Data Science and Advanced Programming — Lecture 10

Introduction to Deep Learning

Simon Scheidegger
Department of Economics, University of Lausanne, Switzerland

November 17th, 2025 | 12:30 - 16:00 | Internef 263

Today's Roadmap

This lecture

- A brief recap on Machine Learning Basics
- Deep Learning Basics
- White-box examples:
- The multi-layer perceptron
- Feed-forward networks
- Network training - SGD
- Error back-propagation
- Some notes on over-fitting

Throughout lectures - hands-on:

- Perceptron
- Gradient descent
- Artificial neural networks: a simple multi-layer perceptron implementation & several examples

The Rise of Neural Networks

TOM SIMONITE BUSINESS 01.25.19 01:35 PM

DEEPMIND BEATS PROS AT STARCRAFT IN ANOTHER TRIUMPH FOR BOTS

A pro gamer and an AI bot duke it out in the strategy game StarCraft, which has become a test bed for research on artificial intelligence. STARCRAFT

IN LONDON LAST month, a team from Alphabet's UK-ba artificial intelligence research unit DeepMind quietly l new marker in the contest between humans and compu On Thursday it revealed the achievement in a three-ho YouTube stream, in which aliens and robots fought to t death.

TOM SIMONITE BUSINESS 10.18.17 01:00 PM

THIS MORE POWERFUL VERSION OF ALPHAGO LEARNS ON ITS OWN

NOAH SHELDON FOR WIRED

AT ONE POINT during his historic defeat to AlphaGo last year, world champion Go play abruptly left the room. The bot had played confounded established theories of the boa moment that came to epitomize the myste AlphaGo.

NEWS BIOLOGY 21 DECEMBER 2017

AI beats docs in cancer spotting

A new study provides a fresh example of machine learning as an important diagnostic tool. Paul Biegler reports.

Deep Learning Software Speeds Up Drug Discovery

Wed, 01/16/2019 - 8:00am 1 Comment by Kenny Walter , Science Reporter - @RandDMagazine

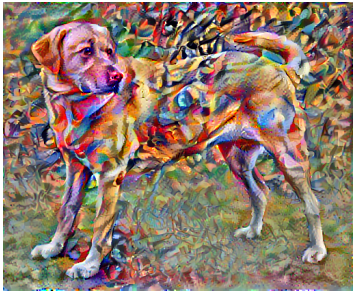
The long, arduous process of narrowing down millions of chemical compounds to just a select few that can be further developed into mature drugs, may soon be shortened, thanks to new artificial intelligence (AI) software.

Music generated by AI

https://openai.com/research/musenet

Style Transfer

https://www.tensorflow.org/tutorials/generative/style_transfer

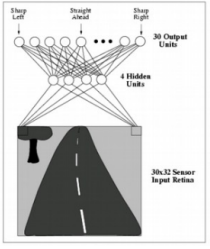


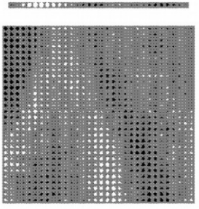
Self-Driving Cars

- Carnegie Mellon University — 1990's: Self Driving Cars S1E2: ALVINN.
- Google — 2017: Waymo.

ALVINN

[Pomerleau 1993]



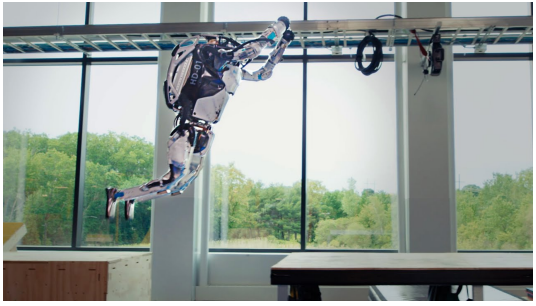


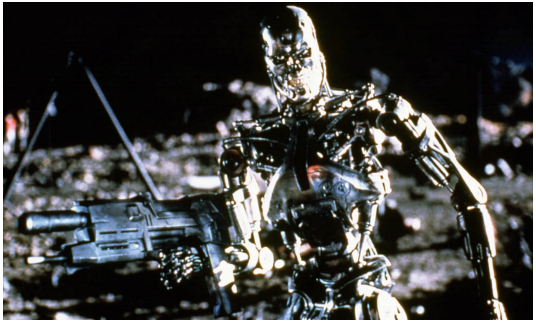
Coloring Old Movies

AI movie restoration — Scarlett O'Hara HD

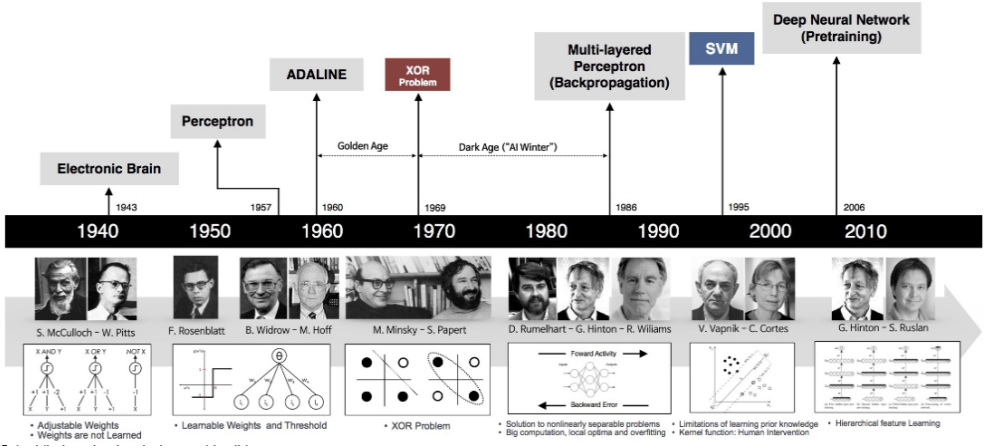
Two-Legged Robots

Boston Dynamics' Atlas Robot Can Do Parkour.





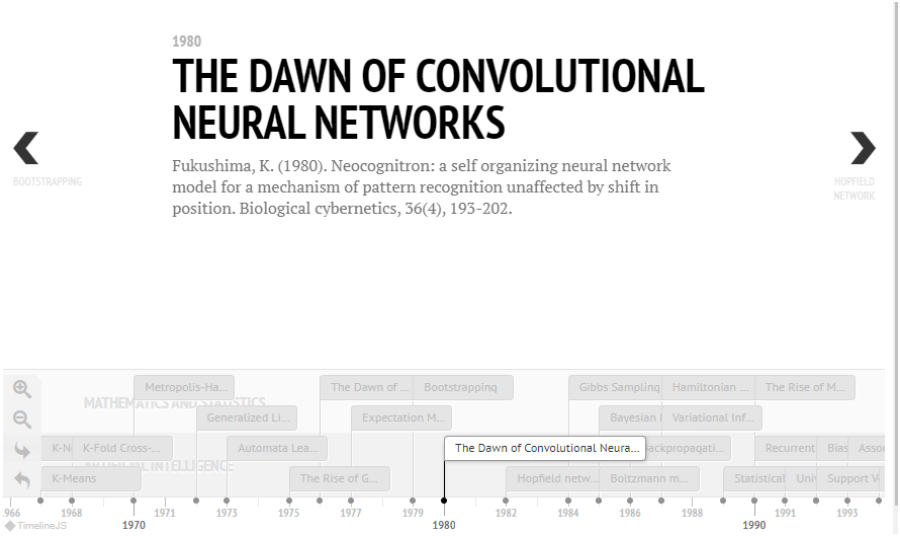
A Timeline of Deep Learning



*J. Schmidhuber clearly missing on this slide

Another Timeline

https://leandromineti.github.io/ML-timeline/



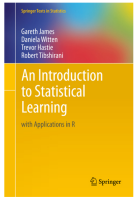
Why Now?

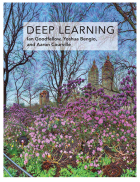
- Neural Networks date back decades, so why the resurgence? (Stochastic Gradient Descent: 1952, Perceptron: 1958, Back-propagation: 1986, Deep Convolutional NN: 1995)
  - Big Data
    - Large Datasets
    - Easier Collection and Storage
  - Hardware
    - GPUs, TPUs,...
  - Software
    - Improved Techniques
    - Toolboxes

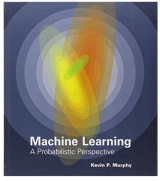
Some Useful Materials

Some useful textbooks:

- ***Machine Learning: a Probabilistic Perspective***
  K. Murphy, MIT Press, 2012. https://www.cs.ubc.ca/~murphyk/MLbook/index.html
- ***An Introduction to Statistical Learning***
  Gareth James, Daniela Witten, Trevor Hastie and Robert Tibshirani; Springer, 8th editon, 2017.
  https://www-bcf.usc.edu/~gareth/ISL/
- ***Deep Learning***
  Ian Goodfellow and Yoshua Bengio and Aaron Courville; MIT Press 2016.
  http://www.deeplearningbook.org







Recap on Machine Learning

Data Scientist (n.):

Person who is better at statistics than any software engineer and better at software engineering than any statistician.

David Donoho (2015). 50 years of Data Science

Data Science

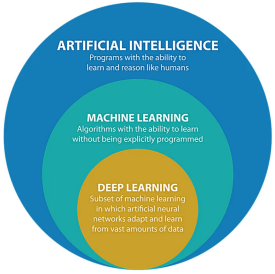
David Donoho (2015). 50 years of Data Science

The activities of Greater Data Science are classified into 6 divisions:

1. Data Exploration and Preparation
2. Data Representation and Transformation
3. Computing with Data
4. Data Modeling
5. Data Visualization and Presentation
6. *Science about Data Science

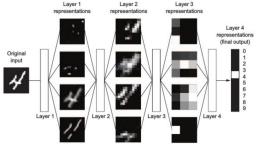
Set some terminology straight

Artificial intelligence (AI)
Can computers be made to "think"? — a question whose ramifications we're still exploring today. A concise definition of the field would be as follows: The effort to automate intellectual tasks normally performed by humans.



