COMS30035, Machine learning: Neural networks

James Cussens

School of Computer Science
University of Bristol

6th September 2024

Acknowledgement

- These slides are adapted from ones originally created by Rui Ponte Costa and later edited by Edwin Simpson.

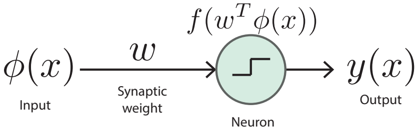
Agenda

- Perceptron
- Neural networks (multi-layer perceptron)
  - Architecture
  - The backpropagation algorithm
  - Gradient descent

See: [Chapter 5, Bishop]

Perceptron – a simplified neural network

- It is the very beginning of neural network models in ML!
- It is directly inspired by how neurons process information:

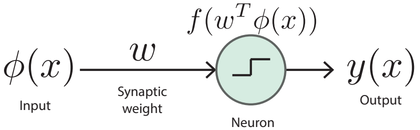


- It is an example of a linear discriminant model given by $y(\mathbf{x}) = f(\mathbf{w}^T\phi(\mathbf{x}))$
  with a nonlinear *activation function* $f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$
- Here the target $t = \{+1, -1\}$.
- And we aim to mimimise the following error $-\sum_{n=1}^{N} \mathbf{w}^T\phi_n t_n$ [1]

[1] Intuitively we want to improve our chances of having $t_n = y_n = -1$ or $t_n = y_n = 1$, which will both decrease our error function.

Perceptron – a simplified neural network

- It is the very beginning of neural network models in ML!
- It is directly inspired by how neurons process information:

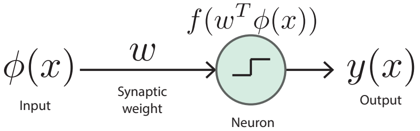


- It is an example of a linear discriminant model given by $y(\boldsymbol{x}) = f(\boldsymbol{w}^T\phi(\boldsymbol{x}))$
  with a nonlinear *activation function* $f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$
- Here the target $t = \{+1, -1\}$.
- And we aim to mimimise the following error $-\sum_{n=1}^{N} \boldsymbol{w}^T\phi_n t_n$ [1]

[1] Intuitively we want to improve our chances of having $t_n = y_n = -1$ or $t_n = y_n = 1$, which will both decrease our error function.

Perceptron – a simplified neural network

- It is the very beginning of neural network models in ML!
- It is directly inspired by how neurons process information:

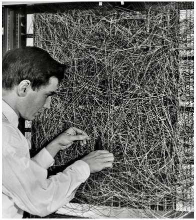


- It is an example of a linear discriminant model given by $y(\boldsymbol{x}) = f(\boldsymbol{w}^T\phi(\boldsymbol{x}))$ with a nonlinear *activation function* $f(a) = \begin{cases} +1, & a \geq 0 \\ -1, & a < 0 \end{cases}$
- Here the target $t = \{+1, -1\}$.
- And we aim to mimimise the following error $-\sum_{n=1}^{N} \boldsymbol{w}^T\phi_n t_n$ [1]

[1] Intuitively we want to improve our chances of having $t_n = y_n = -1$ or $t_n = y_n = 1$, which will both decrease our error function.

Perceptron – a simplified neural network

Example



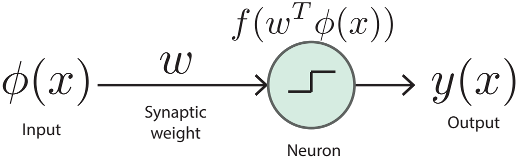
The Perceptron of Rosenblatt (1962)

Perceptrons started the journey to the current *deep learning* revolution!
Frank Rosenblatt used IBM and special-purpose hardware for a parallel implementation of perceptron learning.
Marvin Minksy, showed that such models could only learn *linearly separable problems*.
However, this limitation is only true in the case of single layers!

source: Bishop p193.

