COMPUTER VISION 2018 - 2019

>INTRODUCTION TO NEURAL COMPUTATIONS

UTRECHT UNIVERSITY

ALEXANDROS STERGIOU

OUTLINE

Introduction to Artificial Neurones.

Revisiting history: Perceptrons

How learning works.

Backpropagation algorithm.

Programmatic implementations.

FROM BIOLOGICAL TO ARTIFICIAL NEURONES

REASONS FOR STUDYING NEURAL NETWORKS

Have a good understanding of how the brain functions.

- Very large and complicated in which signals travel to different routes

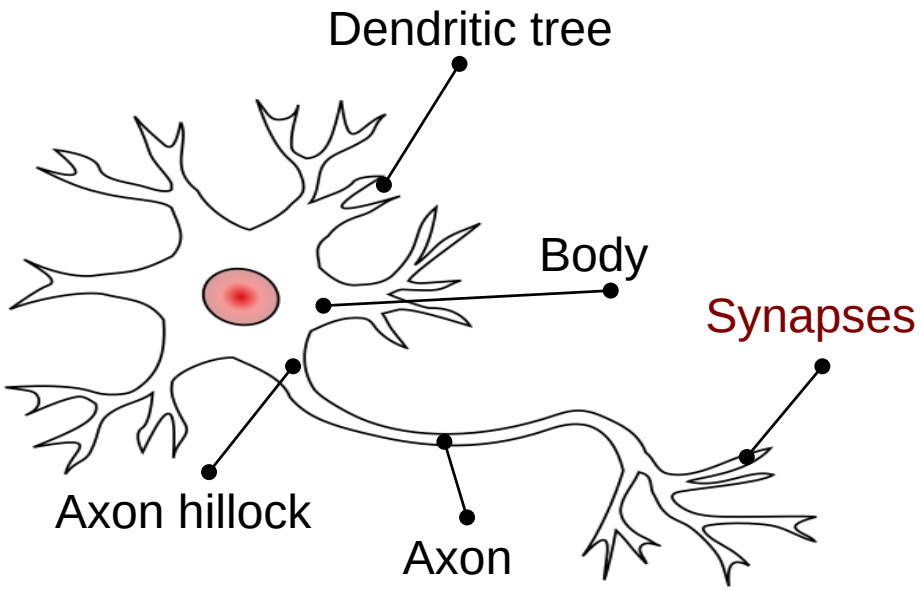
Better grasp the notion of parallel computations and the connection to neurones.

- Different from sequential computations (as in most programs)

Solve practical problems inspired by the brain's functions (our course).

- May not always be based on how the brain works but provide useful solutions to given task(s)

CORTICAL NEURONES

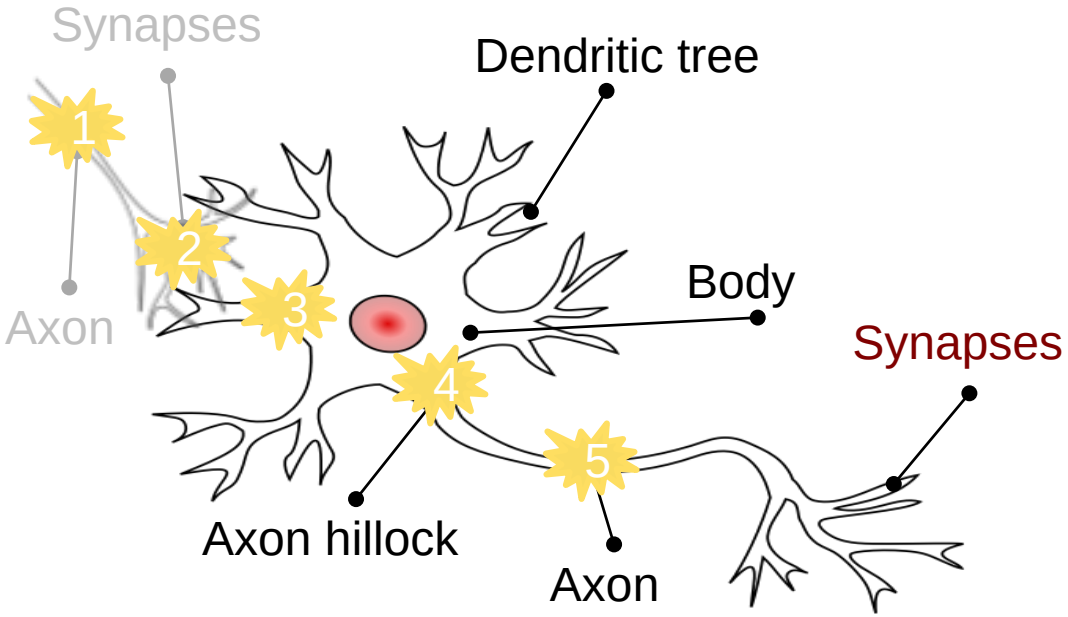


Dendritic tree: receive signal from other neurones.

Axon: send signal to other neurones

Synapses: connection/contact point of axon and dendritic tree

CORTICAL NEURONES[2] – HOW THE SIGNAL TRAVELS



1. A spike of activity travels along the axon.
2. Charge in injected to the post-synaptic neurone at the synapses.
3. A neurone will generate spike if dendritic tree is charged enough.
4. If this charge is large enough the axon hillock is depolarised.
5. The spike is then carried by the axon.

CLOSER LOOK ON SYNAPSES

When spikes that travel along the axon and arrive at the synapses, they cause vesicles of transmitter chemicals to be released. They can include:

- Positive charges
- Negative charges

Based on these chemicals, holes are created in the membrane allowing specific ions in/out.

SUMMARY OF HOW SIGNALS TRAVEL

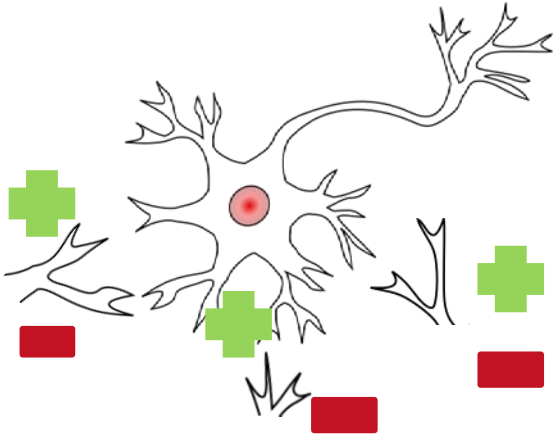
Neurones receive signals from other neurones

- For cortical neurones these signals are called spikes.

The effect of neurones input is controlled by the synaptic weight:

- Weights can be negative or positive

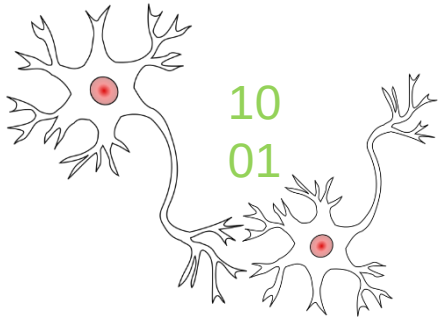
The synaptic weights adapt based on the signals so the network can learn to perform useful tasks.



SYNAPSES IN PROGRAMMATICAL TERMS

If synapses were to be programmed they would be very slow and require a lot of RAM.

- They would require very little power and they would be very small.
- They would adapt based on the signals that were given.



We have roughly 10^{11} neurones each of which has about 10^4 weights. That is more than any workstation.

IDEALISING NEURONES

Modelling things programmatically requires idealisation

- **Idealisation :** The process of creating simplifications of complicated details in order to understand the main principles by allowing to apply mathematics and make analogies to other familiar systems.

Once the main principles are understood, it would be much more easy to increase complexity.

It is also important to make sure we do not remove the main properties of what we idealising.

- e.g. Artificial neurones do communicate with each other but not through spikes of activity, but with discrete values.

LINEAR NEURONES

Simple but computationally limited.