Machine Learning (ML)

Deep Learning as a particular example of an ML technique —



Types of Machine Learning

- **Supervised Learning**.
  Assume that training data is available from which they can learn to predict a target feature based on other features (e.g., monthly rent based on area).
  - **Classification**.
  - **Regression**.
- **Unsupervised Learning**
  Take a given dataset and aim at gaining insights by identifying patterns, e.g., by grouping similar data points.
- **Reinforcement Learning**.

Supervised Regression

- Regression aims at predicting a numerical target feature based on one or multiple other (numerical) features.
- Example: Price of a used car.
  - x : car attributes
  - y : price
  - $y = h(x|\theta)$
  - $h()$: model
  - θ : parameters

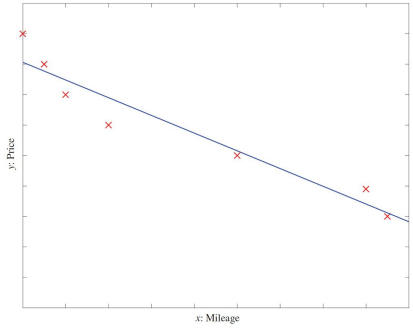


Fig. from Alpaydin (2014)

Supervised Classification

Example 1: Spam Classification

- Decide which emails are Spam and which are not.
- Goal: Use emails seen so far to produce a good prediction rule for **future** data.

Example 2: Credit Scoring

- Differentiating between low-risk and high-risk customers from their income and savings.
- Discriminant: `IF income >` θ_2 `AND savings >` θ_2 `THEN low-risk ELSE high-risk`

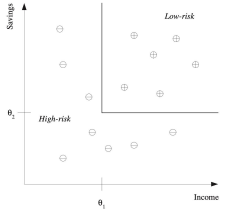
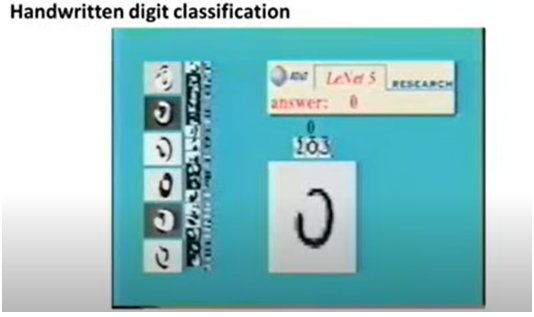


Fig. from Alpaydin (2014)

Handwritten Digit Classification

Movie from the early 90's. We have come a long way since then...

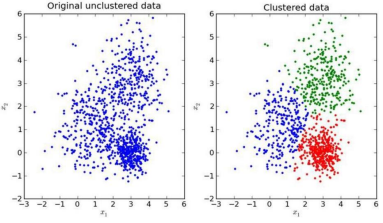
Handwritten Digit Classification – by Yann Lecun



Unsupervised ML

Learning “what normally happens”.

- No output.
- Clustering: Grouping similar instances.
- Example applications:
  - Customer segmentation.
  - Image compression: Color quantization.
  - Bioinformatics: Learning motifs.



Reinforcement Learning

- Learning a policy: A sequence of outputs.
- No supervised output but delayed reward.
- Credit assignment problem.
- Game playing.
- Robot in a maze.
- Multiple agents, partial observability, ...

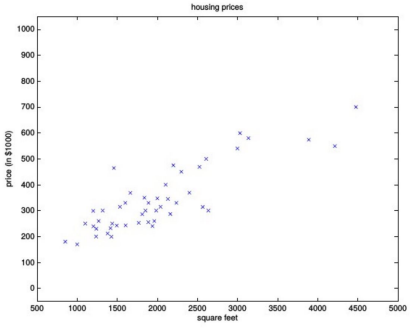
→ DeepMind's Q learning.

Building An ML Algorithm

- Optimize a performance criterion using example data or past experience.
- Role of Statistics: Inference from a sample.
- Role of computer science: Efficient algorithms to
  - Solve the optimization problem.
  - Representing and evaluating the model for inference.

Building An ML Algorithm (II)

Living area (feet ^{2})	Price (1000$s)
2104	400
1600	330
2400	369
1416	232
3000	540
⋮	⋮

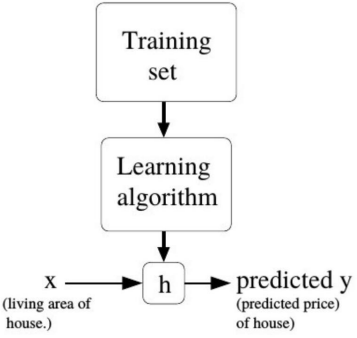


Given data like this, how can we learn to predict the prices of other houses as a function of the size of their living areas?

Building An ML Algorithm (III)

- $x(i)$: **"input" variables** (living area in this example), also called **input features**.
- $y(i)$: **"output" / target variable** that we are trying to predict (price).
- **Training example**: a pair $(x(i), y(i))$.
- **Training set**: a list of m training examples $(x(i), y(i)); i = 1, ..., m$.

To perform supervised learning, we must decide how we're going to represent **functions/hypotheses** h in a computer.



Building An ML Algorithm (IV)

- **Model/Hypothesis:**

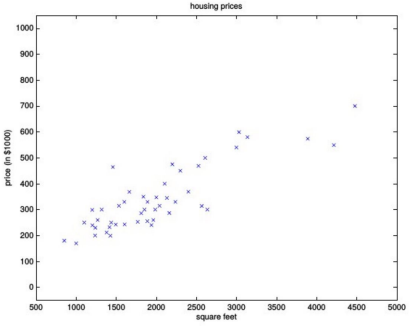
$$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2$$

θ_i's: parameters

- **Cost Function:**

$$J(\theta) = \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

$\implies$ Minimize $J(\theta)$ in order to obtain the coefficients θ.



Building An ML Algorithm (V)

In general, Machine Learning in 3 Steps:

- Choose a model $h(x \mid \theta)$.
- Define a cost function $J(\theta \mid x)$.
- Optimization procedure to find $\theta*$ that minimizes $J(\theta)$.

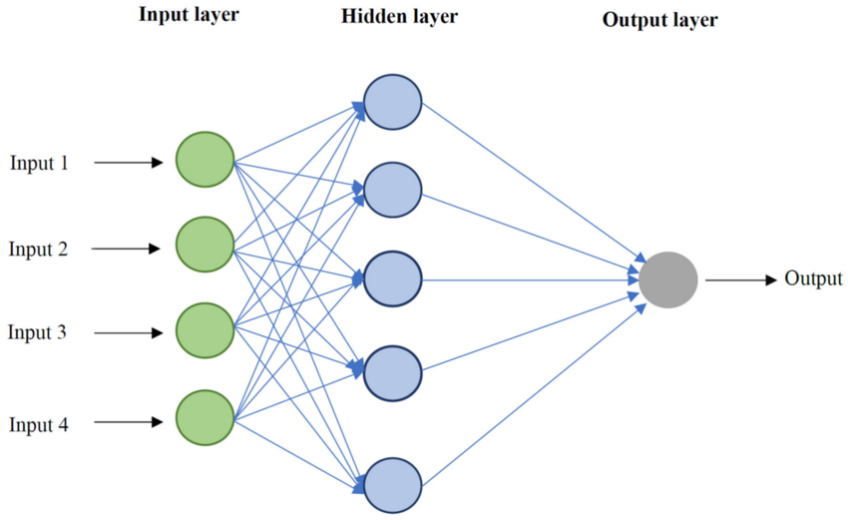
Computationally, we need data, linear algebra, statistics tools, and optimization routines.

Don't Re-Invent The Wheel

Plenty of Frameworks out there

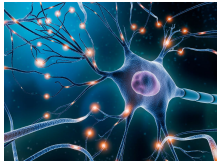
- Tensorflow
- Pytorch
- Caffee
- Scikit-learn
- ...

Artificial Neural Networks



The Brain and The Neuron

- **Biological systems** built of very complex webs of interconnected neurons.
- Highly connected to other neurons, and perform (parallel) computations by combining signals from other neurons.
- Outputs of these computations may be transmitted to one or more other neurons.
- **Artificial Neural Networks** (ANN) built out of a densely interconnected set of simple units (e.g sigmoid units).
- Each unit takes real-valued inputs (possibly the outputs of other units) and produces a realvalued output (which may become input to many other units).



Connectionist Model

- Consider humans
  - Neuron switching time ~ 0.001 second
  - Number of neurons $\sim 10^{10}$
  - Connections per neuron $\sim 10^{4.5}$
  - Scene recognition time ~ 0.1 second
- 100 inference steps doesn't seem like enough $\rightarrow$ a lot of parallel computation
- Properties of artificial neural nets (ANN's):
  - Many neuron-like threshold switching units
  - Many weighted interconnections among units
  - Highly parallel, distributed process

Hebb's Rule

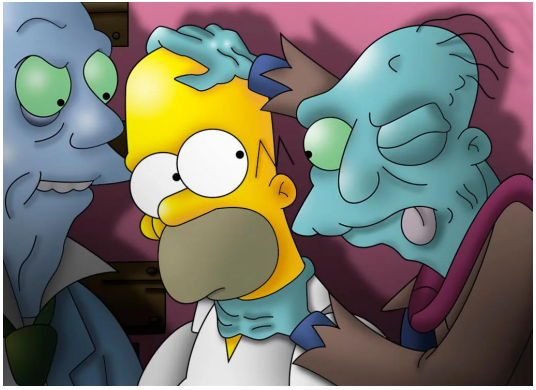
- Hebb's rule says that the changes in the strength of synaptic connections are proportional to the correlation in the firing of the two connecting neurons.
- So if **two neurons consistently fire simultaneously**, then any connection between them will change in strength, becoming stronger.
- However, if the two neurons never fire simultaneously, the connection between them will die away.
- **The idea is that if two neurons both respond to something, then they should be connected.**

Hebb's Rule — Intuition

- Suppose that you have a neuron somewhere that recognizes your grandmother (this will probably get input from lots of visual processing neurons, but don't worry about that).
- Now if your grandmother always gives you a chocolate bar when she comes to visit, then some neurons, which are happy because you like the taste of chocolate, will also be stimulated.
- Since these neurons fire at the same time, they will be connected together, and the connection will get stronger over time.
- **So eventually, the sight of your grandmother, even in a photo, will be enough to make you think of chocolate**. Sound familiar?

