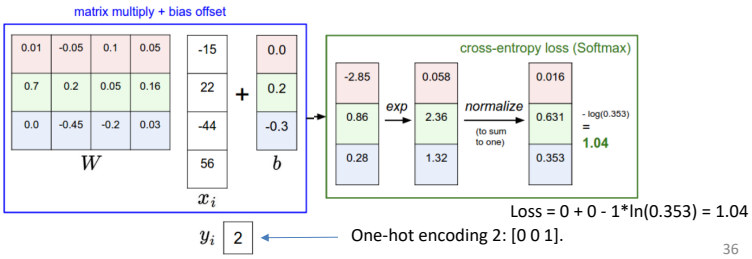
Softmax classification

- The softmax function is a generalization of the logistic function to multiple dimensions.

$$y_k(\mathbf{x}, \mathbf{w}) = \frac{\exp(a_k(\mathbf{x}, \mathbf{w}))}{\sum_j \exp(a_j(\mathbf{x}, \mathbf{w}))}$$

which satisfies $0 \leqslant y_k \leqslant 1$ and $\sum_k y_k = 1$.

- Cross-entropy error function



Loss = 0 + 0 - 1*ln(0.353) = 1.04

y_i 2 ← One-hot encoding 2: [0 0 1].

https://cs231n.github.io/linear-classify/#practical-considerations

The learning or training

- Given a set of features **X = (X_1, X_2, …)** and a target **Y**, an MLP can learn the relationship between the features and the target, for either classification or regression.
- The goal of learning is to assign correct weights for these edges. Given an input vector, these weights determine what the output vector is.
  - An ANN consists of nodes in different layers; input layer, intermediate hidden layer(s) and the output layer.
  - The connections between nodes of adjacent layers have "weights" associated with them.

An example

- **Known** (data set or labeled data)

Input vector		Output
Hours Studied	Mid Term Marks	Final Term Result
35	67	1 (Pass)
12	75	0 (Fail)
16	89	1 (Pass)
45	56	1 (Pass)
10	90	0 (Fail)

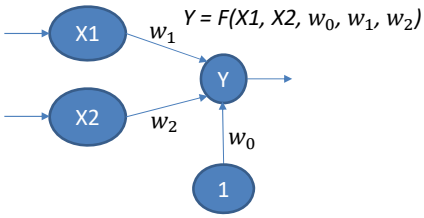
- **Predict** output for unseen input vector

Hours Studied	Mid Term Marks	Final Term Result
25	70	?

https://github.com/rasbt/python-machine-learning-book/blob/master/faq/visual-backpropagation.md

Build a classifier

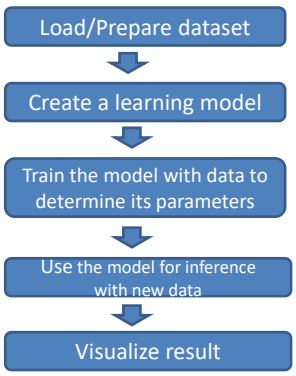
- Create a logistic regression model for the example
  - Two output-layer options: use one neuron, or two neurons