- If they can learn we may have insight on more complicated neurones.

bias

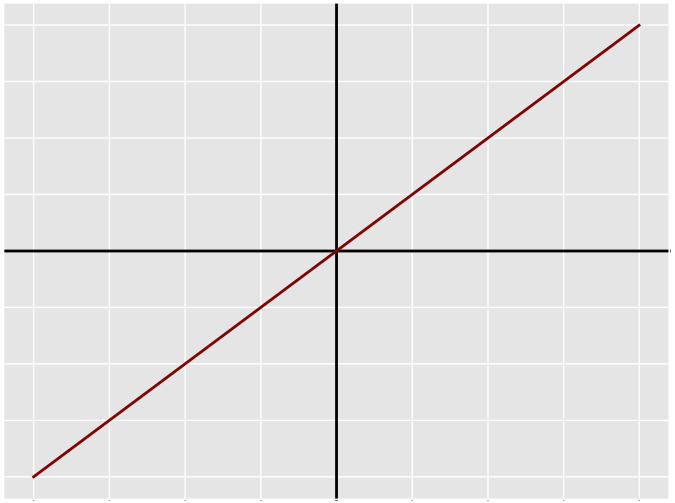
i^{th} input

$$y = b + \sum_{i} x_i w_i$$

output

index over input connections

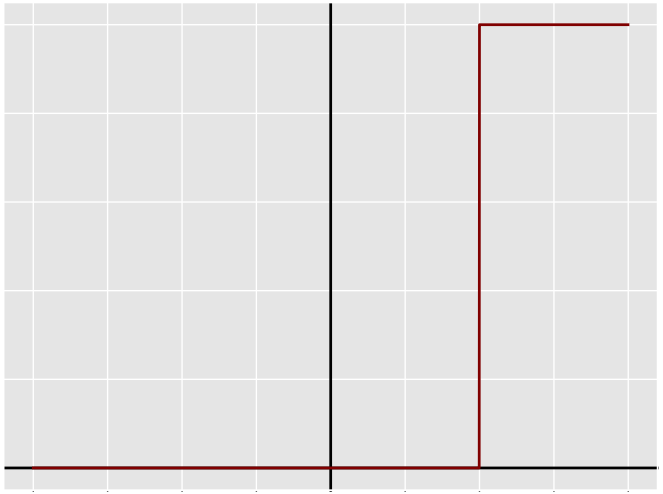
weight on i^{th} input



BINARY THRESHOLD NEURONES

McCulloch & Pitts (1943) and influenced by Von Neumann

- Start by computing a weighted sum of inputs
- Send out a fixed size spike of activity if weighted sum exceeds threshold
- McCulloch and Pitts thought that each spike is similar to the truth value of proposition and each neurone combines truth values to compute the truth value of another proposition.



[1] The McCulloch-Pitts Model of Neuron: [link]

BINARY THRESHOLD NEURONES[2]

There are two equivalent ways of writing the equation for a binary threshold neurone:

$$z = \sum_i x_i w_i$$

$$y = \begin{cases} 1, if\ z \geq \vartheta \\ 0, otherwise \end{cases}$$

$$\vartheta = -b$$

$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1, if\ z \geq 0 \\ 0, otherwise \end{cases}$$

Python snippet:

```
import numpy as np
def y_bthold(z, alpha=.3):
    return np.where(z >= alpha, 1, 0)
```

Python snippet:

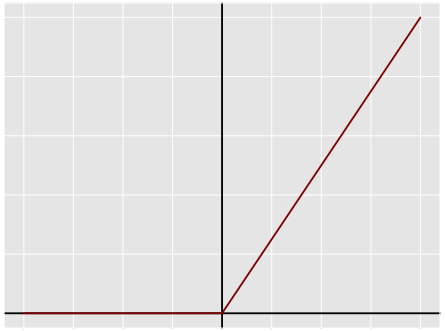
```
import numpy as np
def y_bthold(z, alpha=.3):
    return np.where(z >= 0, 1, 0)
```

RECTIFIER LINEAR NEURONES

They compute a linear weighted sum of their inputs.

The output is a non-linear function of the total input.

- Keep the nice properties of linear systems above zero.
- Have the ability of making decisions in close-to-zero cases.



Python snippet:

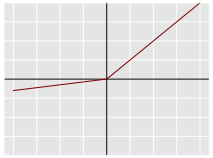
```
import numpy as np
def y_relu(z):
    return np.where(z >= 0, z, 0)
```

$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} z, if\ z \geq 0 \\ 0, otherwise \end{cases}$$

[1] Rectified linear units improve restricted boltzmann machines: [link]

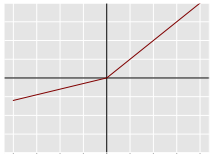
RECTIFIER LINEAR NEURONES[2] - VARIANTS



Python snippet:

```
import numpy as np
def y_leakyrelu(z):
    return np.where(z >= 0, z, 0.01*z)
```

$$y = \begin{cases} z, if\ z \geq 0 \\ 0.01z, otherwise \end{cases}$$



Python snippet:

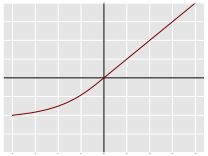
```
import numpy as np
def y_prelu(z, alpha=.2):
    return np.where(z >= 0, z, alpha*z)
```

$$y = \begin{cases} z, if\ z \geq 0 \\ az, otherwise \end{cases}$$

Python snippet:

```
import numpy as np
import math
def y_elu(z, alpha=.3):
    return np.where(z < 0, z / math.sqrt(1+alpha*math.pow(z,2)), z)
```

$$y = \begin{cases} z, if\ z \geq 0 \\ a(e^z - 1), otherwise \end{cases}$$



Python snippet:

```
import numpy as np
import math
def y_isrlu(z, alpha=3):
    return np.where(z < 0, z / math.sqrt(1+alpha*math.pow(z,2)), z)
```

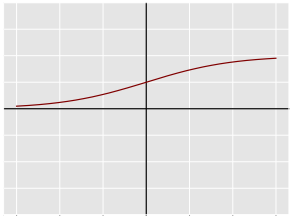
$$y = \begin{cases} z, if\ z \geq 0 \\ \frac{z}{\sqrt{1+az^2}}, otherwise \end{cases}$$

[1] Rectifier Nonlinearities Improve Neural Network Acoustic Models [link]
[2] Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification [link]
[3] Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs) [link]
[4] Improving Deep Learning by Inverse Square Root Linear Units (ISRLUs) [link]

SIGMOID AND TANH NEURONES

$$z = b + \sum_i x_i w_i$$

$$y = \frac{1}{1 + e^{-z}}$$



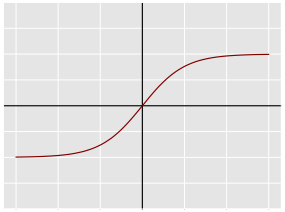
Both give real-valued outputs that is a smooth and bounded function of their total input.

- Sigmoid can be easily interpreted in probabilities as values only range from 0 to 1. Also being a **differentiable function** we can find the slope/derivative at any point of the function.

- Tanh is a re-scaled version of the sigmoid in which the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero.

$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} e^{z} - e^{-z} \\ e^{z} + e^{-z} \end{cases}$$



Python snippet:

```
import numpy as np
def y_sigmoid(z):
    return (1/(1+np.exp(-z)))
```

Python snippet:

```
import numpy as np
def y_tanh(z):
    return ((np.exp(z)-np.exp(-z))/(np.exp(z)+np.exp(-z)))
```

OVERVIEW OF NEURAL NETWORKS

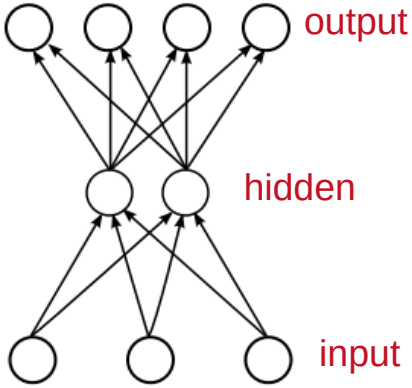
FEED FORWARD NETWORKS

- In practical applications these are the commonest types

of neural network types.

- The first layer is the input and the last is the output.
- If there is more than one hidden layer, we call them

“deep” neural networks.



- They compute a series of transformations that change the similarities between cases.

- The activities of the neurones in each layer are non-linear functions of the activities in the layer below.

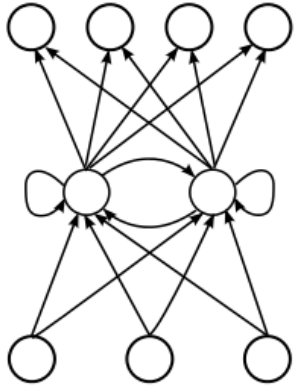
FEED FORWARD NETWORKS[2]: PYTHON TWO LAYER NET

```
X = [[1,1,1],[1,0,1],[1,1,0],[0,1,1],[0,0,1],[0,0,0]] # input dataset
Y = np.array([[1,1,1,1,0,0]]).T # output dataset
syn0 = 2*np.random.random((3,1)) – 1 # random weights
for  i in xrange(_iterations_):
   input  =  X
   layer1  =  relu(np.dot(input,syn0)) #pass information(forward propagation)
   layer1_error = y – layer1 # loss between predicted and actual values
   # multiply loss by slope of relu values for layer 1
   # slope should be the first derivative of the function
   layer1_delta = layer1_error * slope_relu(layer1)
   syn0 += X.T.dot(input.T,layer1_delta) # update weights
```

RECURRENT NETWORKS

- In contrast to feed-forward networks, recurrent networks also include cycles in their connection graph.

