COMP9417 - Week8

Neural Learning

COMP9417 Machine Learning and Data Mining

Term 2, 2022

Acknowledgements

Material derived from slides for the book
"Elements of Statistical Learning (2nd Ed.)" by T. Hastie,
R. Tibshirani & J. Friedman. Springer (2009)
http://statweb.stanford.edu/~tibs/ElemStatLearn/

Material derived from slides for the book
"Machine Learning: A Probabilistic Perspective" by P. Murphy
MIT Press (2012)
http://www.cs.ubc.ca/~murphyk/MLbook

Material derived from slides for the book
"Machine Learning" by P. Flach
Cambridge University Press (2012)
http://cs.bris.ac.uk/~flach/mlbook

Material derived from slides for the book
"Bayesian Reasoning and Machine Learning" by D. Barber
Cambridge University Press (2012)
http://www.cs.ucl.ac.uk/staff/d.barber/brml

Material derived from slides for the book
"Machine Learning" by T. Mitchell
McGraw-Hill (1997)
http://www-2.cs.cmu.edu/~tom/mlbook.html

Material derived from slides for the course
"Machine Learning" by A. Srinivasan
BITS Pilani, Goa, India (2016)

Aims

This lecture will enable you to describe and reproduce machine learning approaches to the problem of neural (network) learning. Following it you should be able to:

- describe Perceptrons and how to train them
- relate neural learning to optimization in machine learning
- outline the problem of neural learning
- derive the method of gradient descent for linear models
- describe the problem of non-linear models with neural networks
- outline the method of back-propagation training of a multi-layer perceptron neural network
- describe the application of neural learning for classification
- describe some issues arising when training deep networks

Introduction

- Neural Learning based on Artificial Neural Networks (ANNs)
  - "inspired by" Biological Neural Networks (BNNs) ...
- but structures and learning methods are different
  - $\text{ANNs} \neq \text{BNNs}$
- ANNs based on simple logical model[1] of biological neuron
  - the "Perceptron"
- ANNs are the basis of Deep Learning (DL)
- entire course on Neural Networks/Deep Learning (COMP9444)
- we focus on neural learning in relation to methods in ML
- $\text{DL} \neq \text{ML}$
- $\text{DL} \neq \text{AI}$!

[1] See: McCulloch and Pitts (1943).

Artificial Neural Networks

Main ideas we will cover:

- Logical threshold units – i.e., Perceptrons
- Loss function for a Perceptron (review)
- Convergence theorem for a Perceptron
- Loss function for a Linear Unit (unthresholded Perceptron)
- Gradient descent for a Linear Unit
- Multilayer networks
- Backpropagation: gradient descent for multilayer networks

Connectionist Models

? Cognition problems?

Consider humans:

- Neuron switching time $\approx .001$ second
- Number of neurons $\approx 10^{10}$
- Connections per neuron $\approx 10^{4-5}$
- Scene recognition time $\approx .1$ second
- 100 inference steps doesn't seem like enough

$\rightarrow$ much parallel computation

✓ Perception!
- vision
- audition
- other sensory inputs

✗ Computation problem of the brain?

Connectionist Models

Properties of artificial neural nets (ANNs):

- Many neuron-like threshold switching units
- Many weighted interconnections among units
- Highly parallel, distributed process of learning
- Emphasis on tuning weights automatically
- ANNs learn *distributed representions* of target function

When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g., raw sensor input)
- Output can be discrete or real-valued
- Output can be a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

pixels: 0-255

- images
- speech (audio)
- video
- ...

- can be a problem!

Applications:

- Speech recognition (now the standard method)
- Image classification (also now the standard method)
- many others ...

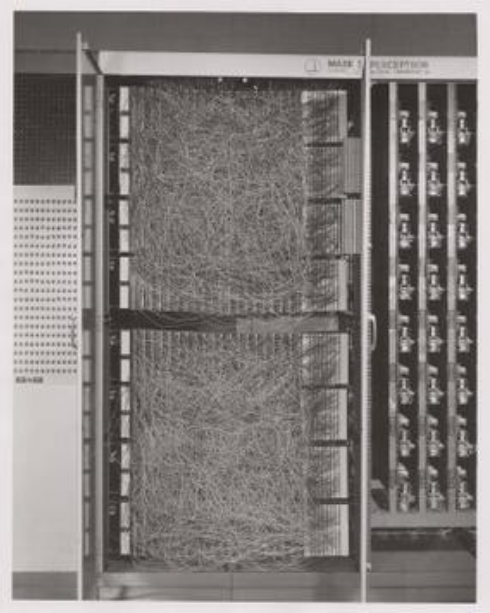
AlphaGo
Alpha zero
Alpha Fold
GPT3

Perceptron

A linear classifier that can achieve perfect separation on linearly separable data is the *perceptron*, originally proposed as a simple *neural network* by F. Rosenblatt in the late 1950s.