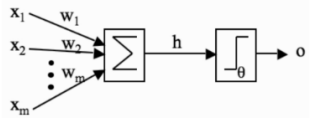
Artificial Neural Networks

- Artificial Neural networks arise from attempts to model human/animal brains
  - Many models, many claims of biological plausibility.
- We will focus on multi-layer perceptron
  - Mathematical properties rather than plausibility.



Model of A Neuron (1943)

- A picture of McCulloch and Pitts' (1943) mathematical model of a neuron.
  1. a set of weighted inputs w_i that correspond to the synapses.
  2. an adder that sums the input signals (equivalent to the membrane of the cell that collects electrical charge).
  3. an activation function (initially a threshold function) that decides whether the neuron fires ('spikes') for the current inputs.
- The inputs x_i are multiplied by the weights w_i, and the neurons sum their values.
- If this sum is greater than the threshold θ then the neuron fires; otherwise it does not.



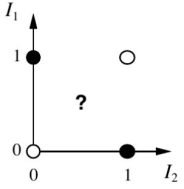
Feed-Forward Networks

In feed-forward networks (a.k.a. multi-layer perceptrons) we let each basis function be another non-linear function of linear combination of the inputs:

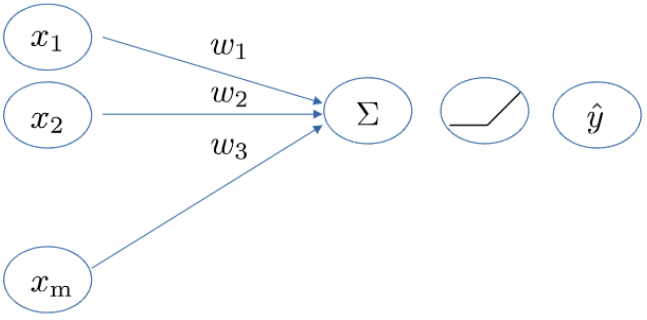
$$\phi_j(x) = f\left(\sum_{j=1}^{M} \cdots\right)$$

Limitations of Perceptrons

- Perceptrons can only solve linearly separable problems in feature space.
- A canonical example of non-separable problem is X-OR.
- Real data sets can look like this too.



The Perceptron: Forward Propagation

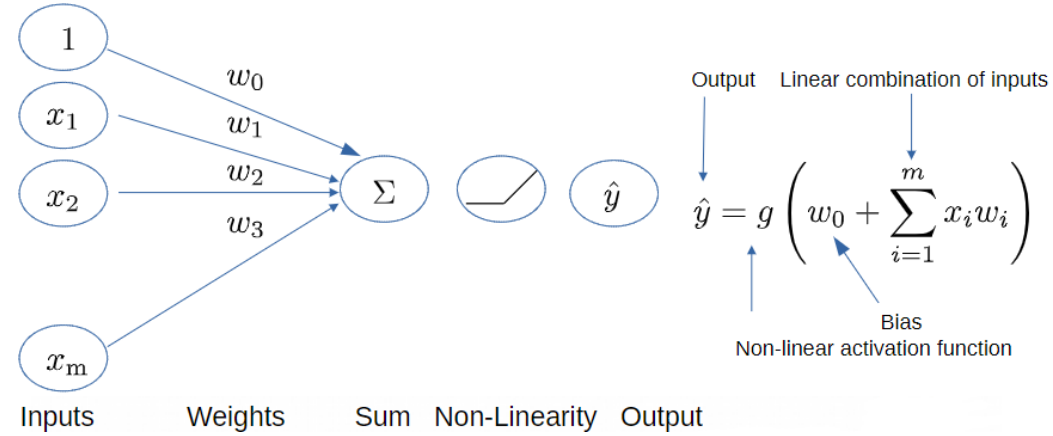


Output
Linear combination of inputs

$$\hat{y} = g\left(\sum_{i=1}^{m} x_i w_i\right)$$

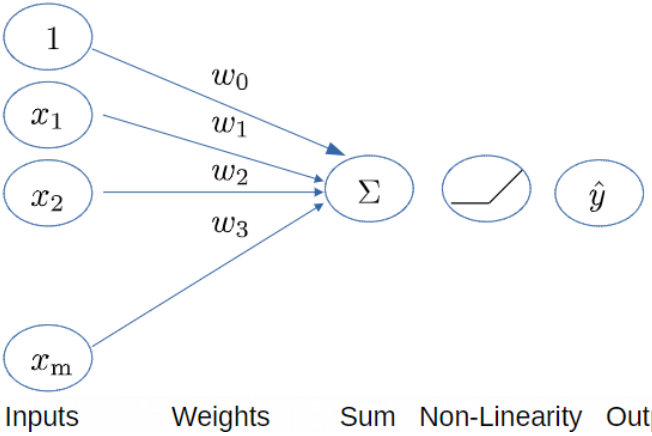
Non-linear activation function

The Perceptron: Forward Propagation



Bias term allows you to shift your activation function to the left or the right

The Perceptron: Forward Propagation



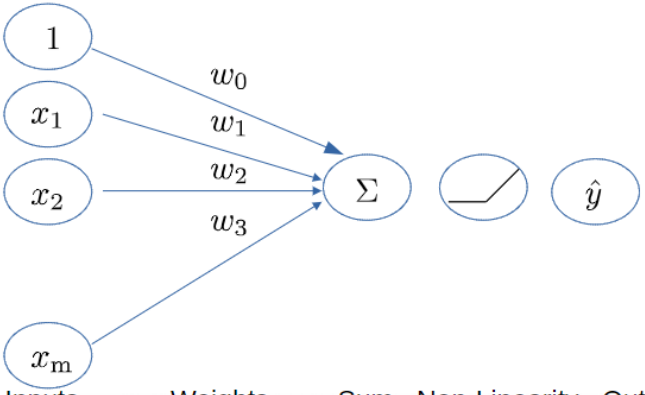
$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

$$\begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix} \text{ and } \boldsymbol{W} = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix}$$

Bias term allows you to shift your activation function to the left or the right

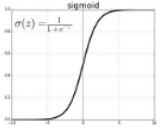
The Perceptron: Forward Propagation



$$\hat{y} = g\left(w_0 + \sum_{i=1}^{m} x_i w_i\right)$$

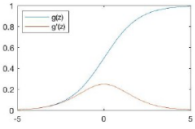
Activation Functions
e.g. sigmoid function

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



A Few Activation Functions

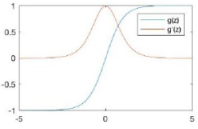
Sigmoid Function



$$g(z) = \frac{1}{1 + e^{-z}}$$

$$g'(z) = g(z)(1 - g(z))$$

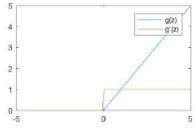
Hyperbolic Tangent



$$g(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

$$g'(z) = 1 - g(z)^2$$

Rectified Linear Unit (ReLU)



$$g(z) = \max(0, z)$$

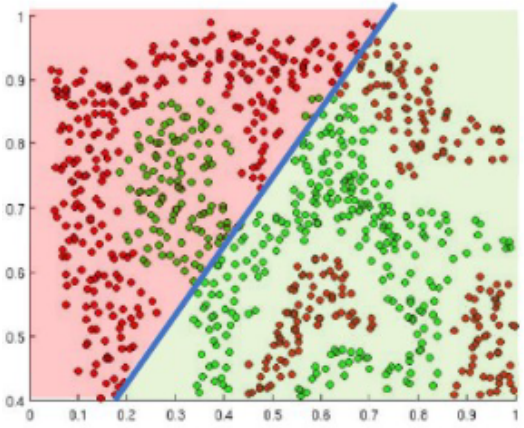
$$g'(z) = \begin{cases} 1, & z > 0 \\ 0, & \text{otherwise} \end{cases}$$

- Needs to be differentiable for gradient-based learning (later)
- Very useful in practice.
- Sigmoid function, e.g., useful for classification (Probability).

Importance of Activation Function

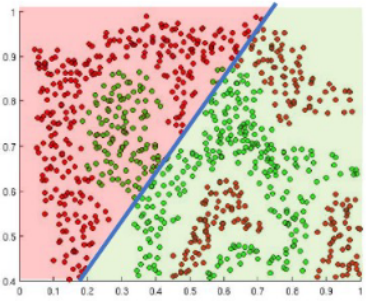
http://openclassroom.stanford.edu/

- The purpose of activation functions is to introduce non-linearities into the network.
- anted to build a Neural Network to distinguish green versus red points?

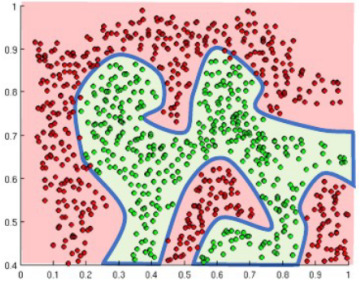


Importance of Activation Function

The purpose of activation functions is to introduce non-linearities into the network.

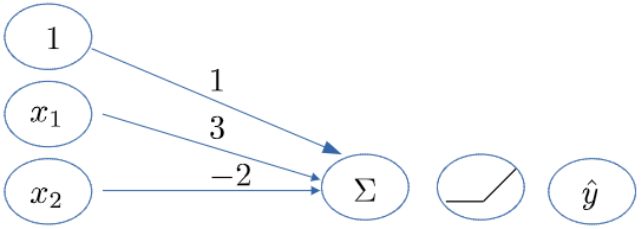


- Linear activation functions produce linear decisions no matter the network size.



- Non-linearities allow us to approximate arbitrarily complex functions.