- This means that neurones can be accessed multiple times when information is fed to the network.

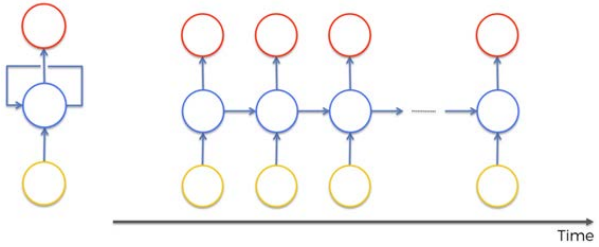


- Their dynamics are complicated and this makes them difficult to train.

- There currently is a lot of interest on efficiently training recurrent networks.

- They resemble biological neurones more than feed-forward neurones.

RECURRENT NETWORKS[2]



Recurrent neural networks are very efficient with modelling sequential data:

- They can be imagined as very deep networks with a single hidden layer per time slice.
- But they also use the same weights at every time slice and they get inputs are every time slice.

They have the ability to remember information in their hidden state for a long time.

- But it is very difficult to train them to use this potential.

FIRST GENERATION OF NEURAL NETWORKS

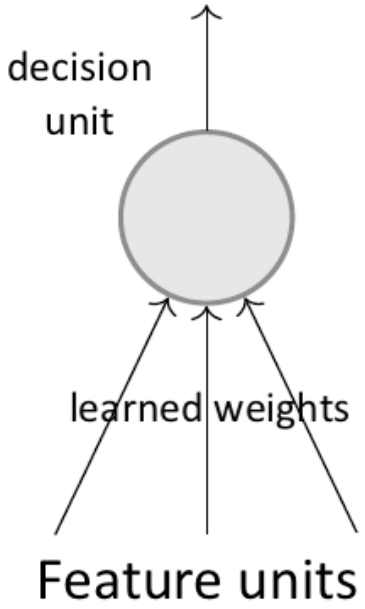
STANDARD PARADIGM FOR STATISTICAL PATTERN RECOGNITION

1. **Feature units:** Convert raw input vector into a vector of feature activations.

Use hand-written programs based on common-sense to define the features.

2. **Learned weights:** Learn how to weight each of the feature activations to get a single scalar quantity.

3. **Decision unit:** If the quantity is above a certain threshold, decide that the input vector is a positive example of the target class.



THE HISTORY OF PERCEPTRONS

- They were initially proposed by F. Rosenblatt in 1957 [link] and popularised in the 60s.
- They appeared to be very powerful and be able to learn many tasks (mostly binary).
- Plenty of claims were made for their capabilities and potential.
- In 1969, Minsky and Papert published a book named “Perceptrons” that analysed what they could do and showed their limitations.
- Based on this, many people thought that limitations were applied to all neural network models.
- The learning procedure of perceptrons is still used today for tasks with large feature vectors that contain millions of features.

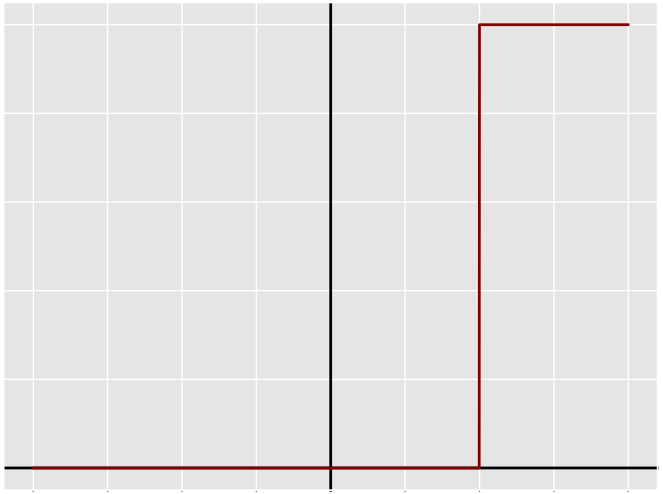
BINARY THRESHOLD NEURONES (AS DECISION UNITS)

McCulloch & Pitts (1943)

- Start by computing a weighted sum of inputs.
- Output 1 if weighted sum exceeds zero.

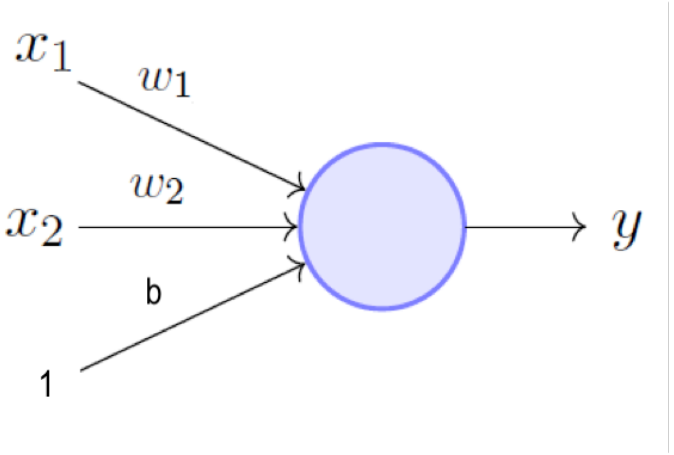
$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1 & z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$



HOW TO LEARN BIASES SAME WAY AS WEIGHTS

- A threshold is equivalent to having a negative bias.
- We can also avoid having to figure out a separate learning rule for the bias by using a trick:
- A bias is exactly equivalent to a weight on an extra input line that always has an activation of 1.



THE PERCEPTRON CONVERGENCE

- Add an extra component with value 1 to each vector. The “bias” weight on this component is minus the threshold. Now we can forget the threshold.
- Pick training cases using any policy that ensures that every training case will keep getting picked.
  - If the output unit is correct, leave the weights alone.
  - If the output unit incorrectly outputs zero, add the input vector to the weight vector.
  - If the output unit incorrectly outputs a 1, subtract the input vector from the weight vector.
- This is guaranteed to find a set of weights that gets the right answer for all the training cases if any as such exist.

SOME WORDS ABOUT THE MATH TO FOLLOW

- For people who are not as familiar with math notations the topics from this point forward may be a bit more difficult to follow.
- You may have to spend some more time on studying the next slides.
- If you are not used to talking about hyper-planes and high-dimensional spaces, now is the time to familiarise yourself.
- When talking about multi-dimensional plains (e.g. 32-dimansional space), visualise a 3D space and say “32” to yourself - that’s at least what I do :)
- REMEMBER: Going from 32 dimensions to 33, creates the same complexity as going from 2D to 3D.

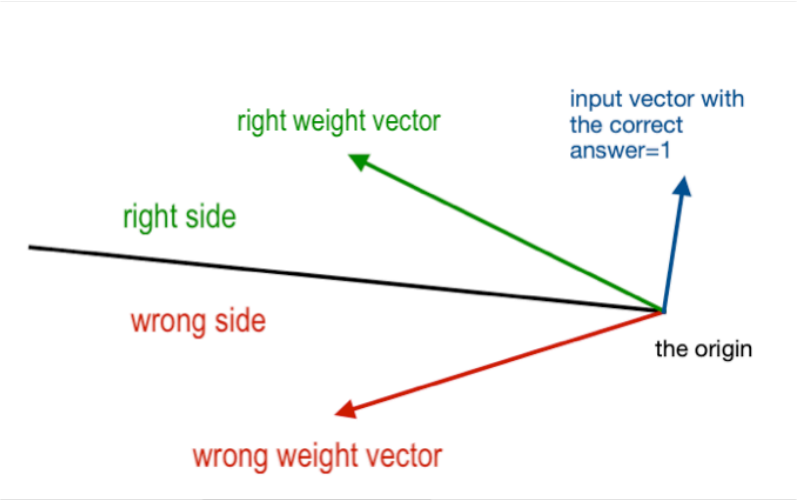
WEIGHT SPACE

- This space has one dimension per weight
- A point in the space represents a particular set of weights.
- Assuming that we have eliminated the threshold, each training case can be represented as a hyperplane through the origin.

- The weights must lie on one side of this hyper-plane to get the correct answer.

