Optimisation of neural networks

Stephen Eglen

Copyright notice

Optimisation

How to set weights in a neural network?

Aims for the lectures:

1. Neuroscience fundamentals
2. Perceptron
3. Multi-layer perceptrons
4. Back propagation algorithm
5. Finding derivatives
6. A catalogue of techniques to improve learning

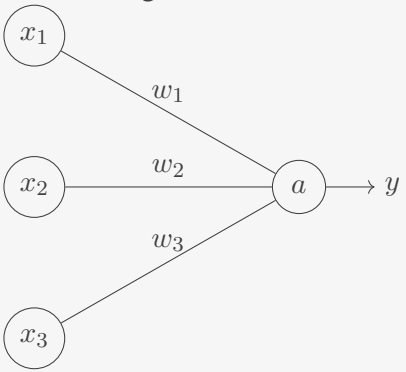
Neuroscience fundamentals

Definitions to cover:

- neuron components: dendrites, cell body (soma), axon (synapses)
- “Spikes” or action potentials (all or nothing)
- Threshold behaviour.
- Rate coding.
- Excitatory vs inhibitory connections
- Cell death as a fact of life.
- distributed processing and graceful degradation

The perceptron (Rosenblatt 1957)

The building block of neural networks:



- x_i: activity of input unit i (binary or [0,1]).
- w_i synaptic weight from unit i.
- $a = \sum_i w_i x_i$ total input to the output unit.
- y: activity of output unit.
- t: desired output (of use later). (μ superscript denotes training sample.)
- $f(\cdot)$: activation function; $y = f(a)$.

Training set

Sample	x_1	x_2	t
$\mu = 1$	0	0	0
$\mu = 2$	0	1	0
$\mu = 3$	1	0	0
$\mu = 4$	1	1	1

Activation functions

Given total weighted input a to neuron, what is its output?

1. identity: $f(a) = a$.
2. threshold: $f(a, \theta) = \begin{cases} 1 & \text{if } a \geq \theta \\ 0 & \text{otherwise} \end{cases}$
3. sigmoidal: $f(a) = \frac{1}{1+\exp(-ka)}$
4. tanh: $f(a) = tanh(a)$
5. rectified linear unit (ReLU): $f(a) = max(0, a)$

How to choose? One key property: differentiable.

These functions are also known as *activation* functions.

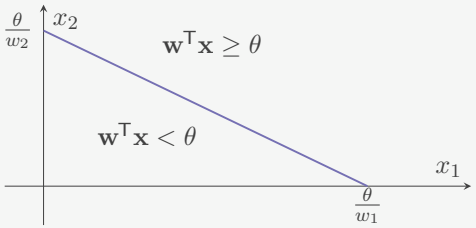
Perceptron decisions

The equation

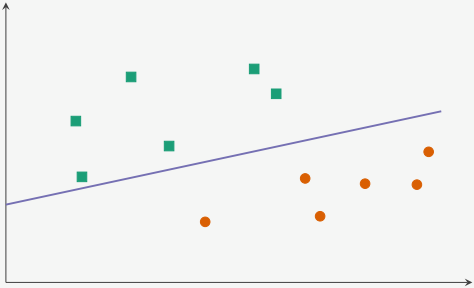
$$\sum_{i=1}^{N} w_i x_i = \theta$$

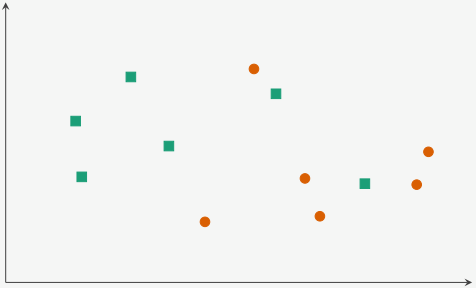
defines a hyperplane in N-dimensional space. This hyperplane cuts the space in two.

$$w_1 x_1 + w_2 x_2 = \theta$$
$$x_2 = \left(\frac{-w_1}{w_2}\right) x_1 + \frac{\theta}{w_2}$$



Linearly separable problems and the perceptron



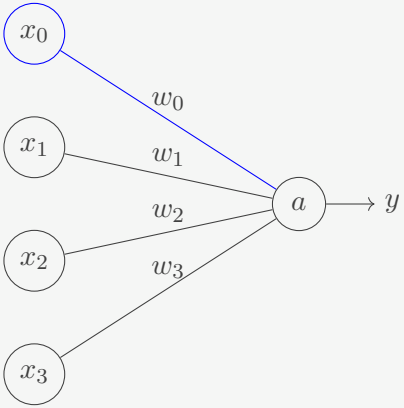


Learning involves adjusting the values of w and θ so that the decision plane can correctly divide the two classes.