Perceptron — An Example



We have: $w_0 = 1$ and $W = \begin{bmatrix} 3 \\ -2 \end{bmatrix}$

$$\hat{y} = g\left(w_0 + \boldsymbol{X}^T \boldsymbol{W}\right)$$

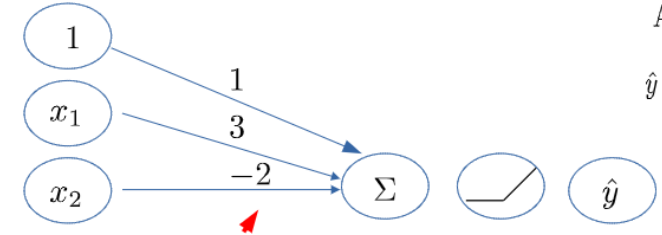
$$= g\left(1 + \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}^T \begin{bmatrix} 3 \\ -2 \end{bmatrix}\right)$$

$$\hat{y} = g\left(1 + 3x_1 - 2x_2\right)$$

This is just a line in $2D$

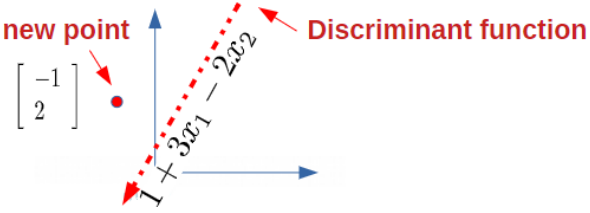
Imagine we have a trained network with weights given. How do we compute the output?

Perceptron — An Example

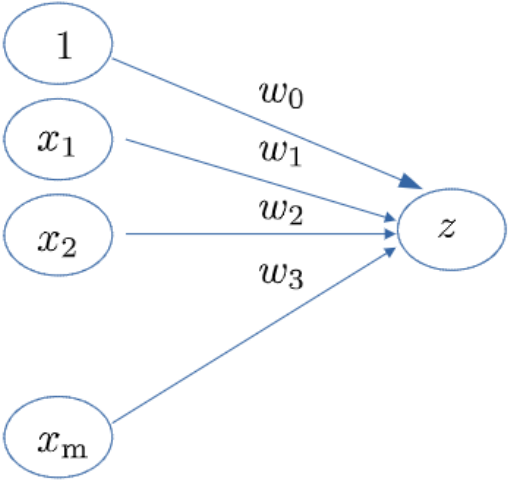


Assume we have input: $X = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$

$$\hat{y} = g(1 + (3 * -1) - (2 * 2))$$
$$= g(-6) \approx 0.002$$

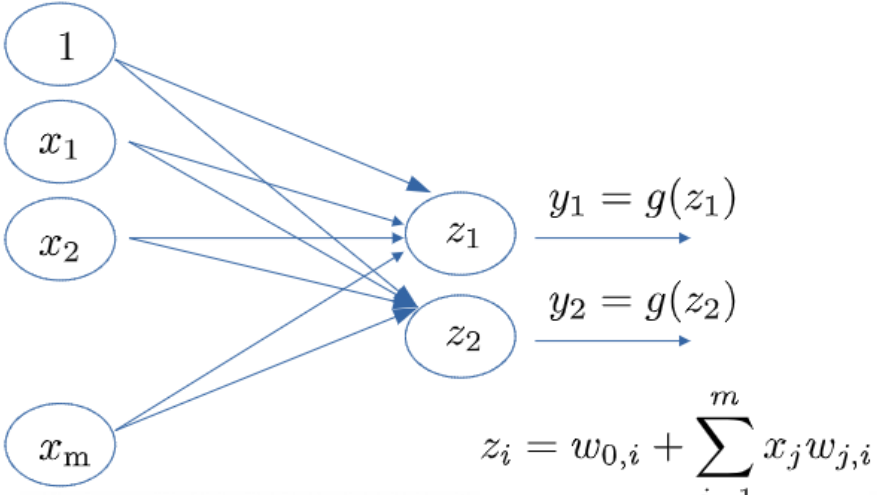


A Perceptron — Simplified

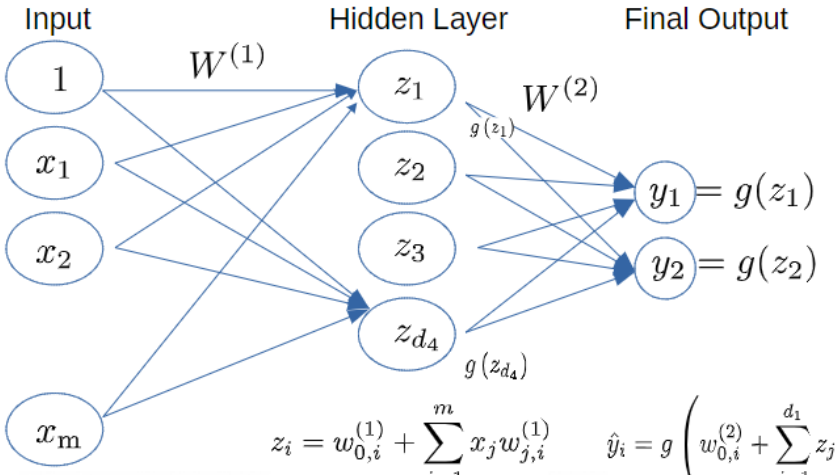


$$z = w_0 + \sum_{j=1}^{m} x_j w_j$$

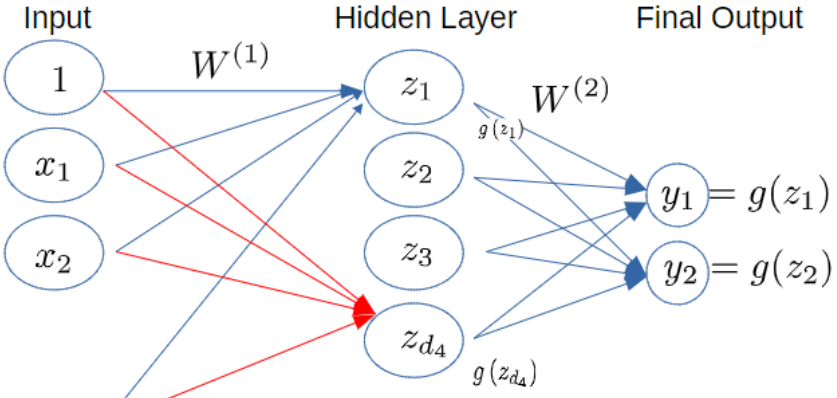
Building A NN With Perceptrons: A Multi-Output Perceptron



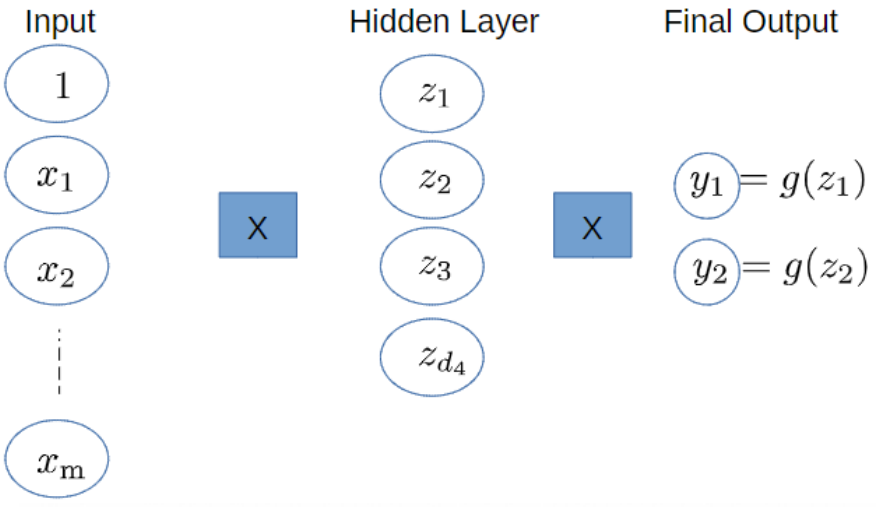
Single Layer NN



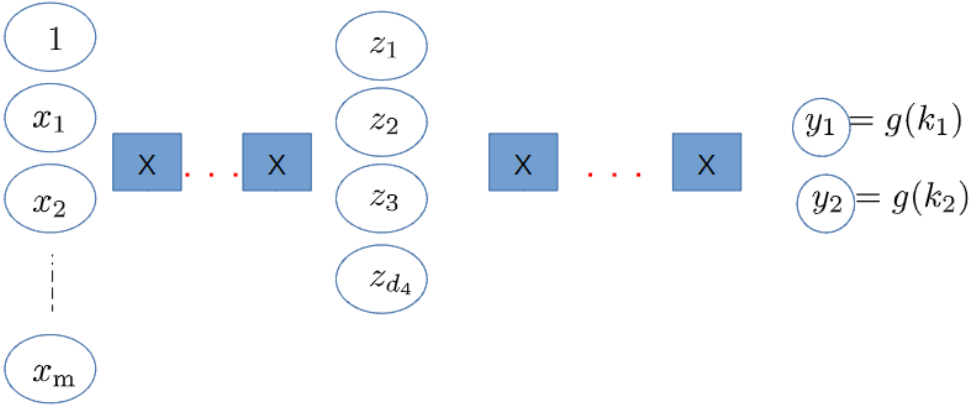
Single Layer NN



Single Layer NN



Fully Connected DNN



Expressiveness of ANN

- **Boolean functions:**
  - Every Boolean function can be represented by a network with a single hidden layer.
  - Might require exponential (in number of inputs) hidden units

.

- **Continuous functions:**
  - Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989].
- Deep NN are in practice superior to other ML methods in presence of large data sets.

Universal Function Approximator

Figs. from Marsland (2014)

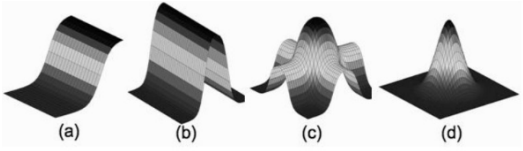


FIGURE 4.9 The learning of the MLP can be shown as the output of a single sigmoidal neuron (a), which can be added to others, including reversed ones, to get a hill shape (b). Adding another hill at 90° produces a bump (c) , which can be sharpened to any extent we want (d), with the bumps added together in the output layer. Thus the MLP learns a local representation of individual inputs.