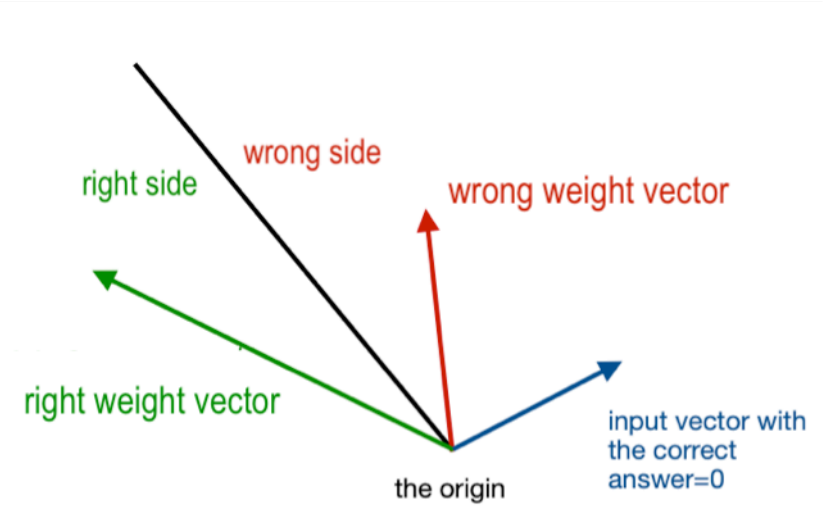
WEIGHT SPACE: TRAINING CASE WITH ANSWER BEING 1

- Each training case defines a plane (shown as a black line)
- The plane goes through the origin and is perpendicular to the input vector.
- On one side of the plane the output is wrong because the scalar product of the weight vector with the input vector has the wrong sign.



WEIGHT SPACE: TRAINING CASE WITH ANSWER BEING 0

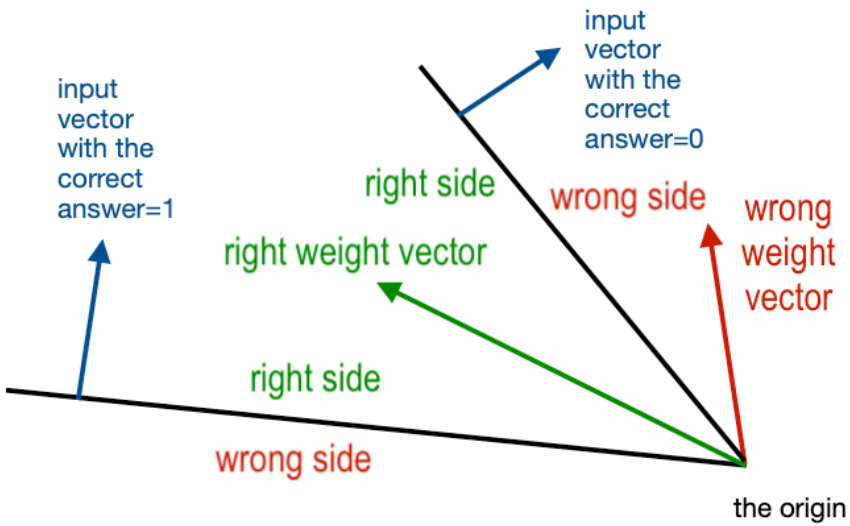
- Each training case defines a plane (shown as a black line)
- The plane goes through the origin and is perpendicular to the input vector.
- On one side of the plane the output is wrong because the scalar product of the weight vector with the input vector has the wrong sign.



WEIGHT SPACE: THE CONE OF FEASIBLE SOLUTIONS

- To get all training cases right we need to find a point on the right side of all the planes.
- There may not be any such point.
- If there are any weight vectors that get the right answer for all cases, they lie in a hyper-cone with its apex in at the origin.
- So the average of two good weight vectors is a good weight vector.

The problem is convex.

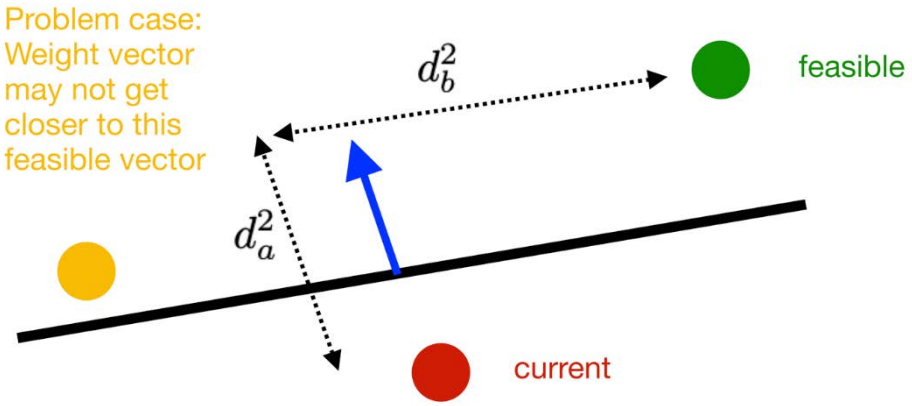


WHY THE PERCEPTRON LEARNING WORKS

- Consider the squared distance

$d^2_a + d^2_b$ between any feasible weight vector and the current weight vector.

- Hopeful claim: Every time the perceptron makes a mistake, the learning algorithm moves the current weight vector closer to all feasible weight vectors



A PROOF FOR CONVERGENCE

- Each time the perceptron makes a mistake, the current weight vector moves to decrease its squared distance from every weight vector in the “generously feasible” region.
- The squared distance decreases by at least the squared length of the input vector.
- So after a finite number of mistakes, the weight vector must lie in a feasible region if this region exists.

THE LIMITATIONS OF PERCEPTRONS

- If you are allowed to choose the features by hand and if you use enough features, you can do anything.
- For each binary input vectors, we can have a separate feature unit for each of the exponentially many binary vectors and so we can make any possible discrimination on binary input vectors.

    - This type of look-up does not generalise and is prone to heavily-overfitting the dataset.

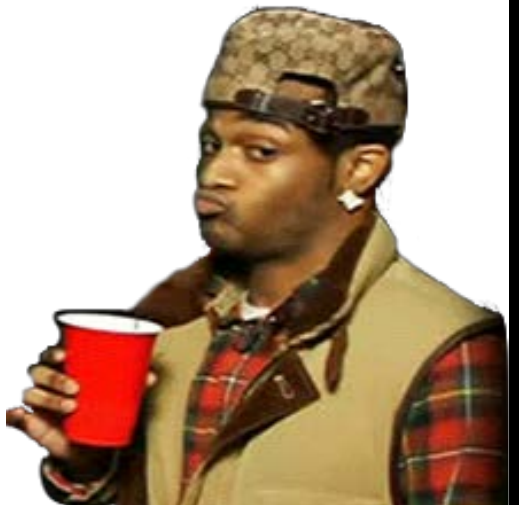
- Once the hand-coded features are determined, there are very strong limitations on what the perceptron can actually learn.

THE LIMITATIONS OF BINARY THRESHOLD NEURONES

- A binary threshold neurone can never tell if two single bit features are the same.
- Positive cases: (1,1) -> 1 , (0,0) -> 1
- Negative cases: (1,0) -> 0 , (0,1) -> 0
- These four pairs produce four inequalities:

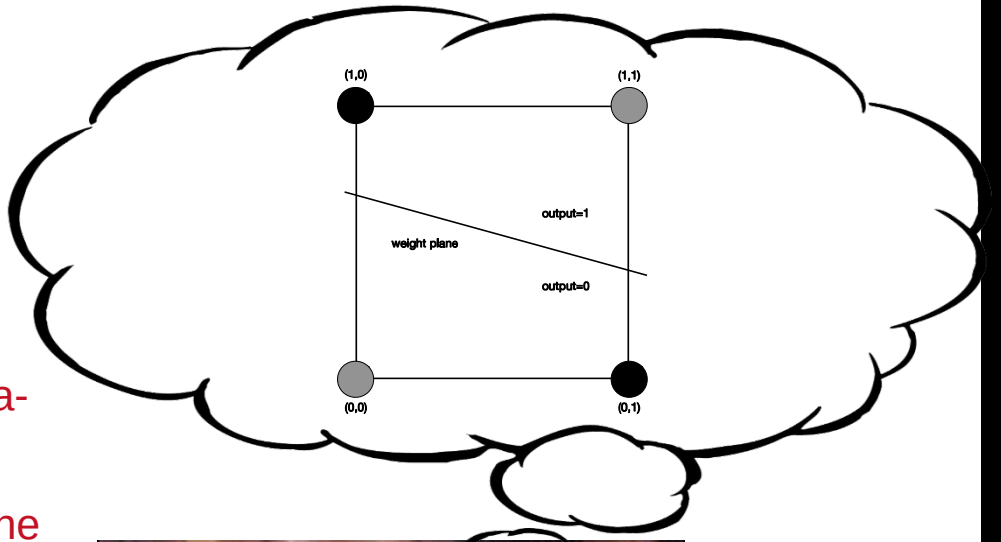
(1) $1 * w_1 + 1 * w_2 \geq \theta => w_1 + w_2 \geq \theta$
(2) $0 * w_1 + 0 * w_2 \geq \theta => 0 \geq \theta$
(3) $1 * w_1 + 0 * w_2 \leq \theta => w_1 \leq \theta$
(4) $0 * w_1 + 1 * w_2 \leq \theta => w_2 \leq \theta$

$(1) + (2) => w_1 + w_2 \geq 2\theta$
$(3) + (4) => w_1 + w_2 \leq \theta$



THE LIMITATIONS OF BINARY THRESHOLD NEURONES (GEOMETRICALLY)

- Imagine a “data-space” in which the axes correspond to components of an input vector.
- Each input vector is a point in this space.
- A weight vector defines a plane in data-space.
- The weight plane is perpendicular to the weight and misses the origin by a distance equal to the threshold.