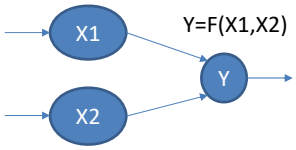


$$Y = Sigmoid(w_0 + w_1X_1 + w_2X_2)$$

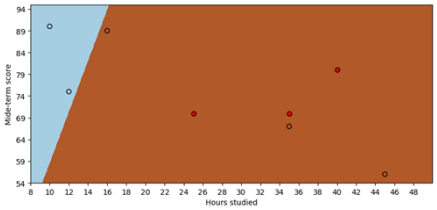
Build a classifier

- A typical machine learning & inference process





$$Y = S(w_0 + w_1X_1 + w_2X_2)$$



Decision boundary for the example

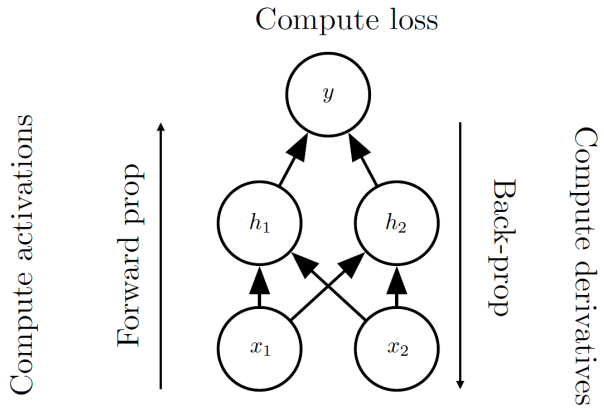
The training/learning process

- NN training is typically an **error-based supervised learning** -- "learning from mistakes".
- Initially all the edge weights of the ANN are randomly assigned.
  - **Forward Pass**: For every input in the training dataset, the ANN is activated and its output is generated.
  - **Backward Pass**: This output is compared with the desired output that we already know, and the error is "propagated" back to the previous layer.

    -- Error back propagation
  - **Weight tuning/update**: This error is noted and the weights are "adjusted" accordingly.

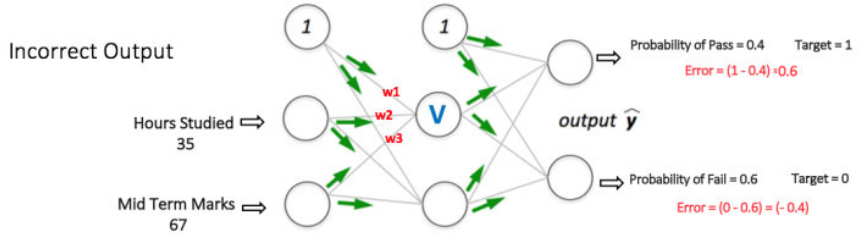
  This process is repeated until the termination condition is met, e.g., the output error is below a predetermined threshold or after a certain epochs.
- Once the training terminates, a "learned" ANN is ready to work with "new" inputs for **inference**.
  - This ANN has learned from several examples (labeled data) and from its mistakes (error back propagation).

Learning is iterative



Visual illustration

- Define an MLP for the problem
  - 2 neurons for the hidden layer
  - 2 neurons (one for Pass, one for Fail) for the output layer
- Train: Forward pass



https://github.com/rasbt/python-machine-learning-book/blob/master/faq/visual-backpropagation.md

Step 1 Forward pass

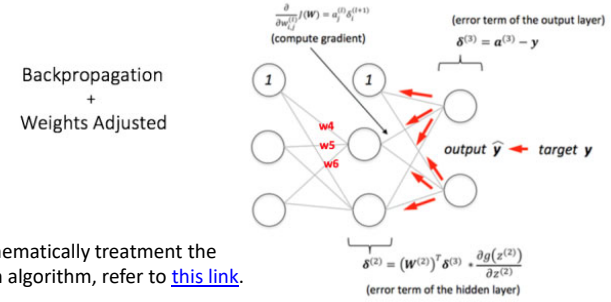
- The network then takes the first training example as input (we know that for inputs 35 and 67, the probability of Pass is 1).
  - Input to the network = [35, 67]
  - Desired output from the network (target) = [1, 0]
- Then output V from the node in consideration can be calculated as below (f is an activation function such as sigmoid):
  - V = f (1*w1 + 35*w2 + 67*w3)

Step 2 Backward pass

- Step 2: Back Propagation and Weight Update.

  (1) Calculate the total error at the output nodes and propagate these errors back through the network using backpropagation to calculate the gradients.

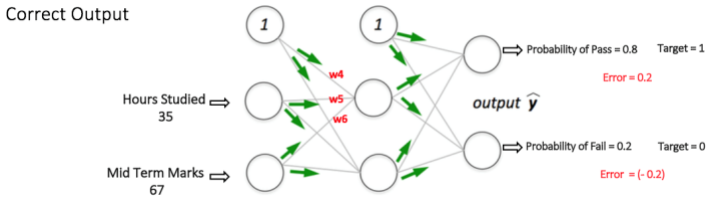
  (2) Then use an optimization method such as Gradient Descent to tune all weights in the network to reduce the error at the output layer.



For a more mathematically treatment the Backpropagation algorithm, refer to this link.

Effect and Iteration

- Effect of weight update



- Repeat this process with all other training examples in our dataset. Then, our network is said to have learnt those examples.