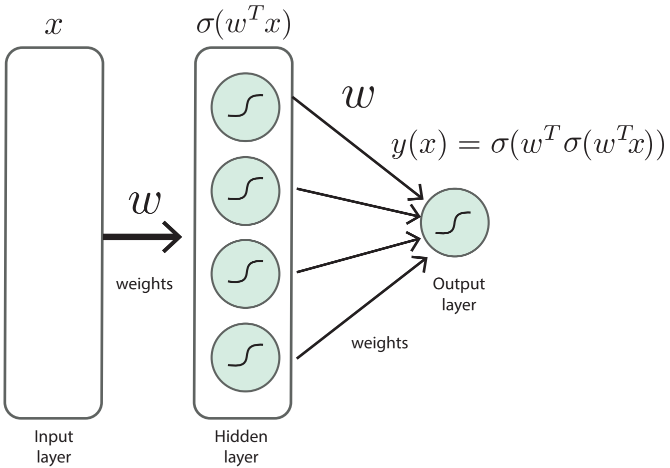
Neural networks

From a single layer perceptron:



Neural networks

To a Multiple Layer Perceptron (MLP) [2]:



[2] Although, we call it perceptron, it typically uses logistic sigmoid activation functions (continous nonlinearities), instead of step-wise discontinous nonlinearities. NB the first and second weight vectors will be different, despite both being denoted by *w* here.

Neural networks

- Neural networks are at heart composite functions of linear-nonlinear functions.
- **Deep learning**[3] refers to neural networks (or MLPs) with more than 1 hidden layer
- They can be applied in any form of learning, but we will focus on supervised learning and classification in particular
- MLP recipe [4]:
  - Define architecture (i.e. how many hidden layers and neurons) [5]
  - Define cost function (e.g. mean squared error)
  - Optimise network using backprop:
    1. Forward pass – calculate activations; generate y_k
    2. Calculate error/cost function
    3. Backward pass – use backprop to compute error gradient
    4. Use gradient to improve parameters

[3] If you would like to learn more take our Applied Deep Learning unit in your 4th year.

[4] Here we focus on simple feedforward nnets but the recipe is the same for any neural network.

[5] Note that this makes them parametric models.

Neural networks

- Neural networks are at heart composite functions of linear-nonlinear functions.
- **Deep learning**[3] refers to neural networks (or MLPs) with more than 1 hidden layer
- They can be applied in any form of learning, but we will focus on supervised learning and classification in particular
- MLP recipe [4]:
  - Define architecture (i.e. how many hidden layers and neurons) [5]
  - Define cost function (e.g. mean squared error)
  - Optimise network using backprop:
    1. Forward pass – calculate activations; generate y_k
    2. Calculate error/cost function
    3. Backward pass – use backprop to compute error gradient
    4. Use gradient to improve parameters

[3] If you would like to learn more take our Applied Deep Learning unit in your 4th year.

[4] Here we focus on simple feedforward nnets but the recipe is the same for any neural network.

[5] Note that this makes them parametric models.

Neural networks – forward pass step-by-step

1. Calculate activations of the hidden layer h: $a_j = \sum_{i=1}^{D} w_{ji}^{(h)} x_i + w_{j0}^{(h)}$ [linear]
2. Pass it through a nonlinear function: $z_j = \sigma(a_j)$ [nonlinear[6]]
3. Calculate activations of the output layer o: $a_k = \sum_{j=1}^{hiddensize} w_{kj}^{(o)} z_j + w_{k0}^{(o)}$ [linear]
4. Compute predictions using a sigmoid: $y_k = \sigma(a_k)$ [nonlinear[7]]
5. All together: $y_k = \sigma \left(\sum_{i=1}^{D} w_{kj} \, \sigma \left(\sum_{i=1}^{D} w_{ji} x_i^{(h)} + w_{j0}^{(h)} \right) + w_{k0}^{(o)} \right)$

[6] In MLP we typically use sigmoid functions.

[7] For classification problems we use a sigmoid at the output, where each output neuron codes for one class.