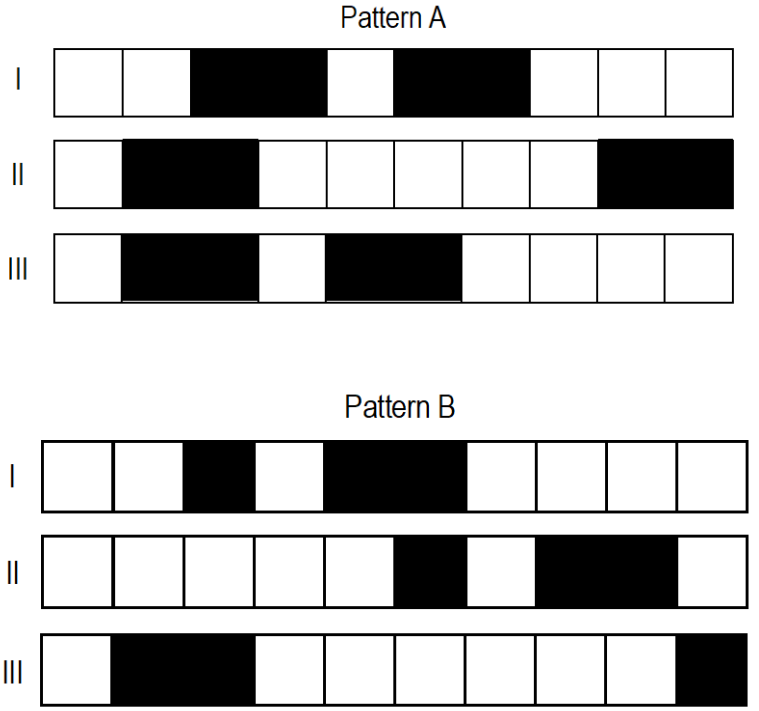


DISCRIMINATING SIMPLE PATTERNS UNDER TRANSLATION WITH WRAP-AROUND

- Suppose we just use pixels as features.

- Can a binary threshold unit discriminate between different patterns that have the same number of pixels?

- Not if the patterns can translate with wrap-around…



WHY IS THIS A PROBLEM?

- The whole point of pattern recognition is to recognise patterns despite transformations like translation.
- Misky and Papert's "Group Invariance Theorem" states that the part of a Perceptron that learns cannot learn to do this if the transformations form a group.
- Translation with wrap-around form a group.
- To deal with such transformations, a Perceptron needs to use multiple feature units to recognise transformations of informative sub-patterns.
- So in pattern recognition the tricky part is solved by hand-coded feature detectors and not by the learning procedure.

LEARNING WITH HIDDEN UNITS

- Networks that do not include any hidden units are very limited in the input-output mappings they can learn to model.
- More linear layers do not help as it is still linear.
- Fixed output linearities are not enough.
- What is required is more layers of adaptive, non-linear hidden-units.
- We need an efficient way of adapting all the weights and not just the last layer.
- Learning the weights going to hidden units is equivalent to learning features.
- This is difficult because no one is giving feedback on what each of the hidden units should do.

LEARNING WITH LINEAR NEURONES

A WAY TO SHOW THAT LEARNING MAKES PROGRESS

The expected behaviour of any network is that as the learning progresses, the actual output values get closer to the target values.

- This can be true even for problems in which there are many different sets of weights that work well and averaging two good sets of weights could lead to a bad set of weights.

Thus the simplest example of such procedure is a linear neurone with a square error measure.

A CLOSER LOOK AT LINEAR NEURONES (AGAIN)

- The neurone has a real-valued output which is a weighted sum of its inputs.
- The aim of learning is to minimise the error summed over all training cases.
- The error is squared difference between the desired output and the actual output. $E = (y - t)^2$

weight vector

$$y = \sum_i w_i x_i = \mathbf{W}^T \mathbf{X}$$

input vector

neurone's estimate of desired output

WHY NOT USE AN ANALYTICAL APPROACH

- It is straight-forward to write down a set of equations, one per training case, and to solve for the best set of weights.
- This is the standard engineering approach so why not use it?
- Scientific answer: We want a method that real neurones could use.
- Engineering answer: We want a method that can be generalised to multi-layer, non-linear neural networks.
- The analytic solution relies on it being linear and having a squared error measure.
- Iterative methods are usually less efficient but are much easier to generalise.

SIMPLE EXAMPLE TO ILLUSTRATE THE LEARNING ITERATIONS

- Every day you grab lunch from the cafeteria which only makes donuts apart from coffee.
- Your options are original glazed, chocolate iced and cherry filled.
- You get portions of each.
- The cashier will only tell you the total price of what you brought.
- After a few days you should be able to figure out the price of each portion.
- You start by random guessing the prices and the adjust them to better fit the observed prices of the whole meals.

SOLVING THE PROBLEM ITERATIVELY

- Each meal price gives a linear constraint on the price of the portions:

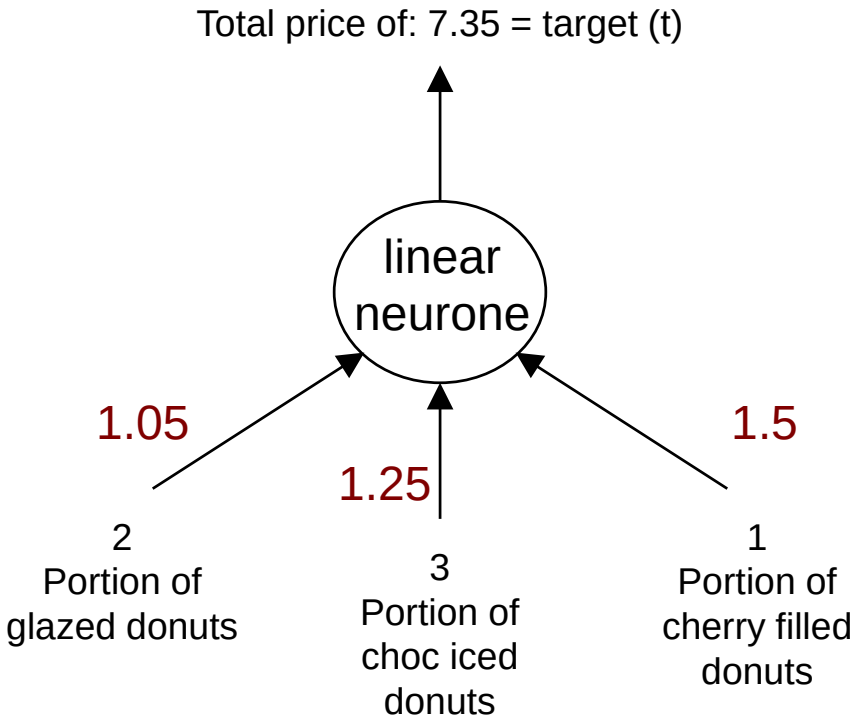
$$price = W_{glazed}X_{glazed} + W_{choc}X_{choc} + W_{cherry}X_{cherry}$$

- The prices of the portions are like the weights of a linear neurone.