Inference or prediction

- **Inference**: If we now want to predict whether a student studying 25 hours and having 70 marks in the mid term will pass the final term, we go through the forward propagation step and find the output probabilities for Pass and Fail.
- Training/Learning is much more computational intensive than Inference.
  - Forward pass + Backward pass; Many iterations
  - Forward pass once

Learning strategy

- The combination of weights which minimizes the error function is considered to be a solution of the learning problem.

Learning strategy:

Minimize the error (E) as a function of all weights

Gradient descent:

$$w_i \leftarrow w_i - \eta \frac{\partial E}{\partial w_i}$$

Partial derivative of error with respect to Weight

$\eta > 0$ is known as the *learning rate*.

Gradient descent optimization

- Weight update: τ labels the iteration step. Different algorithms involve different choices for the weight vector update $\Delta \mathbf{w}^{(\tau)}$

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} + \Delta \mathbf{w}^{(\tau)}$$

- Gradient descent: choose the weight update to comprise a small step in the direction of the negative gradient of the error function.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

$\eta > 0$ is known as the *learning rate*.

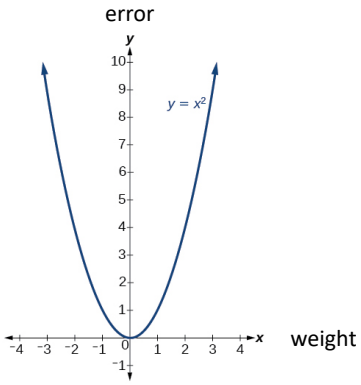
Key questions for weight update now

- Q1. Why can *gradient descent* based weight update minimize the error function?
- Q2. Can the weight update be done for every data sample in the labeled data set instead of for the entire data set?
- Q3. How to calculate the error gradient (with respect to weights) of all layers?

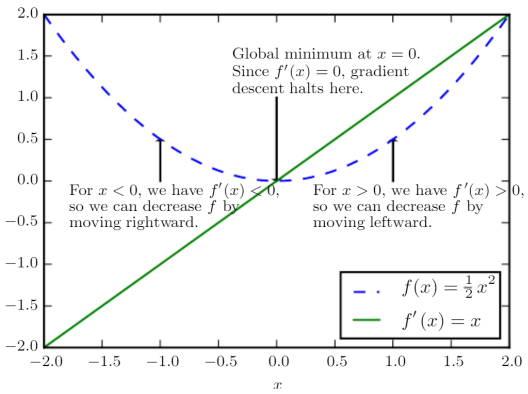
Weight update is when the learning happens!

Q1. Why gradient descent, not ascent?

- Use an example to explain.

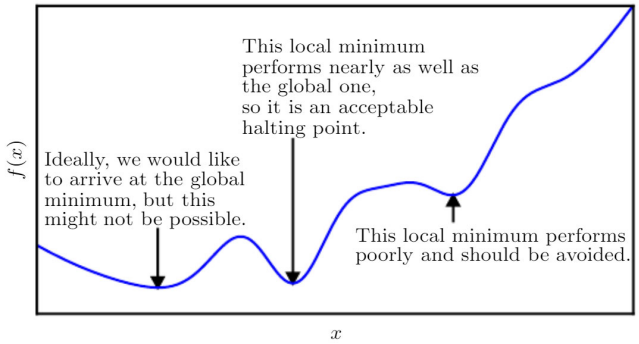


Gradient descent



An illustration of how the gradient descent algorithm uses the derivatives of a function to follow the function downhill to a minimum.

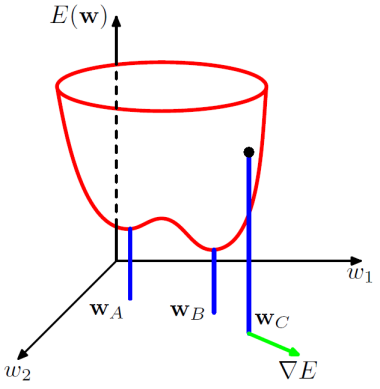
Approximate minimization



- Optimization algorithms may fail to find a global minimum when there are multiple local minima or plateaus present.
- In the context of deep learning, we generally accept such solutions even though they are not truly minimal.

Geometrical view of error function

Geometrical view of the error function $E(\mathbf{w})$ as a surface sitting over weight space. Point $\mathbf{w}_A$ is a local minimum and $\mathbf{w}_B$ is the global minimum. At any point $\mathbf{w}_C$, the local gradient of the error surface is given by the vector ∇E.