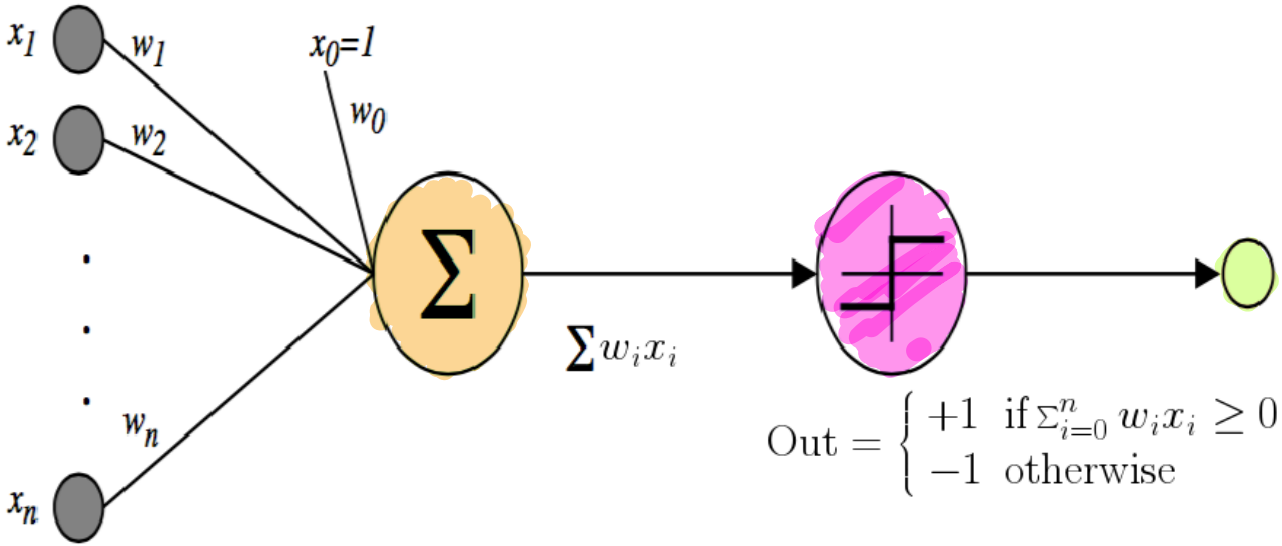
Perceptron

Originally implemented in software (based on the McCulloch-Pitts neuron from the 1940s), then in hardware as a 20x20 visual sensor array with potentiometers for adaptive weights.



Source `http://en.wikipedia.org/w/index.php?curid=47541432`

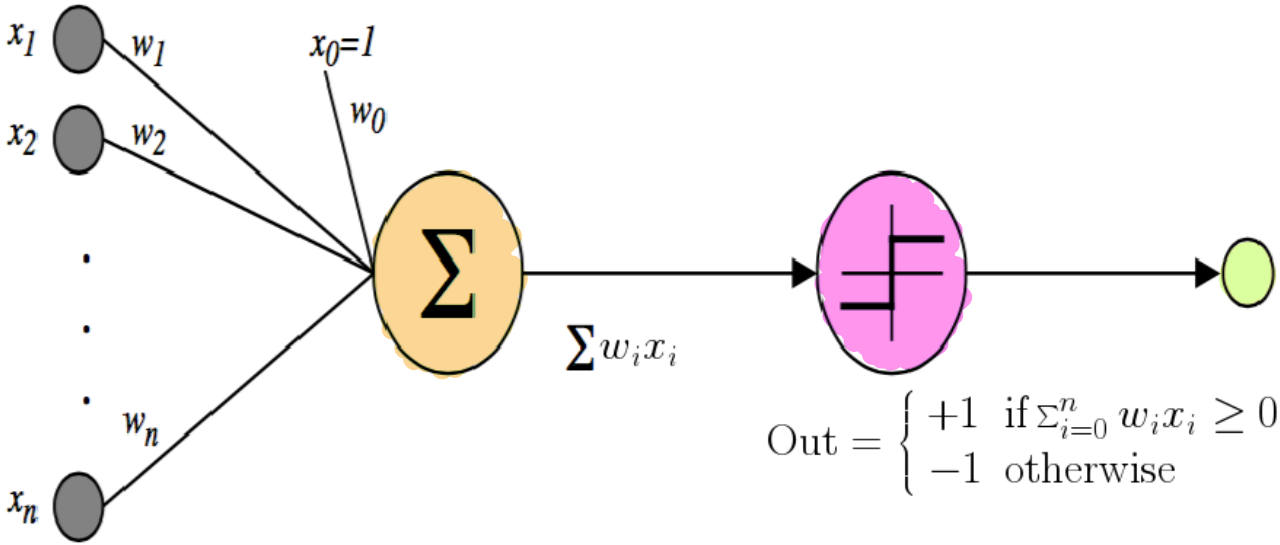
Perceptron



Output o is thresholded sum of products of inputs and their weights:

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

Perceptron



Or in vector notation:

$$o(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} > 0 \\ -1 & \text{otherwise.} \end{cases}$$

Perceptron training algorithm

Algorithm Perceptron(D, η) // perceptron training for linear classification

Input: labelled training data D in homogeneous coordinates; learning rate η.

Output: weight vector $\mathbf{w}$ defining classifier $\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x})$.

```
1  w ← 0 // Other initialisations of the weight vector are possible
2  converged ← false
3  while converged = false do
4      converged ← true
5      for i = 1 to |D| do
6          if y_i w · x_i ≤ 0 then          // i.e., ŷ_i ≠ y_i
7              w ← w + η y_i x_i
8              converged ← false  // We changed w so haven't converged yet
9          end
10     end
11 end
```

$y_i \in \{-1, +1\}$

Will converge for linearly separable D, under conditions

Perceptron Convergence

Perceptron training will converge (under some mild assumptions) for linearly separable classification problems

A labelled data set is linearly separable if there is a linear decision boundary that separates the classes

Perceptron Convergence

Assume:

$x_i \in \mathbb{R}^d$

Dataset $D = \{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$

linearly-separable

At least one example in D is labelled $+1$, and one is labelled -1.

$R = \max_i ||\mathbf{x}_i||_2$

Perceptron := $w \cdot x_i$

What w?

A weight vector $\mathbf{w}^*$ exists s.t. $||\mathbf{w}^*||_2 = 1$ and $\forall i \; y_i \mathbf{w}^* \cdot \mathbf{x}_i \geq \gamma$

data term

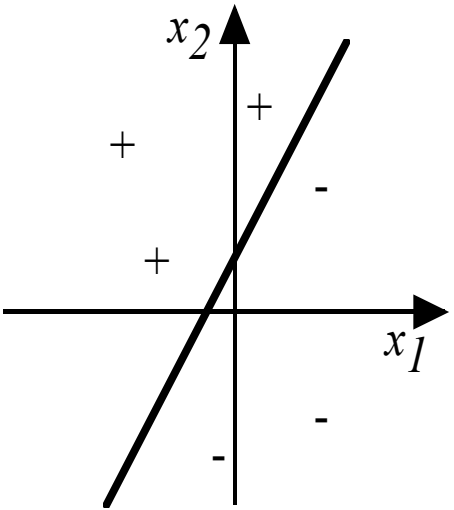
Perceptron Convergence Theorem (Novikoff, 1962)

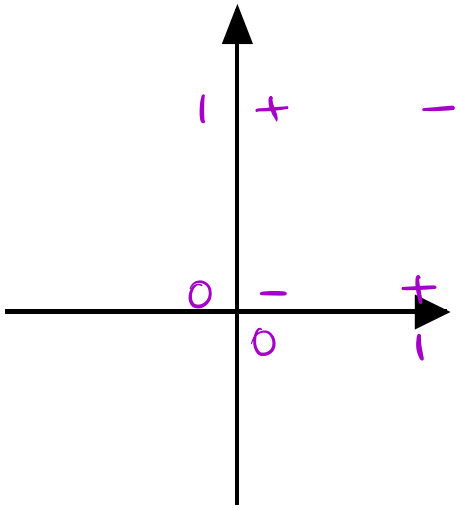
The number of mistakes made by the perceptron is at most $(\frac{R}{\gamma})^2$.

generally firm

γ is typically referred to as the "margin".