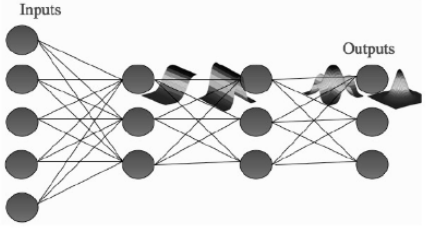
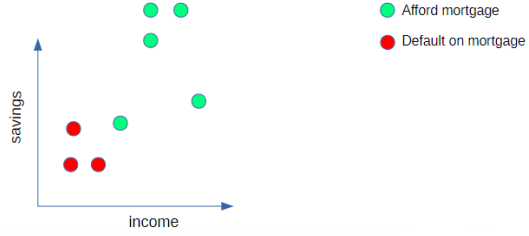


FIGURE 4.10 Schematic of the effective learning shape at each stage of the MLP.

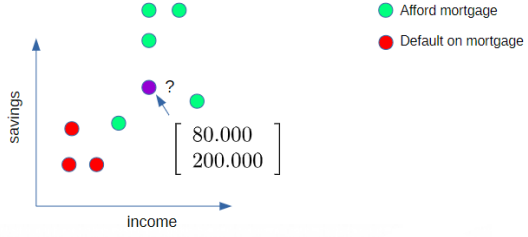
Classification Problem

- Can I afford a loan for a house?
- X: income
- Y: savings

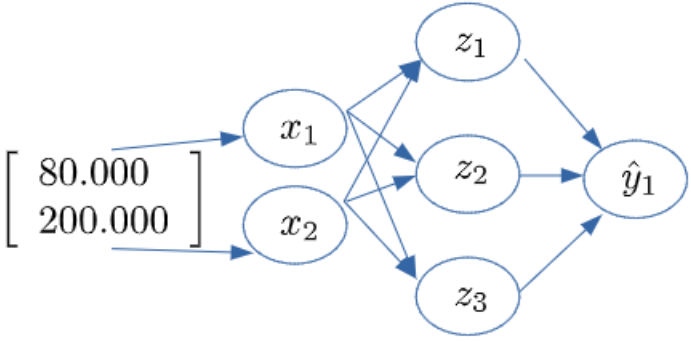


Classification Problem

- Can I afford a loan for a house?
- X: income
- Y: savings

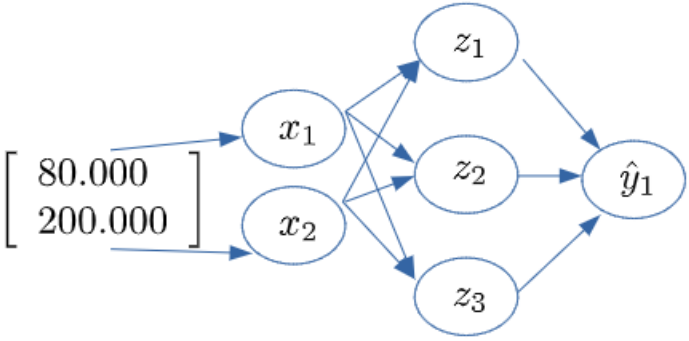


Trying to Do a Prediction



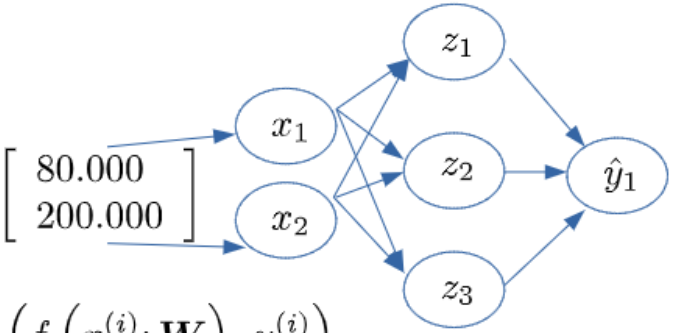
Prediction: 0.2

Trying to Do a Prediction



Prediction: 0.2
Actual: 1

Recall: Quantifying The "Loss"

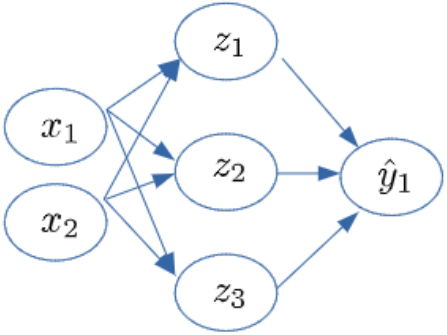


Prediction: 0.2
Actual: 1

Empirical Loss

The empirical loss measures the total loss over our entire data set.

$$\begin{bmatrix} 80.000, 200.000 \\ 120.000, 400.000 \\ 10.000, 12.000 \\ \cdots, \cdots \end{bmatrix}$$



Prediction: 0.2
Actual: 1

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \mathcal{J}\left(f\left(x^{(i)}; \boldsymbol{W} \right), y^{(i)} \right)$$

Predicted Actual

Cross Entropy Loss Function

- Classification: maximum likelihood principle.
- Consider a set of m examples $X = \left\{x^{(1)}, \ldots, x^{(m)}\right\}$ drawn independently from the true but unknown data-generating distribution $p_{\text{data}}\left(x; \theta\right)$.
- Let $p_{\text{model}}\left(x; \theta\right)$ be a parametric family of probability distributions over the same space indexed by θ. In other words, $p_{\text{model}}\left(x; \theta\right)$ maps any configuration x to a real number estimating the true probability $p_{\text{data}}\left(x; \theta\right)$.
- Maximum likelihood estimator for the parameters is given by

$$\begin{aligned}\boldsymbol{\theta}_{ML} &= \arg\max_{\boldsymbol{\theta}} p_{\text{model}}\left(\mathbb{X}; \boldsymbol{\theta}\right) \\ &= \arg\max_{\boldsymbol{\theta}} \prod_{i=1}^{m} p_{\text{model}}\left(x^{(i)}; \boldsymbol{\theta}\right)\end{aligned}$$

Cross Entropy Loss Function

- Numerically more stable: $\boldsymbol{\theta}_{ML} = \arg\max_{\boldsymbol{\theta}} \sum_{i=1}^{m} \log p_{\text{model}} \left(x^{(i)}; \boldsymbol{\theta}\right)$
- Because the arg max does not change when we re-scale the cost function, we can divide by $\mathbf{m}$ to obtain a version of the criterion that is expressed as an expectation with respect to the empirical distribution $\hat{p}_{\text{data}}$ defined by the training data:

$$\theta_{ML} = \arg\max_{\boldsymbol{\theta}} \mathbb{E}_{\mathbf{x} \sim \hat{p}_{\text{data}}} \log p_{\text{model}} \left(x; \boldsymbol{\theta}\right)$$

Cross Entropy Loss Function

- One way to interpret maximum likelihood estimation is to view it as minimizing the dissimilarity between the empirical distribution $\hat{p}_{\text{data}}$, defined by the training set and the model distribution, with the degree of dissimilarity between the two measured by the KL divergence.
- The KL divergence is given by

$$D_{KL}\left(\hat{p}_{\text{data}} \| p_{\text{model}}\right) = \mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}}\left[\log \hat{p}_{\text{data}}\left(x\right) - \log p_{\text{model}}\left(x\right)\right]$$

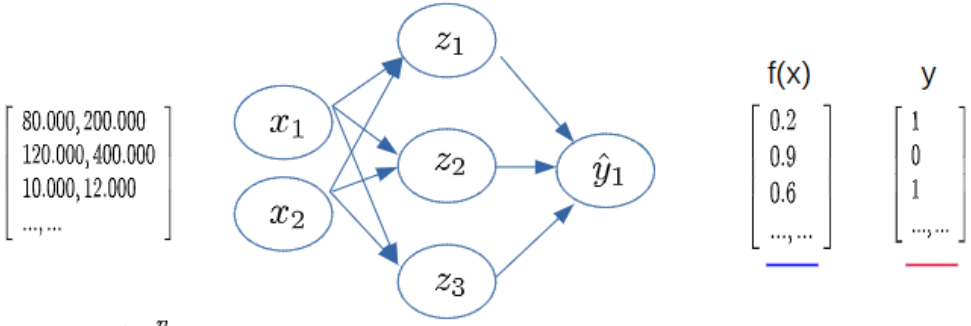
- The term on the left is a function only of the data-generating process, not the model.
  - This means when we train the model to minimize the KL divergence, we need only minimize

$$-\mathbb{E}_{\mathbf{x}\sim\hat{p}_{\text{data}}}\left[\log p_{\text{model}}\left(x\right)\right]$$

  - this of course the same as the maximization in equation.
- Any loss consisting of a negative log-likelihood is a cross-entropy between the empirical distribution defined by the training set and the probability distribution defined by model.