Neural networks – forward pass step-by-step

1. Calculate activations of the hidden layer h: $a_j = \sum_{i=1}^{D} w_{ji}^{(h)} x_i + w_{j0}^{(h)}$ [linear]
2. Pass it through a nonlinear function: $z_j = \sigma(a_j)$ [nonlinear[6]]
3. Calculate activations of the output layer o: $a_k = \sum_{j=1}^{hiddensize} w_{kj}^{(o)} z_j + w_{k0}^{(o)}$ [linear]
4. Compute predictions using a sigmoid: $y_k = \sigma(a_k)$ [nonlinear[7]]
5. All together: $y_k = \sigma\left(\sum_{i=1}^{D} w_{kj}\, \sigma\left(\sum_{i=1}^{D} w_{ji} x_i^{(h)} + w_{j0}^{(h)}\right) + w_{k0}^{(o)}\right)$

[6] In MLP we typically use sigmoid functions.

[7] For classification problems we use a sigmoid at the output, where each output neuron codes for one class.

Neural networks – forward pass step-by-step

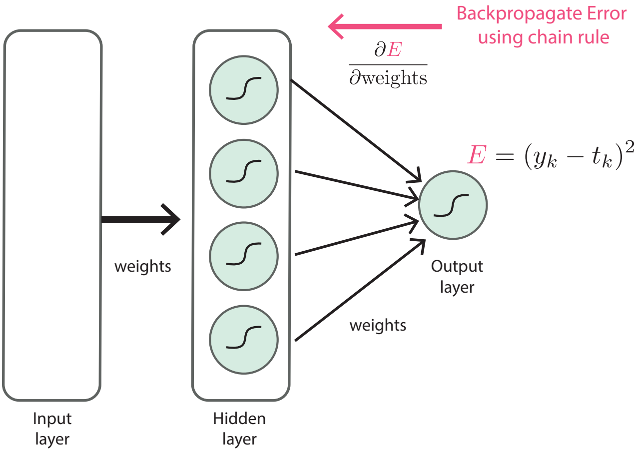
1. Calculate activations of the hidden layer h: $a_j = \sum_{i=1}^{D} w_{ji}^{(h)} x_i + w_{j0}^{(h)}$ [linear]
2. Pass it through a nonlinear function: $z_j = \sigma(a_j)$ [nonlinear[6]]
3. Calculate activations of the output layer o: $a_k = \sum_{j=1}^{hiddensize} w_{kj}^{(o)} z_j + w_{k0}^{(o)}$ [linear]
4. Compute predictions using a sigmoid: $y_k = \sigma(a_k)$ [nonlinear[7]]
5. All together: $y_k = \sigma\left(\sum_{i=1}^{D} w_{kj}\,\sigma\left(\sum_{i=1}^{D} w_{ji} x_i^{(h)} + w_{j0}^{(h)}\right) + w_{k0}^{(o)}\right)$

[6] In MLP we typically use sigmoid functions.

[7] For classification problems we use a sigmoid at the output, where each output neuron codes for one class.

Neural networks – backward pass

We now need to optimise our weights, and as before we use derivatives to find a solution. Effectively backpropagating the output error signal across the network – backpropagation algorithm.



This is a special case where the output layer has a single node. In the general case we could have e.g. $E = \sum_k (y_k - t_k)^2$.

Total error as a sum of datapoint errors

- Typically the error for an entire dataset breaks down into a sum of errors for each individual datapoint:

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

- This means that the error gradient for the entire dataset with respect to some weight w_{ji} will just be the sum of error gradients for each datapoint:

$$\frac{\partial E}{\partial w_{ji}} = \sum_{n=1}^{N} \frac{\partial E_n}{\partial w_{ji}}$$

- So we focus attention on computing the $\frac{\partial E_n}{\partial w_{ji}}$ values.