Q2. Batch-based gradient descent

- Gradient descent: choose the weight update to comprise a small step in the direction of the negative gradient.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

After each such update, the gradient is re-evaluated for the new weight vector and the process repeated.

- Note that the error function is defined with respect to **a training set**, and so each step requires that the entire training set be processed in order to evaluate ∇E

→ Needs batch processing to update weight. It is inefficient!

Stochastic gradient descent (SGD)

- Error functions based on maximum likelihood for a set of independent observations comprise a sum of terms, one for each data point, *n*.

$$E(\mathbf{w}) = \sum_{n=1}^{N} E_n(\mathbf{w})$$

- ***On-line*** gradient descent, known as ***sequential*** or ***stochastic*** gradient descent, updates the weight vector based on *one data point* at a time,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)})$$

This update is repeated by cycling through the data either **in sequence** or by **selecting points at random with replacement**.

- It handles redundancy in the data much more efficiently.

Batch vs. Stochastic gradient descent

- Gradient descent: choose the weight update to comprise a small step in the direction of the negative gradient.

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

- On-line/sequential/stochastic gradient descent updates the weight vector based on one data point at a time,

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)})$$

- It handles redundancy in the data much more efficiently.

An example: Take a data set and double its size by duplicating every data point.

- This simply multiplies the error function by a factor of 2 and so is equivalent to using the original error function.
- Batch methods will require double the computational effort to evaluate the batch error function gradient, whereas online methods will be unaffected.

Q3. Error backpropagation

- The propagation of errors backwards through the network in order to evaluate derivatives, can be applied to many other kinds of network and not just the multilayer perceptron.

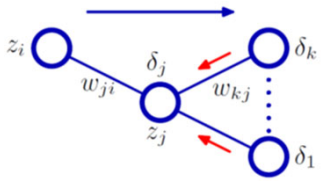


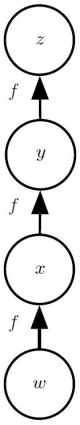
Illustration of the calculation of δ_j for hidden unit j by backpropagation of the δ's from those units k to which unit j sends connections. The blue arrow denotes the direction of information flow during forward propagation, and the red arrows indicate the backward propagation of error information.

- An application of the chain rule of derivatives

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \quad (5.55) \qquad a_j = \sum_i w_{ji} z_i \quad (5.48)$$

Here **activation *a*** denotes the weighted sum of its inputs. $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \quad (5.53)$

Chain rule of derivatives



$$\frac{\partial z}{\partial w} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}\frac{\partial x}{\partial w}$$
$$= f'(y)f'(x)f'(w)$$
$$= f'(f(f(w)))f'(f(w))f'(w)$$

Back-prop avoids computing this twice

Error backpropagation (BP) algorithm

Error Backpropagation

1. Apply an input vector $\mathbf{x}_n$ to the network and forward propagate through the network using (5.48) and (5.49) to find the activations of all the hidden and output units.
2. Evaluate the δ_k for all the output units using (5.54).
3. Backpropagate the δ's using (5.56) to obtain δ_j for each hidden unit in the network.
4. Use (5.53) to evaluate the required derivatives.

$$a_j = \sum_i w_{ji} z_i \quad (5.48) \qquad \frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \quad (5.53)$$

$$z_j = h(a_j) \quad (5.49) \qquad \delta_k = y_k - t_k \quad (5.54)$$

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \quad (5.56)$$

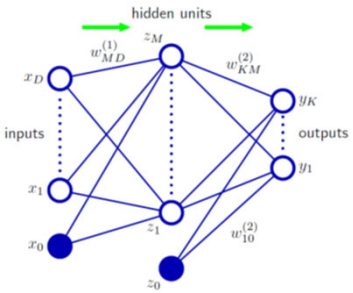
An MLP example

- Consider a two-layer network, with a sum-of-squares error, in which the output units have linear activation functions, so that $y_k = a_k$, while the hidden units have tanh() activation functions given by

$$h(a) \equiv \tanh(a) \quad \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}} \quad \Rightarrow \quad h'(a) = 1 - h(a)^2$$

- The error function is a standard sum-of-squares error function:

$$E_n = \frac{1}{2}\sum_{k=1}^{K}(y_k - t_k)^2 \qquad y_k = a_k = \sum_{j=0}^{M} w_{ki} z_j$$



which has the property (note here $y_k = a_k$)