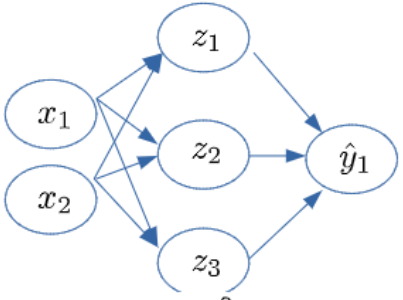
Binary Cross Entropy Loss

Our example was a classification problem with output (0 or 1)



Mean Squared Error (MSE)

$$\begin{bmatrix} 80.000, 200.000 \\ 120.000, 400.000 \\ 10.000, 12.000 \\ \dots, \dots \end{bmatrix}$$



f(x)	y
450.000	470.000
250.000	220.000
190.000	250.000
..., ...	..., ...
Loan requested	Loan required

$$J(W) = \frac{1}{n} \sum_{i=1}^{n} \left(y^{(i)} - f\left(x^{(i)}; W \right) \right)^2$$

Actual Predicted

Network Training

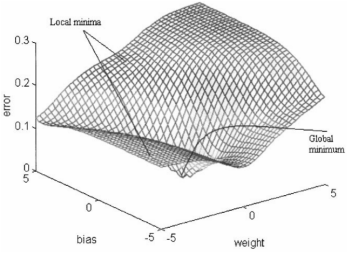
We want to find the network weights that achieve the lowest loss!

$$\boldsymbol{W}^* = \underset{\boldsymbol{W}}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \mathcal{L}\left(f\left(x^{(i)}; \boldsymbol{W}\right), y^{(i)}\right)$$

$$\boldsymbol{W}^* = \underset{W}{\operatorname{argmin}} J(\boldsymbol{W})$$

Gradient Descent in Weight Space

Goal: Given $(x_d, y_d)_{d \in D}$ find w to minimize $J(w) = \frac{1}{2} \sum_{d \in D} (f_w(x_d) - y_d)^2$

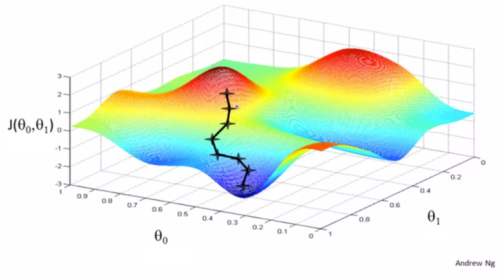


This error measure defines a Surface over the hypothesis (i.e. weight) space

Fig. from Cho & Chow, Neurocomputing 1999

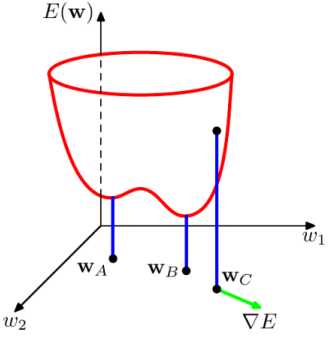
Gradient Descent in Weight Space

- $-W^* = \underset{W}{\text{argmin}} J(W)$
- Randomly pick an initial (w_0, w_1)
- Compute gradient
- Take small steps in the opposite direction of gradient.
- Repeat until convergence



Recall Parameter Optimization

- For either of these problems, the error function $\mathrm{J}(\mathrm{w})$ is nasty ($E(w)$ in the figure)
- Nasty = non-convex
- Non-convex = **has local minima**



Descent Methods In General

- The typical strategy for optimization problems of this sort is a descent method:

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

- These come in many flavors
  - Gradient descent $\nabla J\left(\boldsymbol{w}^{(\tau)}\right)$
  - Stochastic gradient descent $\nabla J_n\left(w^{(\tau)}\right)$
  - Newton-Raphson (second order) ∇^2
- All of these can be used here, stochastic gradient descent is particularly effective - Redundancy in training data, escaping local minima.

Gradient Descent Algorithm

Algorithm:

1. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$ $\longleftarrow$ **Can be computationally expensive**
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial W}$
5. Return weights

Gradient Descent Algorithm

Algorithm:

1. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$
2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$ $\longleftarrow$ **All that matters to train a NN**
4. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial W}$, where η is the **learning rate**
5. Return weights

Stochastic Gradient Descent

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$
2. Loop until convergence:
3. Pick single data point i
4. Compute gradient, $\frac{\partial J_i(W)}{\partial W} \longleftarrow$ Can be noisy
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

Stochastic (mini-batch) Gradient Descent

Algorithm

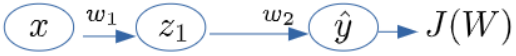
1. Initialize weights randomly $\sim \mathcal{N}\left(0, \sigma^2\right)$
2. Loop until convergence:
3. Pick batch of B data points
4. Compute gradient, $\frac{\partial J(W)}{\partial W} = \frac{1}{B} \sum_{\partial J=1}^{B} \frac{\partial J_k(W)}{\partial W}$
5. Update weights, $\boldsymbol{W} \leftarrow \boldsymbol{W} - \eta \frac{\partial J(\boldsymbol{W})}{\partial \boldsymbol{W}}$
6. Return weights

Mini-Batches While Training

- More accurate estimation of gradient
- Smoother convergence
- Allows for larger learning rates
- Mini-batches lead to fast training!
- Can parallelize computation + achieve significant speed increases on GPU's
- Note: a complete pass over all the patterns in the training set is called an epoch.

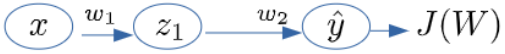
Computing Gradients: Error Backpropagation

- How does a small change in one weight (e.g., w_2) affect the final loss $\mathrm{J}(\mathrm{W})$?



Computing Gradients: Error Backpropagation

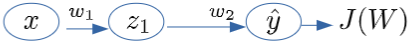
- How does a small change in one weight (e.g., w_2) affect the final loss $\mathrm{J}(\mathrm{W})$?
- Chain rule



$$\frac{\partial J(W)}{\partial w_2} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_2}$$

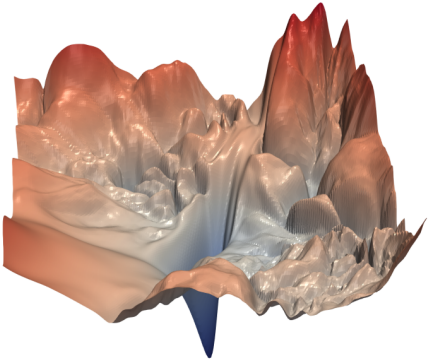
Computing Gradients: Error Backpropagation

- How does a small change in one weight (e.g., w_2) affect the final loss $J(W)$?
- Chain rule
- **Repeat this for every weight in the network using gradients from later layers**



$$\frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_1} \longrightarrow \frac{\partial J(W)}{\partial w_1} = \frac{\partial J(W)}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial z_1} * \frac{\partial z_1}{\partial w_1}$$

Training Neural Networks



See https://papers.nips.cc/paper/7875-visualizing-the-loss-landscape-of-neural-nets.pdf

Loss Function: Can Be Difficult to Optimize

- Remember:
  - Optimization through gradient descent: $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$
  - How can we set the learning rate?

Setting The Learning Rate

- Small learning rate converges slowly and gets stuck in false local minima
- Design an adaptive learning rate that "adapts" to the landscape.

A Few Variants of SGD

Method	Formula
Learning Rate	$w^{(t+1)} = w^{(t)} - \eta \cdot \nabla \ell(w^{(t)}, z) \quad = w^{(t)} - \eta \cdot \nabla w^{(t)}$
Adaptive Learning Rate	$w^{(t+1)} = w^{(t)} - \eta_t \cdot \nabla w^{(t)}$
Momentum [Qian 1999]	$w^{(t+1)} = w^{(t)} + \mu \cdot (w^{(t)} - w^{(t-1)}) - \eta \cdot \nabla w^{(t)}$
Nesterov Momentum [Nesterov 1983]	$w^{(t+1)} = w^{(t)} + v_t; \quad v_{t+1} = \mu \cdot v_t - \eta \cdot \nabla \ell(w^{(t)} - \mu \cdot v_t, z)$
AdaGrad [Duchi et al. 2011]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A_{i,t} + \varepsilon}}; \quad A_{i,t} = \sum_{\tau=0}^{t} \left(\nabla w_i^{(t)} \right)^2$
RMSProp [Hinton 2012]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot \nabla w_i^{(t)}}{\sqrt{A'_{i,t} + \varepsilon}}; \quad A'_{i,t} = \beta \cdot A'_{t-1} + (1-\beta) \left(\nabla w_i^{(t)} \right)^2$
Adam [Kingma and Ba 2015]	$w_i^{(t+1)} = w_i^{(t)} - \frac{\eta \cdot M_{i,t}^{(1)}}{\sqrt{M_{i,t}^{(2)} + \varepsilon}}; \quad M_{i,t}^{(m)} = \frac{\beta_m \cdot M_{i,t-1}^{(m)} + (1-\beta_m) \left(\nabla w_i^{(t)} \right)^m}{1 - \beta_m^t}$

For more details, see https://arxiv.org/pdf/1609.04747.pdf, and also

https://ruder.io/optimizing-gradient-descent/

Weight Initialization

- Before the training process starts: all weights vectors must be initialized with some numbers.
  - There are many initializers of which random initialization is one of the most widely known ones (e.g., with a normal distribution).
  - Specifically, one can configure the mean and the standard deviation, and once again seed the distribution to a specific (pseudo-)random number generator.
  - which distribution to use, then?
  - random initialization itself can become problematic under some conditions: you may then face the vanishing gradients and exploding gradients problems.
- What to do against these problems?
  - e.g. Xavier & He initialization (available in Keras)
  - They are different in the way how they manipulate the drawn weights to arrive at approximately 1 . By consequence, they are best used with different activation functions.
  - Specifically, He initialization is developed for ReLU based activating networks and by consequence is best used on those. For others, Xavier (or Glorot) initialization generally works best.

See, e.g., Glorot & Bengio (2010) - http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf; He et al. (2015) - https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf

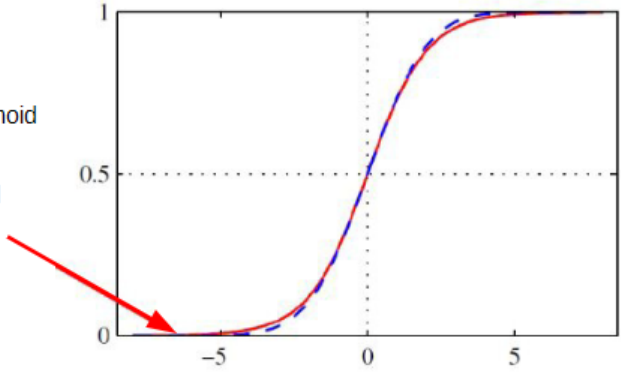
Vanishing Gradients

- Deep learning community often deals with two types of problems during training: vanishing gradients (and exploding) gradients.
  - Vanishing gradients
    - the backpropagation algorithm, which chains the gradients together when computing the error backwards, will find really small gradients towards the left side of the network (i.e., farthest from where error computation started).
    - This problem primarily occurs e.g. with the Sigmoid and Tanh activation functions, whose derivatives produce outputs of $0 < x' < 1$, except for Tanh which produces $x' = 1$ at $x = 0$.
    - Consequently, when using Tanh and Sigmoid, you risk having a suboptimal model that might possibly not converge due to vanishing gradients.
    - ReLU does not have this problem - its derivative is 0 when $x < 0$ and is 1 otherwise.
    - It is computationally faster. Computing this function - often by simply maximizing between $(0, x)$ - takes substantially fewer resources than computing e.g. the sigmoid and tanh functions. By consequence, ReLU is the de facto standard activation function in the deep learning community today.