Computing partial derivatives 1

- Although all values in the network depend on the datapoint n we're focusing on, we will (like Bishop) omit the subscript n from network values to reduce clutter.
- Consider an arbitrary unit j (aka 'node') in an neural network.
- We will associate the *bias* for a unit with a dummy input fixed to 1.
- It computes a value z_j by first computing a_j, a weighted sum of its inputs (from the prevous layer), and then sending a_j to some nonlinear *activation function* h:

$$a_j = \sum_i w_{ji} z_i \tag{1}$$

$$z_j = h(a_j) \tag{2}$$

$$\tag{3}$$

- Since E_n (the error for the nth datapoint) only depends on weight w_{ji} via a_j we can use the chain rule to write:

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

Computing partial derivatives 2

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}}$$

- Since $a_j = \sum_i w_{ji} z_i$, $\frac{\partial a_j}{\partial w_{ji}}$ is easy to compute:

$$\frac{\partial a_j}{\partial w_{ji}} = z_i$$

As for $\frac{\partial E_n}{\partial a_j}$ let's just introduce some notation for now: $\delta_j \equiv \frac{\partial E_n}{\partial a_j}$

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{4}$$

Computing partial derivatives 3

- We compute the $\delta_j \equiv \frac{\partial E_n}{\partial a_j}$ *backwards* starting with the output units.
- For example, if the error (for a single datapoint) is $E_n = \frac{1}{2}\sum_k (y_k - t_k)^2$ then:

$$\delta_k = y_k - t_k \tag{5}$$

- For the hidden units we use the chain rule:

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j}$$

"where the sum runs over all units k to which unit j sends conections" (Bishop).

Computing partial derivatives 4

$$\delta_j \equiv \frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j}$$

can be reformulated to give the *backpropagation formula*:

$$\delta_j = h'(a_j) \sum_k w_{kj} \delta_k \tag{6}$$

Error Backpropagation

As given in Bishop:

1. Apply an input vector $\mathbf{x}_n$ to the network and forward propagate through the network using (1) and (2) to find the activations of all the hidden and output units.
2. Evaluate the δ_k for all the outputs units using e.g. (5).
3. Backpropagate the δ's using (6) to obtain δ_j for each hidden unit in the network.
4. Use (4) to evalute the required derivatives.

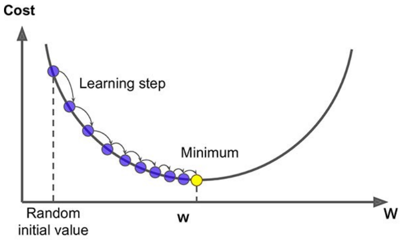
Backpropagation in practice

- It is crucial to organise the computation of all these partial derivatives (i.e. the gradient $\partial E/\partial \mathbf{w}$) efficiently.
- The quantities required will be organised in tensors (generalisation of matrices beyond 2 dimensions).
- The `PyTorch` approach is called *Autograd*. There's lots of useful info/explanation of how it works on the PyTorch site.

PyTorch's Autograd feature is part of what make PyTorch flexible and fast for building machine learning projects. It allows for the rapid and easy computation of multiple partial derivatives (also referred to as gradients) over a complex computation. This operation is central to backpropagation-based neural network learning. The Fundamentals of Autograd

Neural networks – gradient descent [9]

In many ML methods is common to iteratively update the parameters by descending the gradient.



8

In our neural network this means to update the weights using:

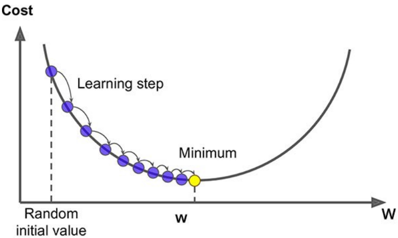
- $w_{ji} = w_{ji} - \Delta w_{ji}$, where $\Delta w_{ji} = \sigma'(y_n - t_n) w_{kj}^T \sigma' x_i$
- $w_{kj} = w_{kj} - \Delta w_{kj}$, where $\Delta w_{kj} = \sigma'(y_n - t_n) z_j$
- This is often done in mini-batches – using a small number of samples to compute Δw.

[8] Figure from https://mc.ai/an-introduction-to-gradient-descent-2/ (link no longer working)

[9] Its called descent because we are minimising the cost function, so descending on the function landscape, which can be quite hilly!

Neural networks – gradient descent [9]

In many ML methods is common to iteratively update the parameters by descending the gradient.