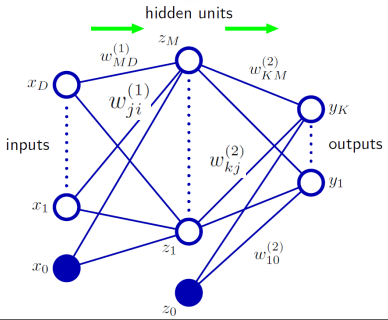
$$\frac{\partial E}{\partial a_k} = y_k - t_k \quad (5.18)$$

BP algorithm steps

- Step 1. For each pattern n in the training set, do a forward propagation using

$$\begin{aligned} a_j &= \sum_{i=0}^{D} w_{ji}^{(1)} x_i \\ z_j &= \tanh(a_j) \\ y_k &= \sum_{j=0}^{M} w_{kj}^{(2)} z_j \end{aligned}$$

- Step 2. Next we compute the δ's for each output unit using



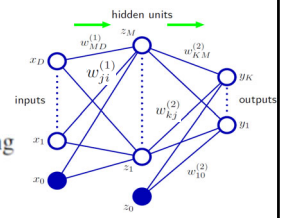
$$\delta_k = y_k - t_k. \qquad \delta_j \equiv \frac{\partial E_n}{\partial a_j}$$

$$\delta_k \equiv \frac{\partial E_n}{\partial a_k}$$

BP algorithm steps

- Step 1. For each pattern n in the training set, do a forward propagation using

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i$$
$$z_j = \tanh(a_j)$$
$$y_k = \sum_{j=0}^{M} w_{kj}^{(2)} z_j$$



- Step 2. Next we compute the δ's for each output unit using

$$\delta_k = y_k - t_k.$$

- Step 3. Then we backpropagate these to obtain δs for the hidden units using

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \qquad \delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k.$$

- Step 4. The derivatives of the first-layer and second-layer weights are given by

$$\frac{\partial E_n}{\partial w_{ji}^{(1)}} = \delta_j x_i, \qquad \frac{\partial E_n}{\partial w_{kj}^{(2)}} = \delta_k z_j$$

Back propagation and SGD

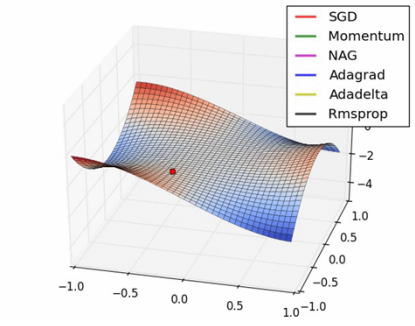
- Backpropagation
  - An efficient method of **computing gradients** in directed graphs of computations, such as neural networks.
  - It is a simple implementation of **chain rule of derivatives**, which allows to compute all partial derivatives in linear time in terms of the graph size (while naive gradient computations would scale exponentially with depth).
- Stochastic gradient descent (SGD)
  - It is an optimization method using the gradient information to update weights.
  - It is one of many known optimization techniques which use gradient information such as RMSProp (Root Mean Square Propagation) and Adam (Adaptive Moment Estimation) etc.

Optimization algorithms

- SGD
  - Can be slow
- SGD with moment
  - Introduce a variable ***v*** as velocity (direction and speed) of moving through parameter space
  - Nesterov moment (Nesterov's accelerated gradient, NAG)
- Algorithms with adaptive learning rates
  - RMSProp (Root Mean Square Propagation)
  - RMSProp with moment
  - Adam: **Ada**ptive **m**oments
- Others

Which optimization algorithm to choose?

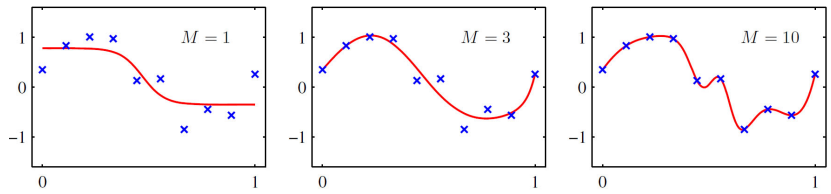
- Research show that, while the family of optimization algorithms with *adaptive learning rates* (represented by RMSprop and AdaDelta) performed fairly robust, no single best algorithm emerged.
- The choice of which optimization algorithm to use, at this point, seems to depend largely on the user's familiarity with the algorithm.



https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a

Overfitting

- The number of input and outputs units in a neural network is generally determined by the dimensionality of the data set, whereas the number M of hidden units is a free parameter that can be adjusted to give the best predictive performance.