Decision Surface of a Perceptron





Represents some useful functions

- What weights represent $o(x_1, x_2) = AND(x_1, x_2)$?
- What weights represent $o(x_1, x_2) = XOR(x_1, x_2)$?

Decision Surface of a Perceptron

Unfortunately, as a linear classifier perceptrons are limited in expressive power

So some functions not representable

- e.g., Boolean function XOR is not linearly separable

For non-linearly separable data we'll need something else

Fortunately, with fairly minor modifications many perceptrons can be combined together to form one model

- *multilayer perceptrons*, the classic "neural network"

Optimization

Minimise error (loss) w.r.t parameters

A general iterative algorithm [2] to optimise some function f:

1. start with initial point $\mathbf{x} = \mathbf{x}_0$ e.g., parameters $\vec{w}$
2. select a search direction $\mathbf{g}$, usually to decrease $f(\mathbf{x})$
3. select a step length η
4. set $\mathbf{s} = \eta\mathbf{g}$
5. set $\mathbf{x} = \mathbf{x} + \mathbf{s}$
6. go to step 2, unless convergence criteria are met

For example, could minimize a real-valued function $f : \mathbb{R}^n \rightarrow \mathbb{R}$

Note: convergence criteria will be problem-specific.

[2] See: Ripley (1996).

Optimization

Usually, we would like the optimization algorithm to quickly reach an answer that is close to being the right one.

- typically, need to minimize a function
  - e.g., error or *loss*
  - optimization is known as *gradient descent* or *steepest descent*
- sometimes, need to maximize a function
  - e.g., probability or *likelihood*
  - optimization is known as *gradient ascent* or *steepest ascent*

Requires function to be *differentiable*.

Perceptron learning

Key idea:

Learning is "finding a good set of weights"

Perceptron learning is simply an iterative weight-update scheme:

$$w_i \leftarrow w_i + \Delta w_i$$

where the component-wise weight update Δw_i depends only on *misclassified* examples and is modulated by a "smoothing" parameter η typically referred to as the "learning rate".

Perceptron learning

Let

$$\Delta w_i = \eta(t - o)x_i$$

error ↓

correct: $t=o \Rightarrow t-o=\emptyset$

no weight update

incorrect: $t \neq o \Rightarrow$

(↓) FN: $\Delta w_i = 1 * w_i$ (↑)

(↑) FP: $\Delta w_i = -1 * w_i$ (↓)

Where:

- $t = c(\mathbf{x})$ is target value in $\{0, 1\}$
- o is perceptron output in $\{0, 1\}$
- η is a small constant called *learning rate*
  - learning rate is a positive number, typically between 0 and 1
  - to simplify things we sometimes assume $\eta = 1$
  - but in practice usually set at less than 0.2, e.g., 0.1
  - η can be varied during learning

Unfortunately, the output o is discontinuous, so not differentiable.

Gradient Descent

Separate thresholding from model:

Consider *linear unit*, where

$$o = w_0 + w_1 x_1 + \cdots + w_n x_n$$

Many possible loss functions!

Let's learn w_i's that minimize the squared error (loss function)

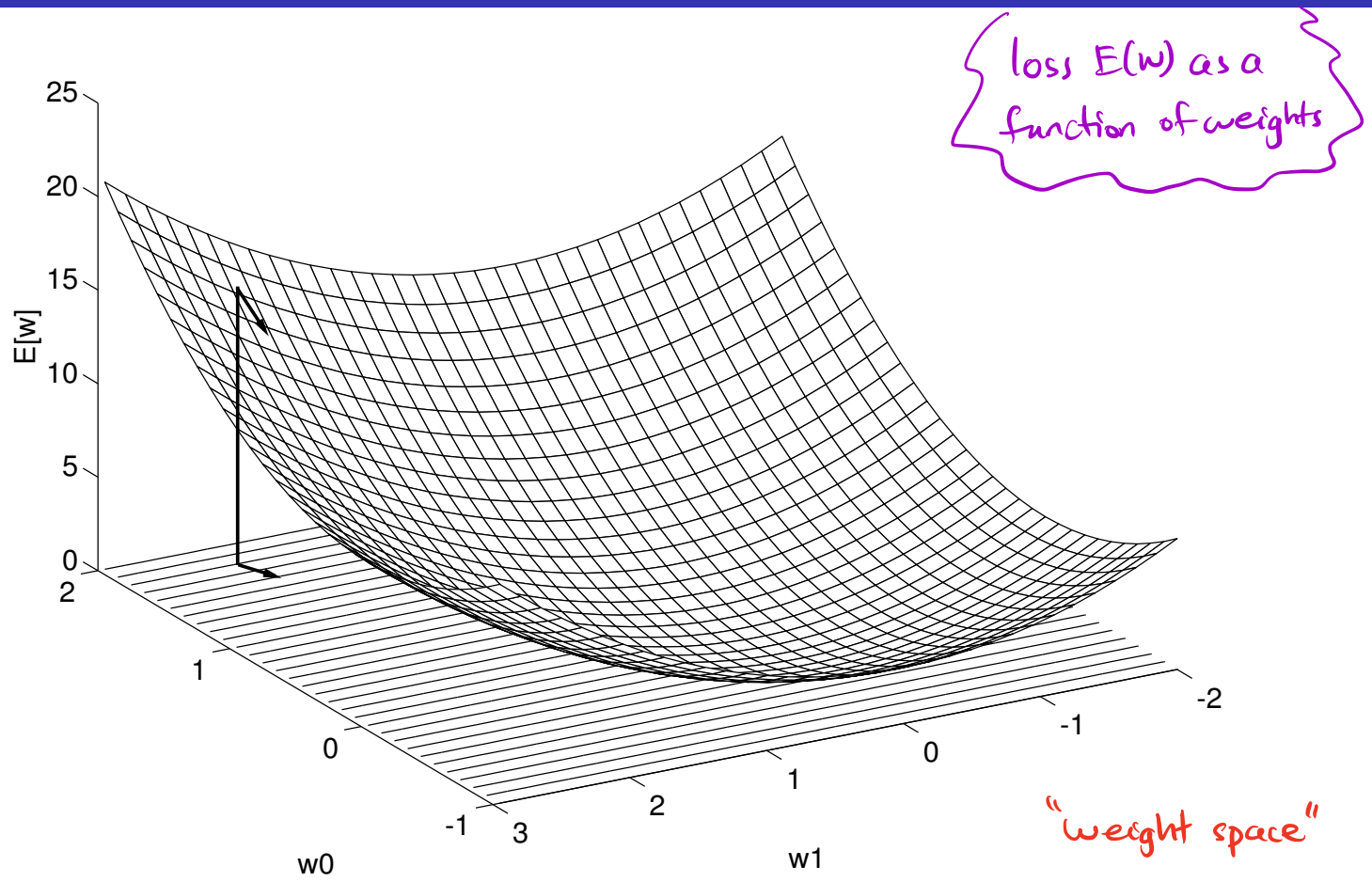
$$E[\mathbf{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