[8]

In our neural network this means to update the weights using:

- $w_{ji} = w_{ji} - \Delta w_{ji}$, where $\Delta w_{ji} = \sigma'(y_n - t_n) w_{kj}^T \sigma' x_i$
- $w_{kj} = w_{kj} - \Delta w_{kj}$, where $\Delta w_{kj} = \sigma'(y_n - t_n) z_j$
- This is often done in mini-batches – using a small number of samples to compute Δw.

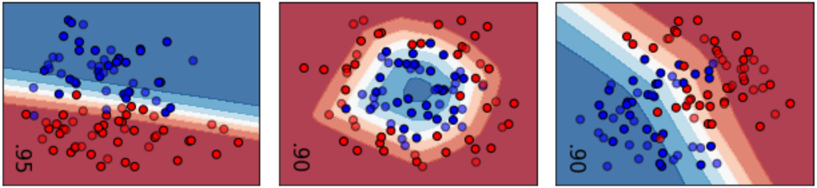
[8] Figure from https://mc.ai/an-introduction-to-gradient-descent-2/ (link no longer working)

[9] Its called descent because we are minimising the cost function, so descending on the function landscape, which can be quite hilly!

Neural networks

Example

Using sklearn we fitted a MLP classifier to the data:

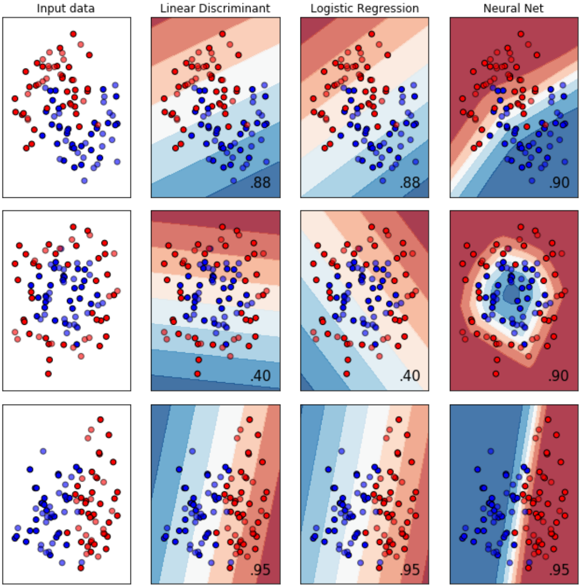


An MLP with one hidden layer can perform well in nonlinear classification problems.

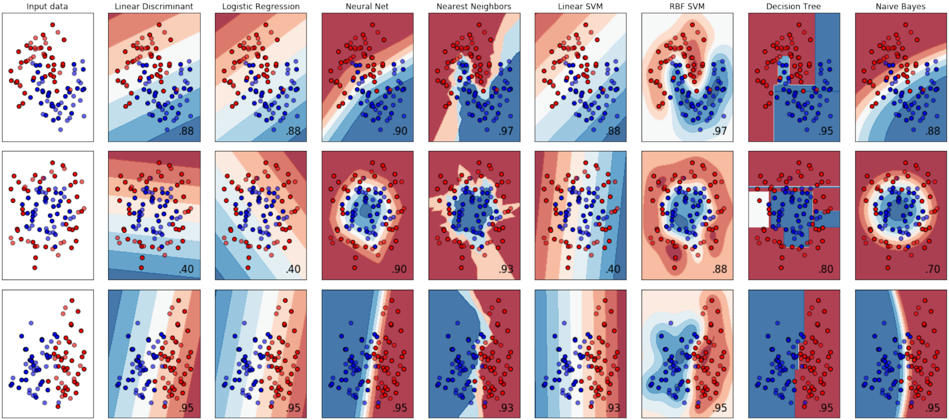
However, because MLPs are highly flexible they can easily *overfit*.

Solutions: *early stopping* (stop when test performance starts decreasing) and *regularisation* methods such as *dropout* (randomly turn off units during training).

Classification methods – overall comparison



Classification methods – overall comparison [we don't cover all these methods]



Quiz and video time!

Watch this very cool video about the perceptron [10].

Go to Blackboard unit page $\gg$ Quizzes $\gg$ Week 1 $\gg$ Classification and neural networks

[Should take you less than 5 minutes]

[10] Note the comment at the end – it underlies all the recent successes using deep learning!