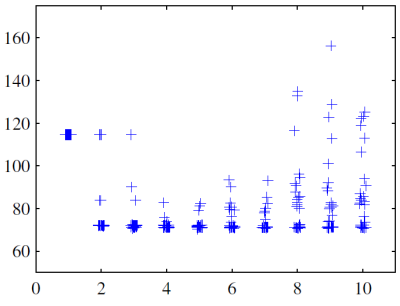


Examples of two-layer networks trained on 10 data points drawn from the sinusoidal data set. The graphs show the result of fitting networks having M = 1, 3 and 10 hidden units, respectively, by minimizing a sum-of-squares error function.

- Note that the generalization error is not a simple function of M. Given the same M, different weights can lead to different errors.

Local minima in the error function

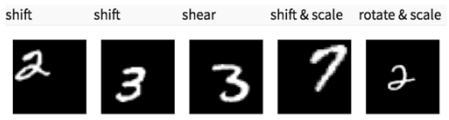
- The generalization error is not a simple function of M due to **the presence of local minima in the error function**.
- Here we see the effect of *choosing multiple random initializations* for the weight vector for a range of values of M.
- The overall best validation set performance in this case occurred for a particular solution having M = 8.
- In practice, one approach to choosing M is to plot a graph of the kind and then to choose the specific solution having the smallest validation set error.



- Plot of **the sum-of-squares test-set error** versus **the number of hidden units** for the polynomial (sinusoidal) data set , with 30 random starts for each network size, showing the effect of local minima.
- For each new start, the weight vector was initialized by sampling from an isotropic Gaussian distribution having a mean of zero and a variance of 10.

Fight overfitting

- Data augmentation
  - Not only add data points (including adding noises)
  - But also transform data (shift, scale, slight rotate etc.), particular for images.

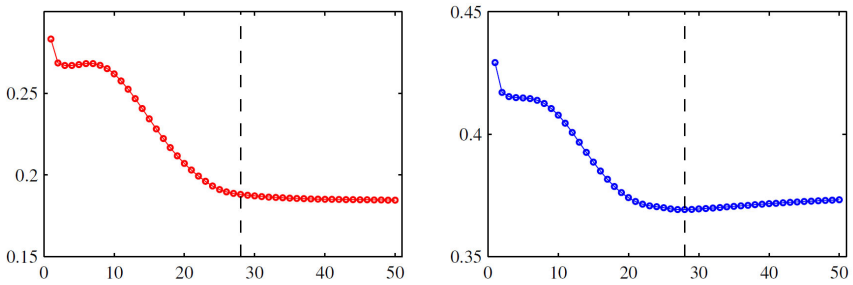


- Regularization via parameter norm penalty (E.g. L2 Weigh decay, L1 sparse weights)
- Early stopping
- Drop out
- Etc.

Early stopping

- An alternative to regularization as a way of controlling the effective complexity of a network is the procedure of *early stopping*.
- The training of nonlinear network models corresponds to an iterative reduction of the error function defined with respect to a set of training data.
  - For many of the optimization algorithms for network training, the error is a non-increasing function of the iteration index.
  - However, the error measured with respect to independent data, generally called a validation set, often shows a decrease at first, followed by an increase as the network starts to over-fit.

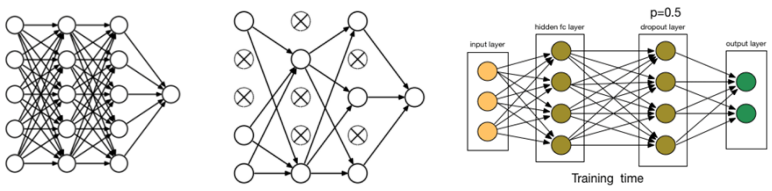
Illustration of early stopping



- An illustration of the behavior of training set error (left) and validation set error (right) during a typical training session, as a function of the iteration step, for the sinusoidal data set.
- The goal of achieving the best generalization performance suggests that training should be stopped at the point shown by the vertical dashed lines, corresponding to the minimum of the validation set error.