target label

predicted label

Where D is set of training examples

Gradient Descent



Gradient Descent

Gradient: derivative of E wrt each component of weight vector $\mathbf{w}$

$$\nabla E[\mathbf{w}] \equiv \left[\frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \cdots \frac{\partial E}{\partial w_n}\right]$$

Gradient vector gives direction of *steepest increase* in error E

Negative of the gradient, i.e., *steepest decrease*, is what we want

Training rule:

$$\Delta \mathbf{w} = -\eta \nabla E[\mathbf{w}]$$

i.e., component-wise

$$\Delta w_i = -\eta \frac{\partial E}{\partial w_i}$$

Derivation of Gradient Descent for Linear Unit

$$-\left[w\frac{\partial}{\partial w_i}x_d + x_d\frac{\partial}{\partial w_i}w\right]$$

$$\begin{aligned}
\frac{\partial E}{\partial w_i} &= \frac{\partial}{\partial w_i}\frac{1}{2}\sum_d (t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d \frac{\partial}{\partial w_i}(t_d - o_d)^2 \\
&= \frac{1}{2}\sum_d 2(t_d - o_d)\frac{\partial}{\partial w_i}(t_d - o_d) \\
&= \sum_d (t_d - o_d)\frac{\partial}{\partial w_i}(t_d - \mathbf{w}\cdot\mathbf{x}_d) \\
\frac{\partial E}{\partial w_i} &= \sum_d (t_d - o_d)(-x_{i,d})
\end{aligned}$$

Training a Linear Unit by Gradient Descent

GRADIENT-DESCENT($training\_examples, \eta$)

Each training example is a pair $\langle \mathbf{x}, t \rangle$, where $\mathbf{x}$ is the vector of input values, and t is the target output value. η is the learning rate (e.g., .05).

Initialize each w_i to some small random value

Until the termination condition is met, Do
- Initialize each Δw_i to zero
- For each $\langle \mathbf{x}, t \rangle$ in $training\_examples$, Do "one epoch"
  - Input the instance $\mathbf{x}$ to the unit and compute the output o
  - For each linear unit weight w_i
    $$\Delta w_i \leftarrow \Delta w_i + \eta(t - o)x_i$$
- For each linear unit weight w_i
  $$w_i \leftarrow w_i + \Delta w_i$$

Training Perceptron vs. Linear unit

Perceptron training rule guaranteed to succeed if (Novicoff's Thm)

- Training examples are linearly separable
- Sufficiently small learning rate η

Linear unit training rule uses gradient descent

- Guaranteed to converge to hypothesis with minimum squared error
  - optimization
- Given sufficiently small learning rate η
- Even when training data contains noise
- Even when training data not separable by ~~H~~ model: $\vec{w}$

Incremental (Stochastic) Gradient Descent

Batch mode Gradient Descent:

Do until termination condition is satisfied

- Compute the gradient $\nabla E_D[\mathbf{w}]$
- $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_D[\mathbf{w}]$

Incremental mode (Stochastic) Gradient Descent:

Do until satisfied

- For each training example d in D
  - Compute the gradient $\nabla E_d[\mathbf{w}]$
  - $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_d[\mathbf{w}]$

"mini-batch" subset $d \in D$

SGD "good approximation" of batch

Incremental (Stochastic) Gradient Descent

Batch:

$$E_D[\mathbf{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$

Incremental:

$$E_d[\mathbf{w}] \equiv \frac{1}{2} (t_d - o_d)^2$$

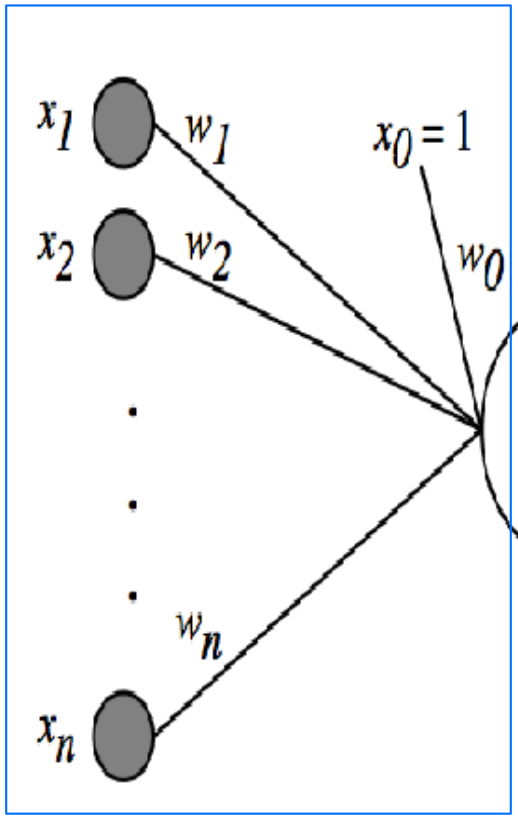
Incremental or *Stochastic Gradient Descent* (SGD) can approximate *Batch Gradient Descent* arbitrarily closely, if η made small enough

Very useful for training large networks (mini-batches), or online learning from data streams

Stochastic implies examples should be selected at random

Sigmoid Unit

"activation function"



Σ

net = $\Sigma_{i=0}^{n} w_i x_i$

$$\text{Out} = \frac{1}{1+e^{-(\Sigma_{i=0}^{n} w_i x_i)}}$$

Sigmoid Unit

Same as a perceptron except that the step function has been replaced by a smoothed version, a sigmoid function.