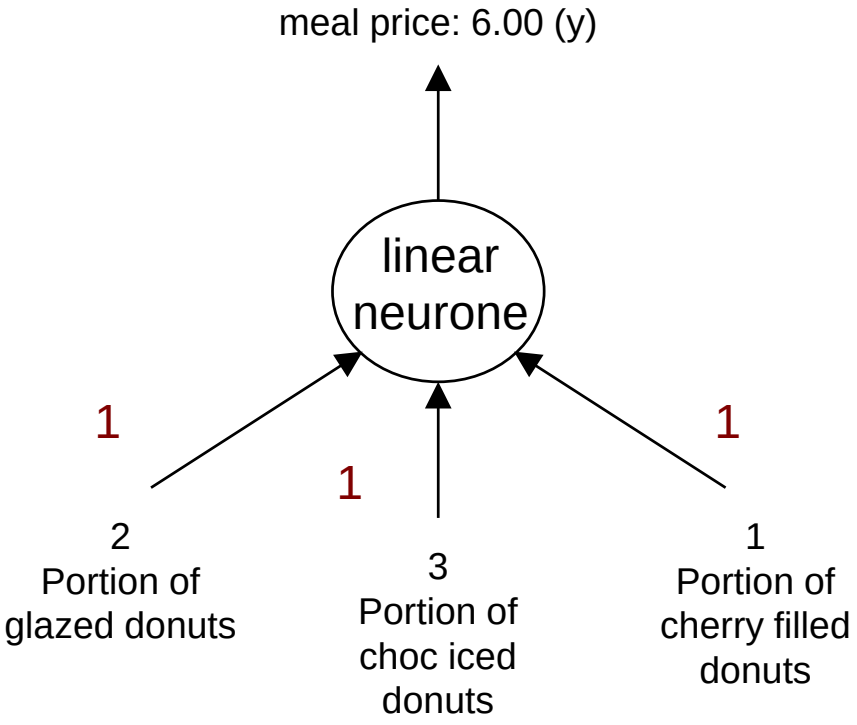
$$\mathbf{W} = (W_{glazed}, W_{choc}, W_{cherry})$$

- We start with guesses for the weights and then adjust the guesses slightly to give a better fit to the prices given by the cashier.

SOLVING THE PROBLEM ITERATIVELY[2]



SOLVING THE PROBLEM ITERATIVELY[2]



- Residual error (t - y) = 1.35
- The delta rule for learning is:

$$\Delta W_i = \varepsilon X_i (t - y)$$

- With a learning rate $\mathcal{E}$ of 1/25 the weight changes are:

  i.(+0.108) for glazed donuts.

  ii.(+0.162) for chocolate iced donuts.

  iii.(+0.054) for cherry filled donuts.

So the new weights are: 1.108, 1.162 and 1.054 .

- (!) Notice that the weights of glazed donuts got worse.

MORE ABOUT THE DELTA RULE

- Define the error as the squared residuals summed over all training cases.

$$E = \frac{1}{2} \sum_{n \in training} (t^n - y^n)^2$$

- Differentiate to get error derivatives for weights.

$$\frac{\partial E}{\partial W_i} = \sum_n \frac{\partial y^n}{\partial W_i} \frac{dE^n}{dy^n} = -\sum_n x_i^n (t^n - y^n)$$

- The batch delta rule changes the weights in proportion to their error derivatives summed over all training cases.

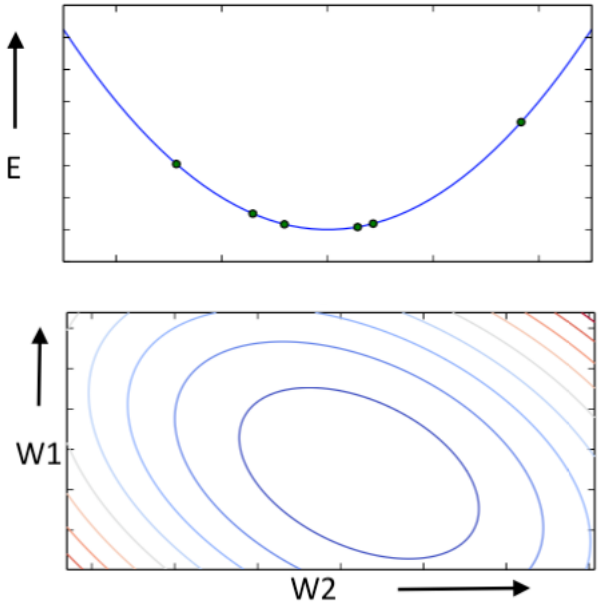
$$\Delta W_i = -\varepsilon \frac{\partial E}{\partial W_i} = \sum_n \varepsilon \, x_i^n (t^n - y^n)$$

SOME CONCLUSIONS FROM THE DONUT PROBLEM

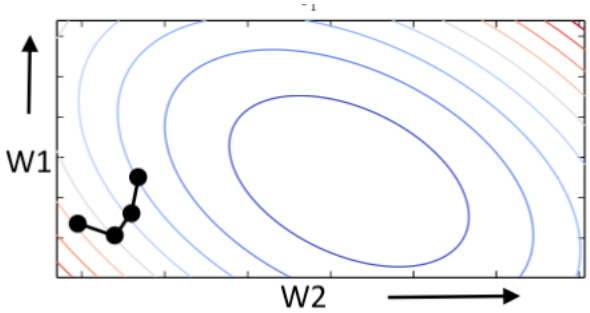
- Will the system eventually find the right answer?
- There may not be a perfect answer.
- But, by decreasing the learning rate we can approximate a good answer!
- Notice that classes that have lower number of examples are more difficult to learn. So there is a need of stratification (i.e. same number of training and testing examples per class).
- In cases that your input dimensions are highly correlated, learning could be very slow. (e.g. if you always get the same amount of chocolate and cherry donuts it would be very hard to find each of its prices).

THE ERROR SURFACE IN EXTENDED WEIGHT SPACE

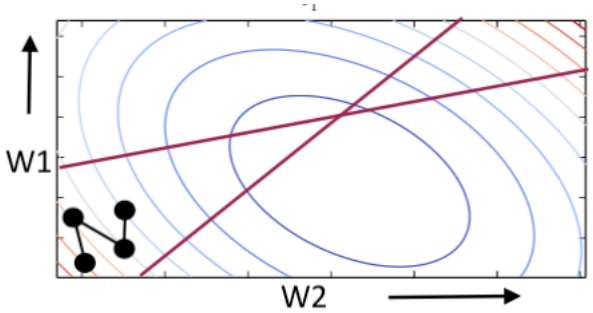
- The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error.
- For a linear neurone with a squared error, it is a quadratic bowl.
- Vertical cross-sections are parabolas.
- Horizontal cross-sections are ellipses.
- For multi-layer, non-linear nets the error surface is much more complicated (for a visualisation [link])



SINGLE EXAMPLE VS BATCH LEARNING



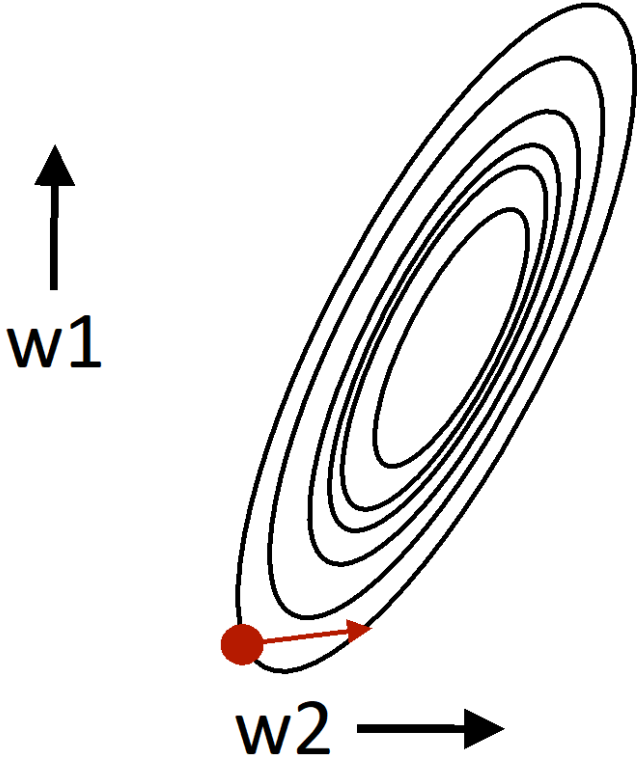
- The simplest batch learning does steepest descent on the error surface.
- This travels perpendicular to the contour lines.



- The simplest single example (i.e. online) learning ping-pongs around the direction of the steepest descent.

HOW LEARNING CAN BE SLOW

- In the case that the ellipse is very elongated, the direction of the steepest descent is almost perpendicular to the direction towards the minimum.
- The red gradient along the short axis of the ellipse and a small component along the long axis of the ellipse.
- This is far from optimal!