Drop out

- An ensemble learning technique
  - At every iteration, it randomly selects some nodes and removes them along with all of their incoming and outgoing connections.
  - So each iteration has a different set of nodes and this results in a different set of outputs.



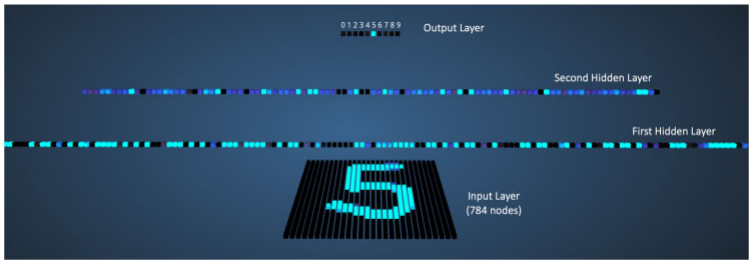
 https://www.analyticsvidhya.com/blog/2018/04/fundamentals-deep-learning-regularization-techniques/

Transfer learning

- Learning by random initialization of weights can result in uncertainty.
  - Final outcome is sensitive to the initially chosen weights.
  - Training can take too long or too many epochs to converge.
- Transfer learning means that the actual learning can use an already proven network layers with set of weights as the starting point for training the neural network.
  - Re-use proven network layers and set of weights
  - Transfer learning is very commonly used, for example, in the feature extraction process of CNNs.
  - In the first part of Lab 3B, you will exploit the idea of transfer learning for an image classification task.

3d Visualization of an MLP

- Adam Harley has created a 3d visualization of a MLP which has already been trained (using Backpropagation) on the MNIST Database of handwritten digits.
- The network takes 784 numeric pixel values as inputs from a 28 x 28 image of a handwritten digit (it has 784 nodes in the Input Layer corresponding to pixels). The network has 300 nodes in the first hidden layer, 100 nodes in the second hidden layer, and 10 nodes in the output layer (corresponding to the 10 digits)



- In Lab 1, you will design neural networks for Handwritten Digits Recognition.

Pros and cons of MLP

- Pros:
  - Capability to learn non-linear models.
  - Capability to learn models in real-time (on-line learning).
- Cons:
  - Training is slow.
  - MLP with hidden layers exists *more than one local minimum*.
    - Therefore different random weight initializations can lead to different validation accuracy.
    - Risk of getting stuck in local minima.
  - MLP requires tuning a number of *hyper-parameters* such as the number of hidden neurons, layers etc.
  - MLP is sensitive to *feature scaling*.

Step size η
Number of layers
Number of hidden units
Initial weights
Activation functions

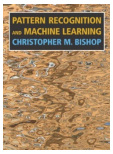
Summary

- ANN uses perceptron as the building block. It may be viewed as a structured organization and connections of perceptrons.
  - Strictly speaking, they are not perceptrons, because of different nonlinear functions rather than the step function is used.
- A typical learning & inference process includes Dataset preparation, Model definition (hyper parameters), Parameter training, Inference, Result visualization.
- The purpose of NN learning is to determine the parameters for the selected model, e.g., the MLP weights.
  - Typically error-based supervised learning
  - Error Back-propagation algorithm
  - Stochastic gradient descent and other optimization algorithms
  - Common measures to fight overfitting
  - Learning does not have to start from scratch. Transfer learning is often a first try.

Home work

- Review slides and answer the following questions.
  - Explain the basic functionality of artificial neuron.
  - Explain if there is any relationship between linear/logistic regression and artificial neuron?
  - Why is training much more computation intensive than inference?
  - Explain intuitively why the perceptron cannot solve nonlinear classification problems?
  - Explain how MLP is constructed? How weights are initialized and trained? Why is designing MLP complicated?
  - Use an example to explain why gradient descent rather than ascend based weight update is the right direction towards minimizing the error function?
  - How to calculate the influence of weight change on the learning error? (Hint: the error back propagation process)

Major References



- Bishop's PRML book, 2006.
- R. Rojas. "Neural Networks: A Systematic Introduction". Springer-Verlag, Berlin, 1996
- Ian Goodfellow and Yoshua Bengio and Aaron Courville, "Deep Learning". MIT press, 2016.
- A neural network playground

  https://playground.tensorflow.org