Note: in practice, particularly for deep networks, sigmoid functions are much less common than other non-linear activation functions that are easier to train.

For example, the default activation function for deep networks is the Rectified Linear Unit (ReLU) or variants.

However, sigmoids are mathematically convenient.

Sigmoid Unit

Why use the sigmoid function $\sigma(x)$?

$$\frac{1}{1+e^{-x}}$$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1-\sigma(x))$

We can derive gradient descent rules to train

- One sigmoid unit
- *Multi-layer networks* of sigmoid units $\rightarrow$ Backpropagation

We will use this to derive Backpropagation to train a *Multi-layer Perceptron* (MLP)

Notation:

- x_{ji} = the ith input to unit j
- w_{ji} = weight associated with ith input to unit j
- $net_j = \sum_i w_{ji} x_{ji}$ = (weighted sum of inputs for unit j)
- o_j = output computed by unit j
- t_j = the target output for unit j
- σ = the sigmoid function
- $outputs$ = the set of units in the final layer of the network
- $Downstream(j)$ = the set of units whose immediate inputs include the output of unit j

Derivation of SGD Training for MLP

Stochastic gradient descent means we need to descend the gradient of the error E_d with respect to each training example $d \in D$.

Update each weight w_{ji} by adding to it Δw_{ji}, where

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}}$$

E_d is error on example d, summed over all output units in the network

$$E_d(\mathbf{w}) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

Derivation of SGD Training for MLP

Weight w_{ji} can influence the rest of the network only through net_j

Apply the chain rule:

$$\begin{aligned} \frac{\partial E_d}{\partial w_{ji}} &= \frac{\partial E_d}{\partial net_j} \frac{\partial net_j}{\partial w_{ji}} \\ &= \frac{\partial E_d}{\partial net_j} x_{ji} \end{aligned}$$

What about $\frac{\partial E_d}{\partial net_j}$? Two cases to consider, where:

- unit j is an output node of the network
- unit j is an internal node ("hidden unit") of the network

Case 1: Training rule for output unit weights

net_j can influence the network only through o_j, so apply chain rule again:

$$\frac{\partial E_d}{\partial net_j} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial net_j}$$

Taking the first term and applying the chain rule:

$$\frac{\partial E_d}{\partial o_j} = \frac{\partial}{\partial o_j} \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2 \qquad \frac{\partial}{\partial o_j} \frac{1}{2}(t_j - o_j)^2$$

$$= -(t_j - o_j) \qquad = \frac{1}{2} 2(t_j - o_j) \frac{\partial}{\partial o_j}(t_j - o_j)$$

$$= -(t_j - o_j)$$

because we only need to consider output $k = j$.

Case 1: Training rule for output unit weights

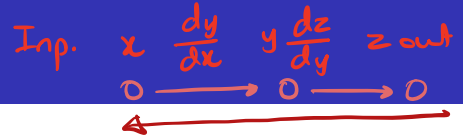
For the second term, note that $o_j = \sigma(net_j)$, and recall that the derivative $\frac{\partial o_j}{\partial net_j}$ is the derivative of the sigmoid function, that is, $\sigma(net_j)(1 - \sigma(net_j)$, so

$$\begin{aligned} \frac{\partial o_j}{\partial net_j} &= \frac{\partial \sigma(net_j)}{\partial net_j} \\ &= o_j(1 - o_j) \end{aligned}$$

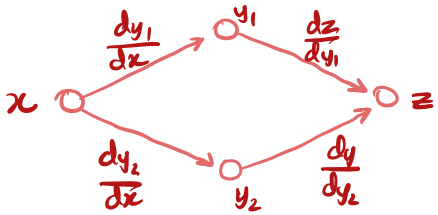
Substituting the results for both terms into the original expression:

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

Case 1: Training rule for output unit weights



Multiple parts: $\frac{dz}{dx} = \frac{dz}{dy_1}\frac{dy_1}{dx} + \frac{dz}{dy_2}\frac{dy_2}{dx}$



We can now implement weight update as:

$$\Delta w_{ji} = -\eta \frac{\partial E_d}{\partial w_{ji}} = \eta\,(t_j - o_j)\,o_j(1 - o_j)\,x_{ji}$$

We will use the notation δ_i to denote the quantity $-\frac{\partial E_d}{\partial net_i}$ for unit i.

Case 2: Training rule for hidden unit weights

Internal unit j can only influence the output by *all* paths through $Downstream(j)$, i.e., all nodes directly connected to net_j.

$$\begin{aligned}
\frac{\partial E_d}{\partial net_j} &= \sum_{k \in Downstream(j)} \frac{\partial E_d}{\partial net_k} \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \frac{\partial net_k}{\partial o_j} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \frac{\partial o_j}{\partial net_j} \\
&= \sum_{k \in Downstream(j)} -\delta_k \, w_{kj} \, o_j(1 - o_j)
\end{aligned}$$

Case 2: Training rule for hidden unit weights

Rearranging terms and using δ_j to denote $-\frac{\partial E_d}{\partial net_j}$

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

and the weight update

$$\Delta w_{ji} = \eta \, \delta_j \, x_{ji}$$

Backpropagation Algorithm

SGD training of MLP

Initialize all weights to small random numbers.