THE BACKPROPAGATION ALGORITHM

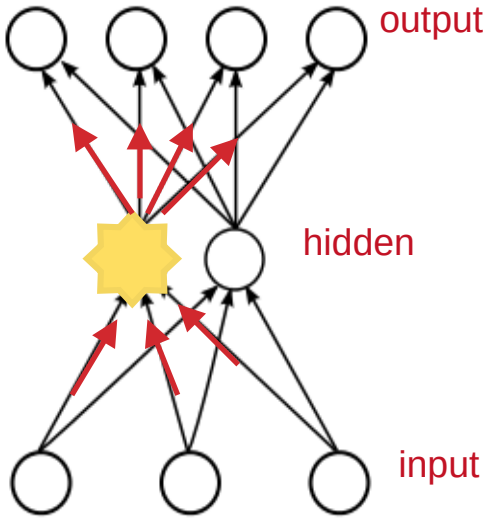
LEARNING WITH HIDDEN UNITS (A LOOK BACK)

- Networks without any hidden units are very limited in the input-output mappings that they can produce.
- Adding a layer of hand-coded features makes the much more powerful but there is still the problem of designing the features.
- Optimally, we would like to find good features without requiring insights into the task or repeated trial and error where we guess some features and see how well they work.
- We want to automate the process of designing features for a specific task and seeing how well they perform.

LEARNING BY PERMUTING WEIGHTS

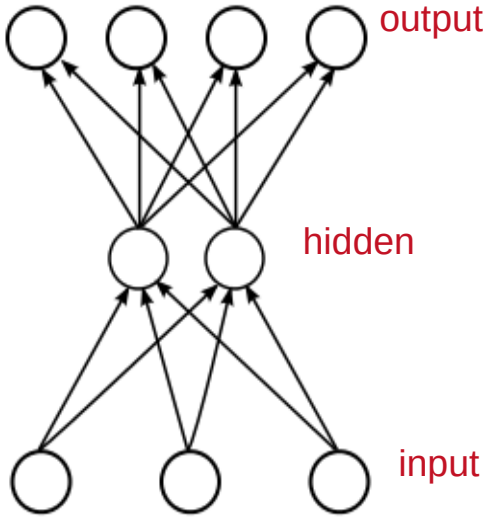
- This idea is based on evolution.
- Randomly permute one weight and find if it improves performance. If so, save the change.

- This can be considered as a form of reinforcement learning.
- It is highly inefficient: multiple forward passes need to be performed on a representative set just to change one weight.
- Towards the end of learning, large weight permutations are almost guaranteed to worsen performance, because the weights need to have the right relative values.



LEARNING BY PERMUTING WEIGHTS^2 (ALTERNATIVES)

- An alternative is to randomly permute all the weighs in parallel and correlate the performance gain to the weight changes.
- Not that better, because we need plenty of trails to find if all these changes do actually have an improvement (and cancel-out the noise produced).
- Another idea: randomly permute the activities of the hidden units.
- Once we know how we want a hidden activity to change on a given training case, we can compute how to change the weights.
- There are fewer activities than weights, but still inefficient.



MAIN IDEA OF BACKPROPAGATION

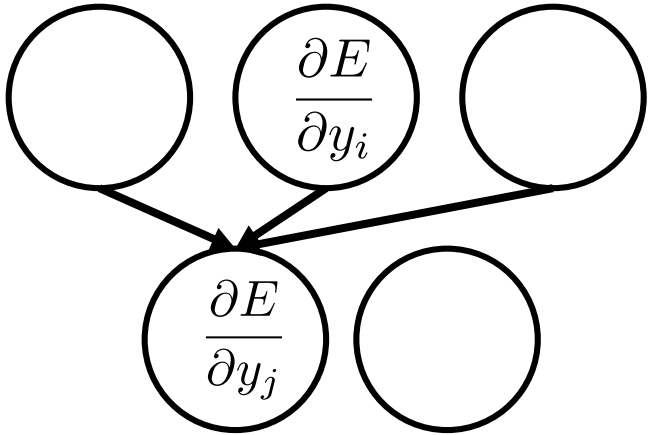
- We do not know what the hidden units are learning, but we can compute how fast the error changes as we change a hidden unit.
- Instead of using desired activities to train the hidden units, use error derivatives.
- Each error derivative affects many output units and can therefore have many separate effects on the error. These effects can be combined.
- We can compute error derivatives for all the hidden units efficiently at the same time.
- Once we have the error derivatives for the hidden units, its easy to get the error derivatives for the weights going into a hidden neurone.

THE IDEA BEHIND BACKPROPAGATION

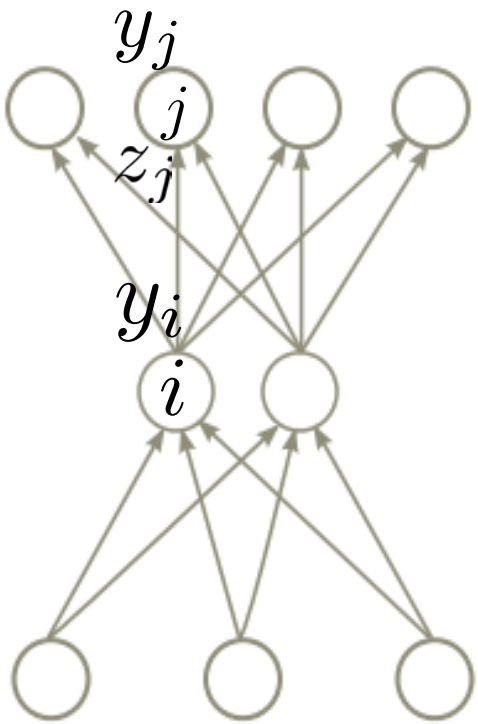
- First convert the discrepancy between each output and its target value into an error derivative.
- Then compute the error derivatives in each hidden layer from error derivatives in the layer above.
- Then use error derivatives with respect to the activities based on the incoming weights.

$$E = \frac{1}{2} \sum_{i \in output} (t_i - y_i)^2$$

$$\frac{\partial E}{\partial y_i} = -(t_i - y_i)$$



BACKPROPAGATION IN ONE SLIDE



$$\frac{\partial E}{\partial z_j} = \frac{\partial y_j}{\partial z_j}\frac{\partial E}{\partial y_j} = y_j(1 - y_j)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{\partial z_j}{\partial y_j}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$

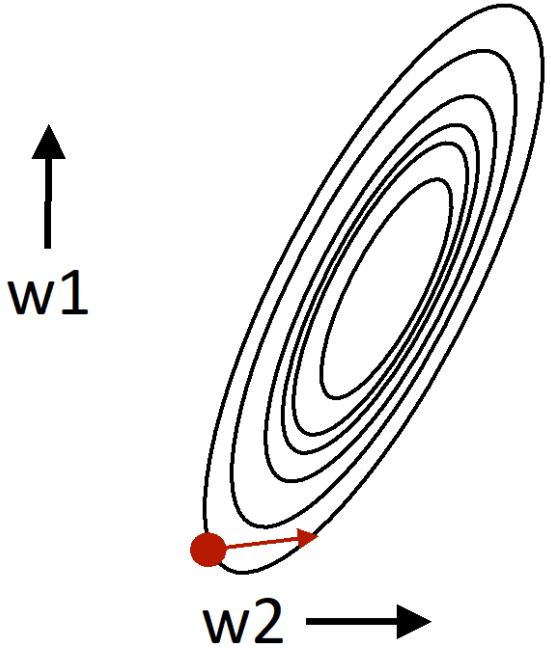
ERROR DERIVATIVES TO LEARN

- **Backpropagation is an efficient algorithm for computing the error derivative for every weight in the network based on a single training case.**
- For a fully specified learning procedure, we need to make a lot of other decisions about how to use these error derivatives:

- Optimisation: How to use the error derivatives in individual cases to discover a good set of weights?
- Generalisation: How to ensure that the learned weights for cases that we did not yet see?

OPTIMISATION ISSUES WITH ERROR DERIVATIVES

- How often weights are updated ?
- Online: after each training example.
- Full batch: after a full sweep through the training data.
- Mini-batch: after a small sample of the training data.
- How much to update ?
- Have a fixed learning rate.
- Adapt the global learning rate.
- Adapt the learning rate on each connection.