Vanishing Gradients

Problem with Sigmoid
→ Saturation

Gradient too small



Vanishing Gradients

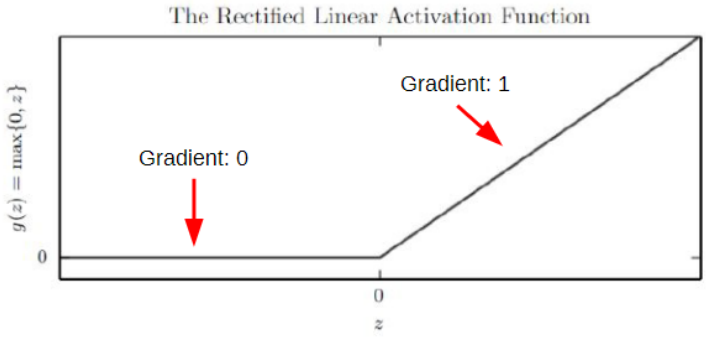


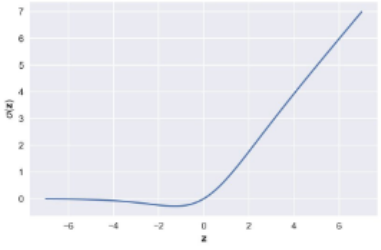
Fig. From Goodfellow et al. (2016)

Swish Activation Function

See, P Ramachandran, B Zoph, Q V. Le (2017) - https://arxiv.org/pdf/1710.05941v1.pdf

- Nevertheless, it does not mean that it cannot be improved.
  - Swish activation function.
  - Instead, it does look like the de facto standard activation function, with one difference: the domain around 0 differs from ReLU.
- Swish is a smooth function. That means that it does not abruptly change direction like ReLU does near $x = 0$.
  - Swish is non-monotonic. It thus does not remain stable or move in one direction, such as ReLU.
  - It is in fact this property which separates Swish from most other activation functions, which do share this monotonicity.
- In applications - Swish could be better than ReLu.

$$f(x) = x * \text{sigmoid}(x)$$
$$= x * \left(1 + e^{-x}\right)^{-1}$$



Intermezzo — Action Required

- Let's look (again) at this notebook for 2 minutes: → `01_GradientDescent_and_StochasticGradientDescent.ipynb`

A Geometric Interpretation

- In 3D, the following mental image may prove useful. Imagine two sheets of colored paper: one red and one blue.
- Put one on top of the other.
- Crumple them together into a small ball. That crumpled paper ball is your input data, and each sheet of paper is a class of data in a classification problem.
- What a neural network (or any other machine-learning model) is meant to do is figure out a transformation of the paper ball that would uncrumple it, so as to make the two classes cleanly separable again.
- With deep learning, this would be implemented as a series of simple transformations of the 3D space, such as those you could apply on the paper ball with your fingers, one movement at a time.

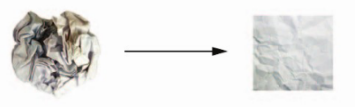
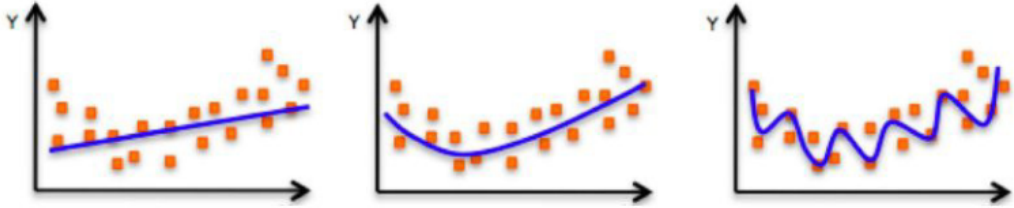


Figure 2.9 Uncrumpling a complicated manifold of data

Fig. from Chollet 2017

Notes On Overfitting



Early Stopping

Stop training before we have a chance to overfit

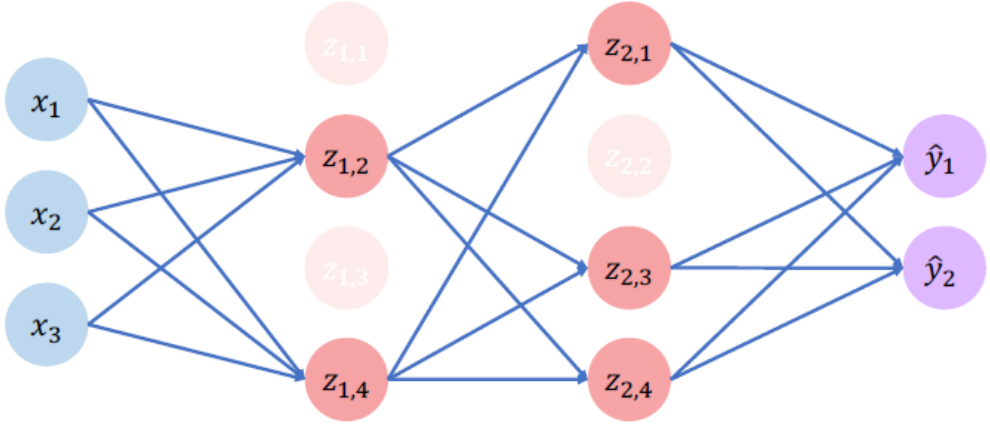
Notes On Regularization

- Regularization is a technique that constrains our optimization problem to discourage complex models.
- We use it to improve the generalization of our model on unseen data.

Regularization in NN: Dropout

Srivastava et al., 2014 --- http: //jmlr.org/papers/v15/srivastava14a.html

- During training, randomly set some activations to 0
- Typically 'drop' 50% of activations in layer
- Forces network to not rely on any node



Regularization In NN: Dropout

- It is an efficient way of performing model averaging with neural networks.
- Can be interpreted as some sort of bagging.
- Now, we assume that the model's role is to output a probability distribution. In the case of bagging, each model i produces a probability distribution $p^{(i)}(\mathrm{y} \mid x)$.
- The prediction of the ensemble is given by the arithmetic mean of all these distributions:

$$\frac{1}{k}\sum_{i=1}^{k} p^{(i)}(y \mid x)$$

- In the case of dropout, each sub-model defined by mask vector μ defines a probability distribution $p(\mathrm{y} \mid x, \mu)$.
- The arithmetic mean over all masks is given by $\sum p(\boldsymbol{\mu})p(y \mid x, \boldsymbol{\mu})$ where $p(\mu)$ is the probability distribution that was $^{\mu}$ use to sample μ at training time.

Remark: Batch Normalization

- https://arxiv.org/abs/1502.03167
- Batch normalization is used to stabilize and perhaps accelerate the learning process.
- It does so by applying a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1 .
- Suppose we built a neural network with the goal of classifying gray-scale images. The intensity of every pixel in a gray-scale image varies from 0 to 255. Prior to entering the neural network, every image will be transformed into a 1 dimensional array. Then, every pixel enters one neuron from the input layer. If the output of each neuron is passed to a sigmoid function, then every value other than 0 (i.e. 1 to 255) will be reduced to a number close to 1 . Therefore, it's common to normalize the pixel values of each image before training. Batch normalization, on the other hand, is used to apply normalization to the output of the hidden layers.

Building an MLP From Scratch

Marsland (2014)

Computing the computational complexity of this algorithm is very easy.

- The recall phase loops over the neurons, and within that loops over the inputs, so its complexity is $O(mn)$.
- The training part does this same thing, but does it for T iterations, so costs $O(Tmn)$.

The Perceptron Algorithm

- **Initialisation**
  - set all of the weights w_{ij} to small (positive and negative) random numbers
- **Training**
  - for T iterations or until all the outputs are correct:
    - for each input vector:
      - compute the activation of each neuron j using activation function g:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } \sum_{i=0}^{m} w_{ij}x_i > 0 \\ 0 & \text{if } \sum_{i=0}^{m} w_{ij}x_i \leq 0 \end{cases} \tag{3.4}$$

      - update each of the weights individually using:

$$w_{ij} \leftarrow w_{ij} - \eta(y_j - t_j) \cdot x_i \tag{3.5}$$

- **Recall**
  - compute the activation of each neuron j using:

$$y_j = g\left(\sum_{i=0}^{m} w_{ij}x_i\right) = \begin{cases} 1 & \text{if } w_{ij}x_i > 0 \\ 0 & \text{if } w_{ij}x_i \leq 0 \end{cases} \tag{3.6}$$

Perceptron Implementation

Marsland (2014)

```
for data in range(nData): # loop over the input vectors
  for n in range(N) : # loop over the neurons
  # Compute sum of weights times inputs for each neuron
  # Set the activation to 0 to start
    activation[data][n]= 0
  # Loop over the input nodes (+1 for the bias node)
  for m in range(M+1) :
    activation[data][n] += weight[m][n] * inputs[data][m]
  # Now decide whether the neuron fires or not
  if activation[data][n] > 0 :
    activation[data][n] = 1
  else:
    activation [data][n] = 0
```

```
# Compute activations
activations = np.dot(inputs,self.weights)
# Threshold the activations
return np.where(activations>0,1,0)
```

Perceptron Implementation (II)

- The weight update for the entire network can be done in one line (where eta is the learning rate, η):

```
self.weights-=eta*np.dot(np.transpose(inputs),self.activations-targets)
```