Until termination condition satisfied, Do

For each training example, Do

Input training example $\langle \mathbf{x}, \mathbf{t} \rangle$ to the network and compute the network outputs

Fprop

For each output unit k

$$\delta_k \leftarrow o_k(1 - o_k)(t_k - o_k)$$

For each hidden unit h *

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

Update each network weight w_{ji}

$$w_{ji} \leftarrow w_{ji} + \Delta w_{ji}$$

where

$$\Delta w_{ji} = \eta \delta_j x_{ji}$$

* assumes 1 hidden layer MLP

Bprop

More on Backpropagation

Regularisation e.g.,
add penalty on weights
$\|w\|^2$
to the loss

A solution for learning highly complex models . . .

- Gradient descent over entire *network* weight vector
- Easily generalized to arbitrary directed graphs
- Can learn probabilistic models by maximising likelihood

Minimizes error over *all* training examples

- Training can take thousands of iterations $\rightarrow$ slow!
- Using network after training is very fast

More on Backpropagation

Will converge to a local, not necessarily global, error minimum

- May be many such local minima
- In practice, often works well (can run multiple times)
- Often include weight *momentum* α

$$\Delta w_{ji}(n) = \eta \delta_j x_{ji} + \alpha \Delta w_{ji}(n-1)$$

"adaptive"
Adagrad
Adam
...

- Stochastic gradient descent using "mini-batches"

Nature of convergence

- Initialize weights near zero
- Therefore, initial networks near-linear
- Increasingly non-linear functions possible as training progresses

More on Backpropagation

Models can be very complex

- Will network generalize well to subsequent examples?
  - may *underfit* by stopping too soon
  - may *overfit* ...

Many ways to regularize network, making it less likely to overfit

- Add term to error that increases with magnitude of weight vector

$$E(\mathbf{w}) \equiv \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (t_{kd} - o_{kd})^2 + \gamma \sum_{i,j} w_{ji}^2$$

- Other ways to penalize large weights, e.g., weight decay
- Using "tied" or shared set of weights, e.g., by setting all weights to their mean after computing the weight updates
- Many other ways ...

Expressive Capabilities of ANNs

"Deeper is better"! (usually)

Boolean functions:

- Every Boolean function can be represented by network with single hidden layer
- but 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]
- Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].

Being able to approximate any function is one thing, being able to *learn* it is another ...

How complex should the model be ?

With four parameters I can fit an elephant, and with five I can make him wiggle his trunk.

John von Neumann

Neural networks for classification

Sigmoid unit computes output $o(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x})$

Output ranges from 0 to 1

Example: binary classification

$$o(\mathbf{x}) = \begin{cases} \text{predict class } 1 & \text{if } o(\mathbf{x}) \geq 0.5 \\ \text{predict class } 0 & \text{otherwise.} \end{cases}$$

Questions:

- what error (loss) function should be used ?
- how can we train such a classifier ?

Neural networks for classification

Minimizing square error (as before) does not work so well for classification

If we take the output $o(\mathbf{x})$ as the *probability* of the class of $\mathbf{x}$ being 1, the preferred loss function is the *cross-entropy*

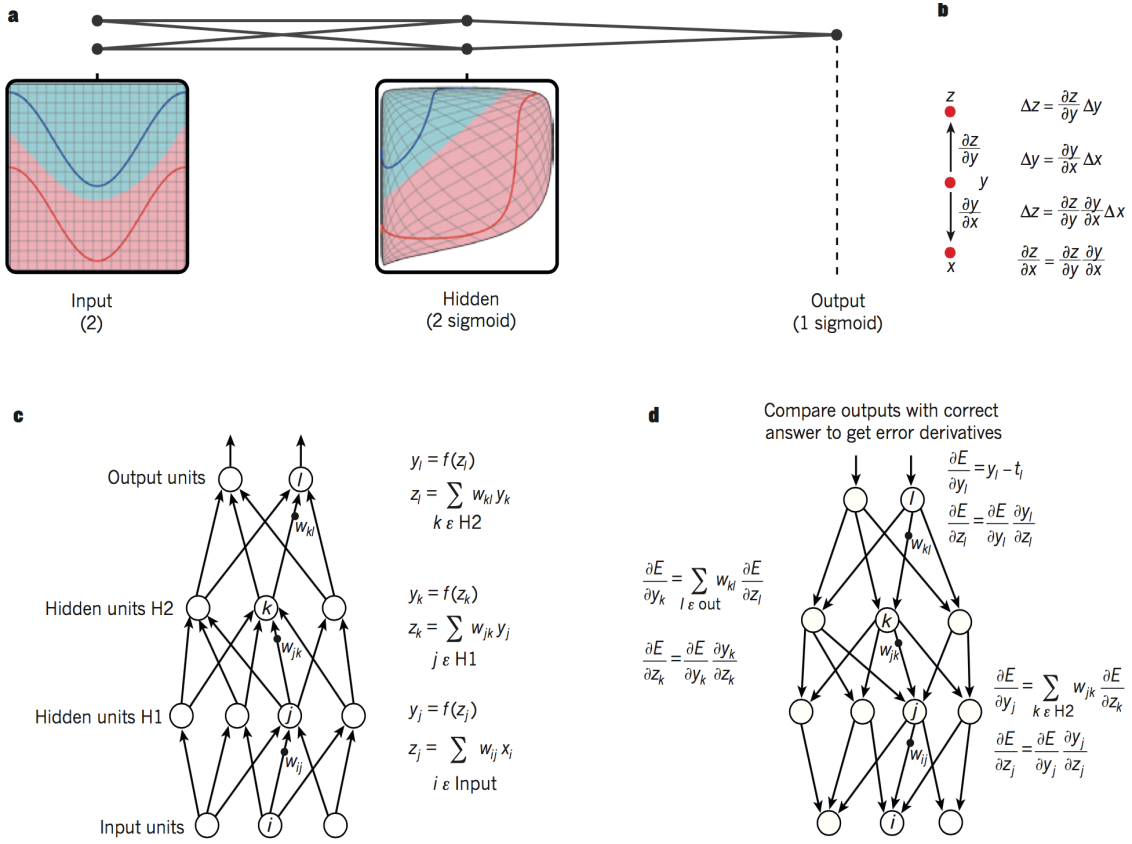
$$-\sum_{d \in D} t_d \log o_d + (1 - t_d) \log (1 - o_d)$$

where:

$t_d \in \{0, 1\}$ is the class label for training example d, and o_d is the output of the sigmoid unit, interpreted as the probability of the class of training example d being 1.