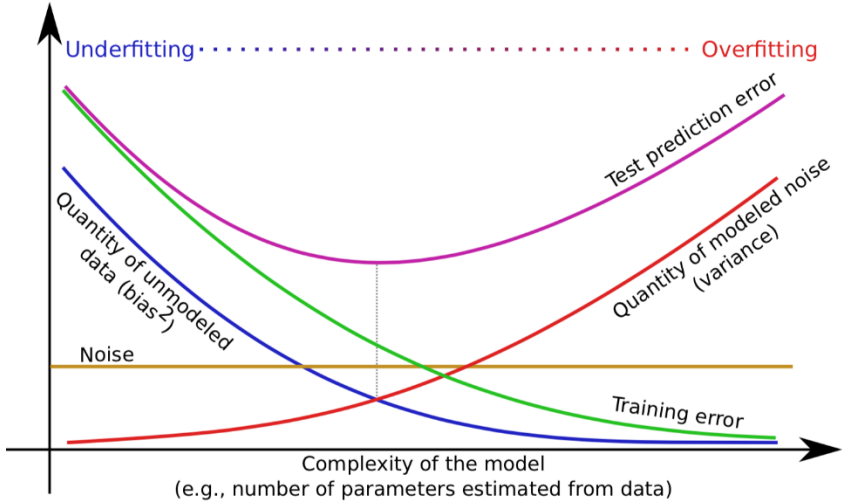
THE PROBLEMS OF OVERFITTING AND UNDERFITTING

- The training data contains information about the regularities in the mapping from input to output. But also contains noise:

- Target values may be unreliable.

- Sampling errors that corresponds to accidental regularities just because the particular cases were chosen.

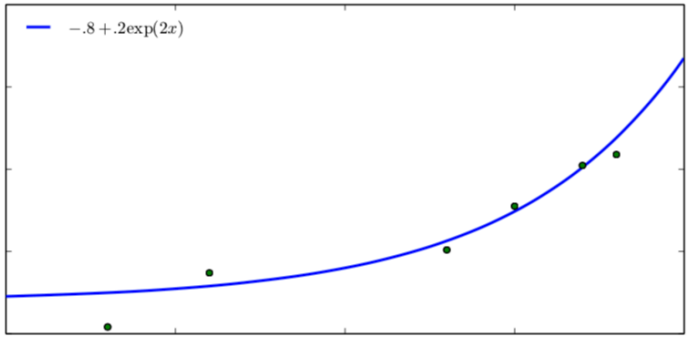
- When creating the model it cannot distinguish between real regularities and regularities caused by noise.



THE PROBLEMS OF OVERFITTING AND UNDERFITTING[2]

Consider the blue line to be our training data.

We will try and build a model that approximates, the best way possible, the **ground truth** from the observations (data points).

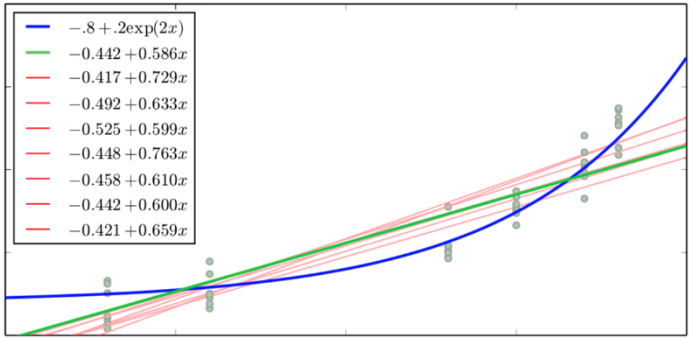


THE PROBLEMS OF OVERFITTING AND UNDERFITTING[3]

Fitting a linear function

The method can generalise very well -> So the **errors due to overfitting are decreased**

But the distance from the data points is still large so **errors due to underfitting are increased**

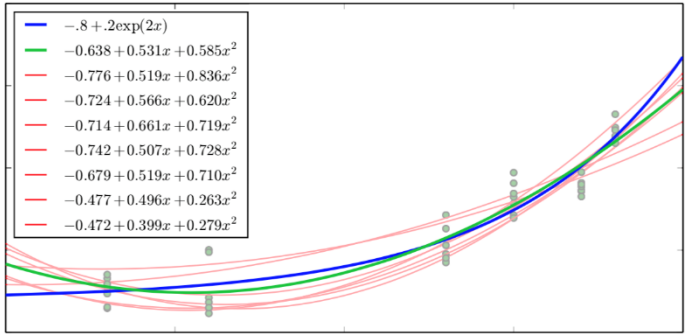


THE PROBLEMS OF OVERFITTING AND UNDERFITTING[4]

Fitting a quadratic polynomial

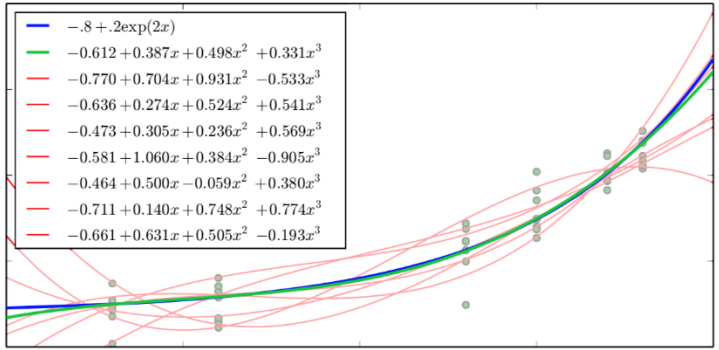
The method can generalise less-> So the **errors due to overfitting are worsen**

But the mapping of points is better, so **errors due to underfitting are decreased**

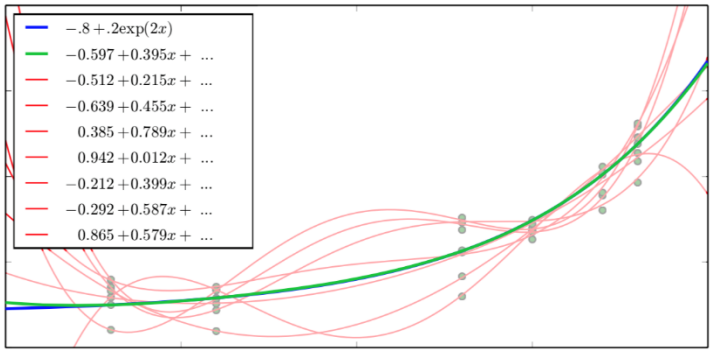


THE PROBLEMS OF OVERFITTING AND UNDERFITTING[5]

What about a cubic polynomial?



Or a fourth order polynomial?



So we can conclude that increasing the model complexity decreases the bias (i.e. the error associated with underfitting) and increases variance (i.e. error associated with overfitting).

It would not be surprising if a very complicated model can only fit a very small amount of data.

WAYS OF REDUCING OVERFITTING

- A large body of research has been done:

- Weight-decay [link]
- Early-stopping [link]
- Learning rate warm-ups [link]
- Dropout [link]
- Batch normalisation [link]

NEURAL NETWORKS IN COMPUTER VISION

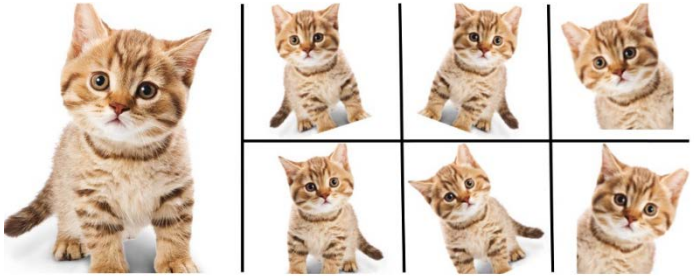
THINGS THAT MAKE VISION DIFFICULT

- Segmentation: Real scenes are cluttered with other objects.
- It is very difficult to tell which pieces go together as part of an object.
- Parts of an object can be hidden behind other objects (commonly referred as occlusion).
- Lighting: The intensities of the pixels are determined as much by the lighting as by the objects.
- Deformation: Objects can deform in a variety of non-affine ways.
- e.g. a hand written seven can have a line in the middle section.
- Affordances: Object classes may also be defined by how they are used.
- e.g. chairs are used for sitting but they have varieties on their appearances.

THINGS THAT MAKE VISION DIFFICULT[2]

- Viewpoint: Changes in viewpoint cause changes in images that standard learning methods cannot cope with.

- Information hops between input dimensions (i.e. pixels)



INVARIANT FEATURES

- Extract a large number of features that are invariant under transformations.
- E.g. pair of roughly parallel lines with a red dot between
- With enough invariant features, there is only one way to assemble them into an object.
- We do not need to represent the relationships between features directly because they are captured by other features.
- But for recognition, we must avoid forming features from parts of different objects.

TOWARDS VIEWPOINT INVARIANCE

- Humans are very good at viewpoint invariance that it is very hard to appreciate how difficult it is.
- It is considered as one of the main difficulties in making computers perceive.
- We still don’t have generally accepted solutions.
- There are many different approaches:
- Use redundant invariant features.
- Put a box around the object and use normalised pixels.
- Use Convolutional Neural Networks (next lecture).

SUPPLEMENTARY MATERIAL

- Part II, Chapter 6 from Ian Goodfellow's *Deep Learning* (pages 166 to 224)
- Part 1, Chapter 3 from Francois Chollet's *Deep Learning with Python* (pages 56 to 92)