The Multi-layer Perceptron Algorithm

- **Initialisation**
  - initialise all weights to small (positive and negative) random values
- **Training**
  - repeat:
    - for each input vector:

      Forwards phase:

      - compute the activation of each neuron j in the hidden layer(s) using:

$$h_\zeta = \sum_{i=0}^{L} x_\iota v_{i\zeta} \tag{4.4}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)} \tag{4.5}$$

      - work through the network until you get to the output layer neurons, which have activations (although see also Section 4.2.3):

$$h_\kappa = \sum_j a_j w_{j\kappa} \tag{4.6}$$

$$y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)} \tag{4.7}$$

      Backwards phase:

      - compute the error at the output using:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa (1 - y_\kappa) \tag{4.8}$$

      - compute the error in the hidden layer(s) using:

$$\delta_h(\zeta) = a_\zeta (1 - a_\zeta) \sum_{k=1}^{N} w_\zeta \delta_o(k) \tag{4.9}$$

      - update the output layer weights using:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta \delta_o(\kappa) a_\zeta^{\text{hidden}} \tag{4.10}$$

      - update the hidden layer weights using:

$$v_\iota \leftarrow v_\iota - \eta \delta_h(\kappa) x_\iota \tag{4.11}$$

    - (if using sequential updating) randomise the order of the input vectors so that you don't train in exactly the same order each iteration
  - until learning stops (see Section 4.3.3)
- **Recall**
  - use the Forwards phase in the training section above

Recipe For Using MLP

- **Select inputs and outputs for your problem**
  - Before anything else, you need to think about the problem you are trying to solve and make sure that you have data for the problem, both input vectors and target outputs.
  - At this stage, you need to choose what features are suitable for the problem and decide on the output encoding that you will use - standard neurons or linear nodes.
  - These things are often decided for you by the input features and targets that you have available to solve the problem.
  - Later on in the learning it can also be useful to re-evaluate the choice by training networks with some input feature missing to see if it improves the results at all.

Recipe For Using MLP (II)

- **Normalize inputs**
  - Re-scale the data by subtracting the mean value from each element of the input vector, and divide by the variance (or alternatively, either the maximum or minus the minimum, whichever is greater).
- **Split the data into training, testing, and validation sets**
  - You cannot test the learning ability of the network on the same data that you trained it on, since it will generally fit that data very well (often too well, overfitting and modeling the noise in the data as well as the generating function).
  - Recall: we generally split the data into three sets, one for training, one for testing, and then a third set for validation, which is testing how well the network is learning during training.

Recipe For Using MLP (III)

- **Select a network architecture**
  - You already know how many input nodes there will be, and how many output neurons.
  - You need to consider whether you will need a hidden layer at all, and if so how many neurons it should have in it.
  - You might want to consider more than one hidden layer.
  - The more complex the network, the more data it will need to be trained on, and the longer it will take.
  - It might also be more subject to over-fitting.
  - The usual method of selecting a network architecture is to try several with different numbers of hidden nodes and see which works best.

Recipe For Using MLP (IV)

- **Train a network**
  - The training of the NN consists of applying the MLP algorithm to the training data.
  - This is usually run in conjunction with early stopping, where after a few iterations of the algorithm through all of the training data, the generalization ability of the network is tested by using the validation set.
  - The NN is very likely to have far too many degrees of freedom for the problem, and so after some amount of learning it will stop modeling the generating function of the data, and start to fit the noise and inaccuracies inherent in the training data. At this stage the error on the validation set will start to increase, and learning should be stopped.
- **Test the network**
  - Once you have a trained network that you are happy with, it is time to use the test data for the first (and only) time. This will enable you to see how well the network performs on some data that it has not seen before, and will tell you whether this network is likely to be usable for other data, for which you do not have targets.

Action Required — MLP

- The implementation is a batch version of the algorithm, so that weight updates are made after all of the input vectors have been presented.
- The central weight update computations for the algorithm can be implemented as:

```
deltao=(targets-self.outputs)*self.outputs*(1.0-self.outputs)
deltah = self.hidden(1.0-self.hidden)*np.dot(deltao,np.transpose(self.weights2))

updatew1 = np.zeros((np.shape(self.weigths1)))
updatew2 = np.zeros((np.shape(self.weigths2)))

updatew1 = eta*(np.dot(np.transpose(inputs),deltah[:,:-1]))
updatew1 = eta*(np.dot(np.transpose(self.hidden),deltao))

self.weights1 += updatew1
self.weights2 += updatew2
```

Evaluating A Classifier

- How can we assess the prediction quality of a classifier?
- Initially, we'll consider the case of binary classification (and extend it later to multi-class classification).
- Confusion matrix shows the performance of a classifier.

		Predicted	
		0 (No)	**1** (Yes)
Actual	**0** (No)	True Negatives (TN)	False Positives (FP)
	1 (Yes)	False Negatives (FN)	True Positives (TP)

Action Required — MLP (II)

- There are a few improvements that can be made to the algorithm, and there are some important things that need to be considered:
  - how many training data points are needed
  - how many hidden nodes should be used
  - how much training the network needs
- We will look at the improvements first, and then move on to practical considerations.
- The first thing that we can do is check that this MLP can indeed learn the logic functions, especially the XOR.
- See **02_Multi-layer_Perceptron.ipynb**

Recall: Different Activations

- In the algorithm described above, we used sigmoid neurons in the hidden layer and the output layer.
- This is fine for classification problems, since there we can make the classes be 0 and 1.
- However, we might also want to perform regression problems, where the output needs to be from a continuous range, not just 0 or 1 .
- The sigmoid neurons at the output are not very useful in that case. We can replace the output neurons with linear nodes that just sum the inputs and give that as their activation.
- This does not mean that we change the hidden layer neurons; they stay exactly the same, and we only modify the output nodes.
- They are not models of neurons anymore, since they don't have the characteristic fire/don't fire pattern.
- Even so, they enable us to solve regression problems, where we want a real number out, not just a 0/1 decision.

Different Activation Function

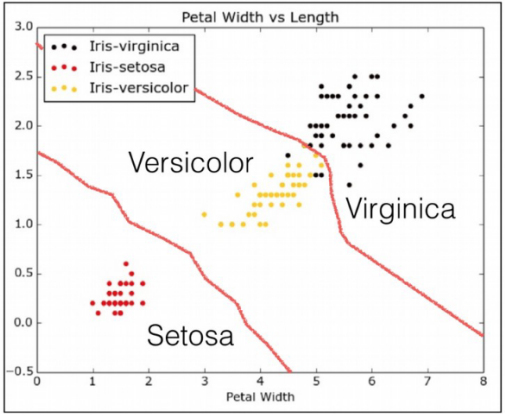
02_Multi-layer_Perceptron.ipynb

```
# Different types of output neurons
if self.outtype == 'linear':
  return outputs
elif self.outtype == 'logistic':
    return 1.0/(1.0+np.exp(-self.beta*outputs))
elif self.outtype == 'softmax':
    normalisers = np.sum(np.exp(outputs),axis=1)*np.ones((1,np.shape(outputs)[0]))
    return np.transpose(np.transpose(np.exp(outputs))/normalisers)
else:
    print("error")
```

Test — Iris Data Set

https://archive.ics.uci.edu/ml/datasets/Iris

Iris Versicolor | Iris Setosa | Iris Virginica



Test — MNIST

- http://yann.lecun.com/exdb/mnist/
- The MNIST database of handwritten digits, available from this page, has a training set of 60,000 examples, and a test set of 10,000 examples. It is a subset of a larger set available from NIST. The digits have been size-normalized and centered in a fixed-size image.
- MNIST: Modified National Institute of Standards and Technology database.
- Action required - see `02_Multi-layer_Perceptron.ipynb`.

Action Required

Look at the test functions on the right-hand side*. Pick three of those test functions (from Genz 1987).

- Approximate a 2-dimensional function stated below with Neural Nets based 10,50,100,500 points randomly sampled from $[0,1]^2$. Compute the average and maximum error.
- The errors should be computed by generating 1,000 uniformly distributed random test points from within the computational domain.
- Plot the maximum and average error as a function of the number of sample points.
- Repeat the same for 5-dimensional and 10-dimensional functions. Is there anything particular you observe?

*Choose the parameters w and c in meaningful ways.

$$\text{oscillatory: } f_1(x) = \cos\left(2\pi w_1 + \sum_{i=1}^{d} c_i x_i\right),$$

$$\text{product peak: } f_2(x) = \prod_{i=1}^{d} (c_i^{-2} + (x_i - w_i)^2)^{-1},$$

$$\text{corner peak: } f_3(x) = \left(1 + \sum_{i=1}^{d} c_i x_i\right)^{-(d+1)},$$

$$\text{Gaussian: } f_4(x) = \exp\left(-\sum_{i=1}^{d} c_i^2 \cdot (x_i - w_i)^2\right),$$

$$\text{continuous: } f_5(x) = \exp\left(-\sum_{i=1}^{d} c_i \cdot |x_i - w_i|\right),$$

$$\text{discontinuous: } f_6(x) = \begin{cases} 0, & \text{if } x_1 > w_1 \text{ or } x_2 > w_2, \\ \exp\left(\sum_{i=1}^{d} c_i x_i\right), & \text{otherwise.} \end{cases}$$

Varying test functions can be obtained by altering the parameters $c = (c_1, \ldots, c_n)$ and $w = (w_1, \ldots, w_n)$. We chose these parameters randomly from $[0, 1]$. Similarly to Barthelmann et al. [2000], we normalized the c_i such that $\sum_{i=1}^{d} c_i = b_j$, with b_j depending on d, f_j according to

j	1	2	3	4	5	6
b_j	1.5 d		1.85	7.03	20.4	4.3

Furthermore, we normalized the w_i such that $\sum_{i=1}^{d} w_i = 1$.

Action Required (II)

- Play with the architecture.
  - Number of hidden layers.
  - activation functions.
  - choice of the stochastic gradient descent algorithm.
  - Monitor the performance with respect to the architecture.

Questions?