To train sigmoid units for classification using this setup, can use *gradient ascent* with a similar weight update rule as that used to train neural networks by gradient descent – this will yield the *maximum likelihood* solution.

Deep Learning



Y. Lecun et al. (2015) Nature (521) 436–444.

Deep Learning

Deep learning is a vast area that has exploded in the last 15 years.

Beyond scope of this course to cover in detail.

See: "Deep Learning" I. Goodfellow *et al.* (2017) – there is an online copy freely available.

Course COMP9444 Neural Networks / Deep Learning

Deep Learning

Question: How much of what we have seen carries over to deep networks ?

Answer: Most of the basic concepts.

We mention some important issues that differ in deep networks.

Deep Learning: Architectures

Most successful deep networks *do not* use the fully connected network architecture we outlined above.

Instead, they use more specialised architectures for the application of interest (*inductive bias*).

Example: Convolutional neural nets (CNNs) have an alternating layer-wise architecture inspired by the brain's visual cortex. Works well for image processing tasks, but also for applications like text processing.

Example: Long short-term memory (LSTM) networks have recurrent network structure designed to capture long-range dependencies in *sequential* data, as found, e.g., in natural language (although now often superseded by *transformer* architectures).

Example: Autoencoders are are kind of *unsupervised* learning method. They learn a mapping from input examples to *the same examples as output* via a compressed (lower dimension) hidden layer, or layers.

Deep Learning: Activation Functions

Problem: in very large networks, sigmoid activation functions can *saturate*, i.e., can be driven close to 0 or 1 and then the gradient becomes almost 0 – effectively halts updates and hence learning for those units.

Solution: use activation functions that are non-saturating., e.g., "Rectified Linear Unit" or ReLu, defined as $f(x) = \max(0, x)$.

Problem: sigmoid activation functions are not zero-centred, which can cause gradients and hence weight updates become "non-smooth".

Solution: use zero-centred activation function, e.g., $tanh$, with range $[-1, +1]$. Note that $tanh$ is essentially a re-scaled sigmoid.

Derivative of a ReLu is simply

$$\frac{\partial f}{\partial x} = \begin{cases} 0 & \text{if } x \leq 0 \\ 1 & \text{otherwise.} \end{cases}$$

Deep Learning: Regularization

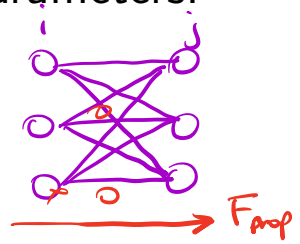
"fully connected networks": MLP

Problem ① overfitting ; ② efficiency

Deep networks can have millions or billions or parameters.

Hard to train, prone to overfit.

What techniques can help ?



Example: dropout

- for each unit u in the network, with probability p, "drop" it, i.e., ignore it and its adjacent edges during training
- this will simplify the network and prevent overfitting
- can take longer to converge
- but will be quicker to update on each epoch
- also forces exploration of different sub-networks formed by removing p of the units on any training run

"dropout $\approx$ l_2 Regularisation"

Back-propagation and computational graphs

See next tute/lab for computational graphs & implementations

Most deep learning models do not rely on manual derivation of training rules as we did, but rely on *automatic differentiation* based on computational graphs. See, e.g., Strang (2019).

Summary

- ANNs since 1940s; popular in 1980s, 1990s; recently a revival

  Complex function fitting. Generalise core techniques from machine learning and statistics based on linear models for regression and classification.

  Learning is typically stochastic gradient descent. Networks are too complex to fit otherwise.

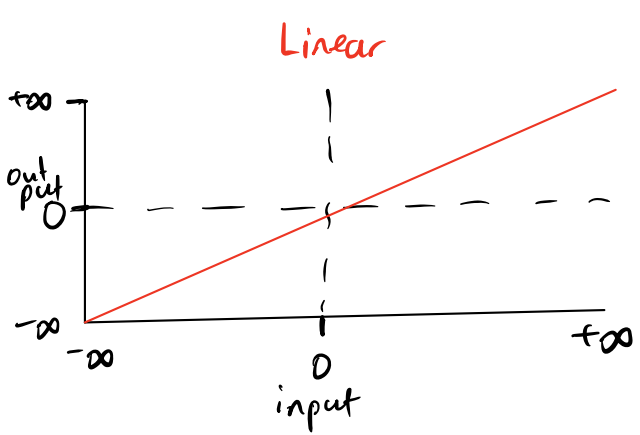
  Many open problems remain. How are these networks actually learning ? How can they be improved ? What are the limits to neural learning ?

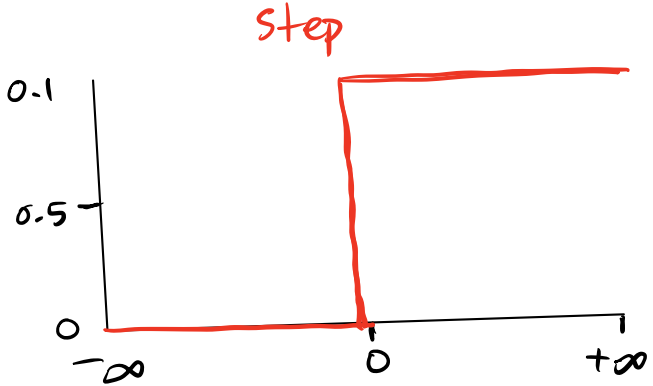
McCulloch, W. and Pitts, W. (1943). A Logical Calculus of the Ideas Immanent in Nervous Activity. *Bulletin of Mathemetical Biophysics*, 5:115–133.

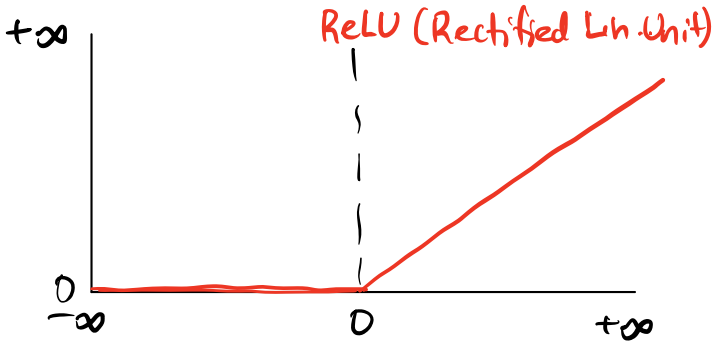
Ripley, B. (1996). *Pattern Recognition and Neural Networks*. Cambridge University Press.

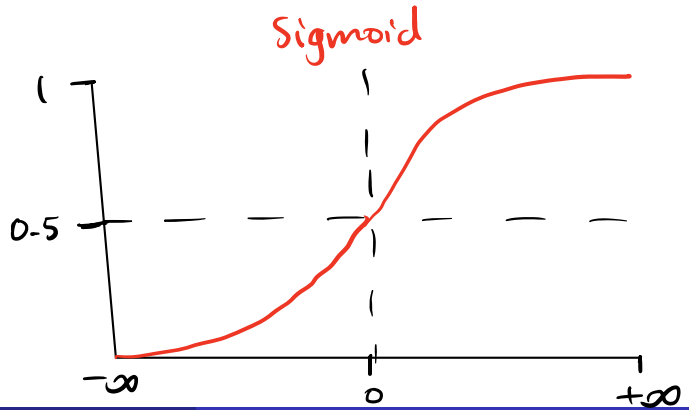
Strang, G. (2019). *Linear Algebra and Learning from Data*. Wellesley - Cambridge Press.

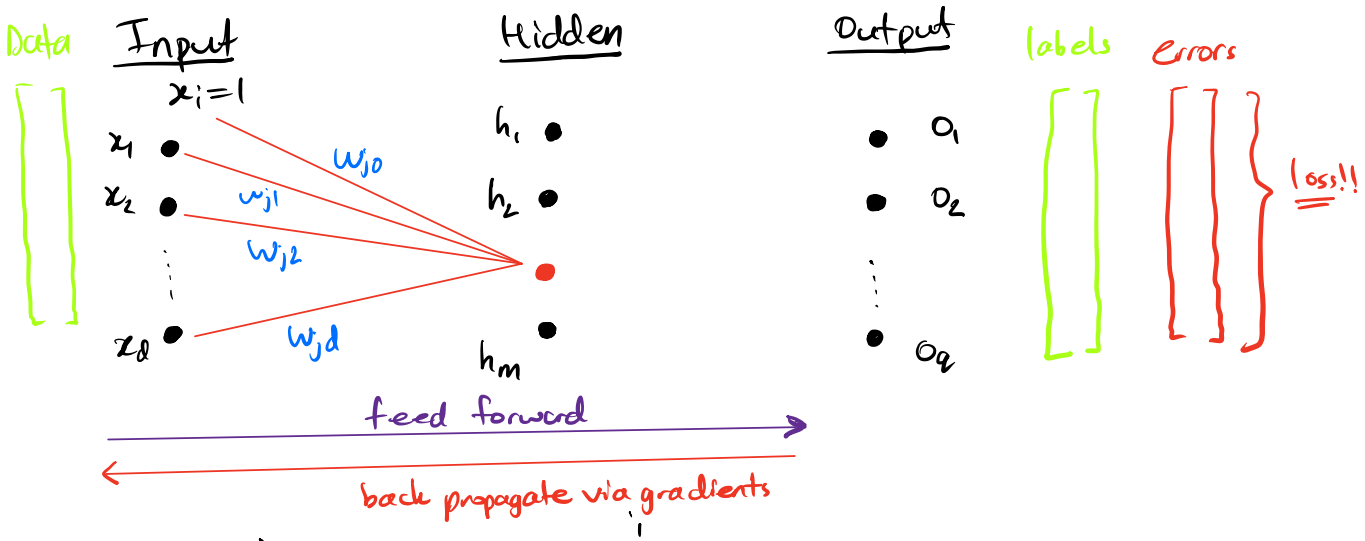
Some activation functions

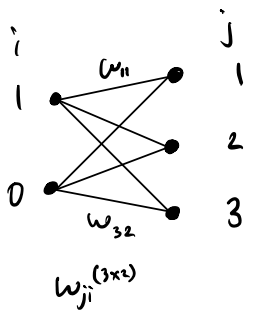


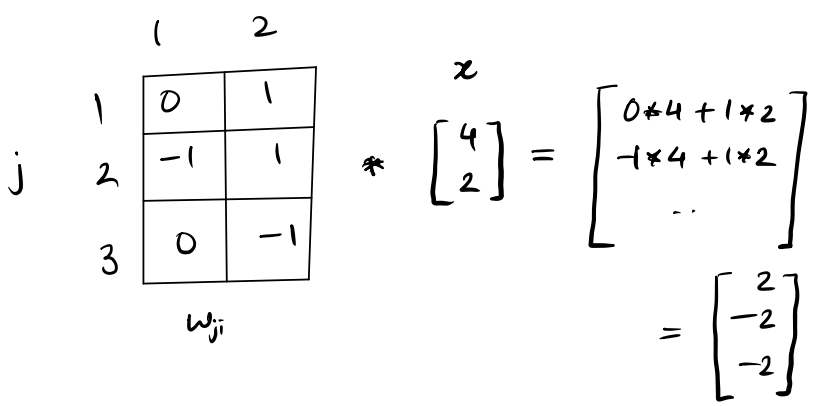







Data
Input
Hidden
Output
labels
Errors
loss!!
$x_i = 1$
x_1
x_2
x_d
w_{j0}
w_{j1}
w_{j2}
w_{jd}
h_1
h_2
h_m
o_1
o_2
o_q
feed forward
back propagate via gradients


i
j
1
0
1
2
3
w_{11}
w_{32}
$w_{ji}^{(3\times2)}$


i
1
2
j
1
2
3
0
1
−1
1
0
−1
w_{ji}
x
$\begin{bmatrix} 4 \\ 2 \end{bmatrix} = \begin{bmatrix} 0*4+1*2 \\ -1*4+1*2 \\ \vdots \end{bmatrix}$
$= \begin{bmatrix} 2 \\ -2 \\ -2 \end{bmatrix}$