Live coding example with fixed weights

Threshold & Bias

The threshold, θ, is just another weight $w_0 = \theta$ from a new input unit fixed at $x_0 = -1$. This new input unit is called the **bias** unit.

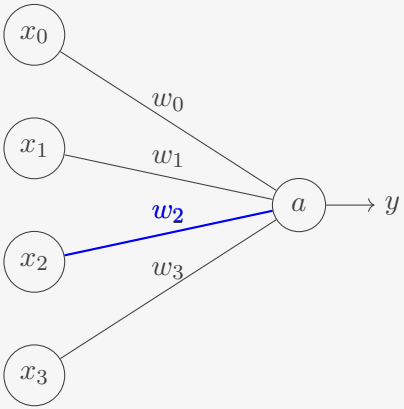


Instead of comparing $\sum_{i=1}^{N} w_i x_i$ with θ, we compare $\sum_{i=0}^{N} w_i x_i$ with 0, or

$$y = \begin{cases} 1 & \text{if } \sum_{i=0}^{N} w_i x_i > 0, \\ 0 & \text{otherwise} \end{cases}$$

Now, learning is about jiggling **weights** only.

Intuitively... (positive valued inputs)



$$a = \sum_{i=0}^{N} w_i x_i \qquad y = \text{step}(a)$$

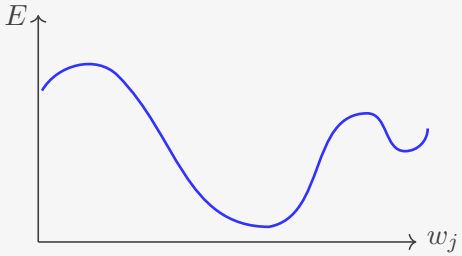
Consider w_j's contribution to a in different cases:

1. $y = t$ Perceptron has classified input correctly – **do nothing**
2. $x_j = 0$ Changing w_j will not affect $\sum_i w_i x_i$ – **do nothing**
3. $x_j \neq 0, y < t \implies a$ is too low – **increase** w_j
4. $x_j \neq 0, y > t \implies a$ is too high – **decrease** w_j

The local rule:

$$\boxed{\Delta w_j \propto (t - y) x_j}$$

Perceptron learning rule: gradient descent



$$y = f(\mathbf{w} \cdot \mathbf{x}) \qquad \text{e.g.} \quad f(a) = 1/(1+\exp(-a)), \quad f(a) = a$$

$$E = \frac{1}{2}(t-y)^2 \qquad t \text{ is target output}$$

$$\Delta w_j = -\epsilon \frac{\partial E}{\partial w_j} = -\epsilon \frac{\partial E}{\partial y}\frac{\partial y}{\partial w_j} = \epsilon(t-y)f'(\mathbf{w}\cdot\mathbf{x})x_j$$

Perceptron convergence theorem (PCT)

Starting from initial weights $\mathbf{w} = \mathbf{0}$, the **perceptron convergence theorem** guarantees that *if* there are ideal weights $\mathbf{w}s$ separating two classes, the perceptron rule will stop training after finite number of updates.
The network weights $\mathbf{w}$ at the end of training will then perfectly solve the classification task.
(Note that $\mathbf{w}$ will be similar, but not necessarily equal, to $\mathbf{w}s$.)

See (Dayan and Abbott, page 327) (Dayan and Abbott, 2001) for proof of convergence.

Variants of Perceptron learning rule

$$y = H(\mathbf{w}^\mathsf{T}\mathbf{x})$$
$$\Delta\mathbf{w} = \epsilon(t - y)\mathbf{x}$$

No need to change weights if it is working correctly.
Versus more general form (delta rule)

$$y = f(\mathbf{w}^\mathsf{T}\mathbf{x})$$
$$\Delta\mathbf{w} = \epsilon(t - y)f'(\mathbf{w}^\mathsf{T}\mathbf{x})\mathbf{x}$$

Back propagation

XOR problem

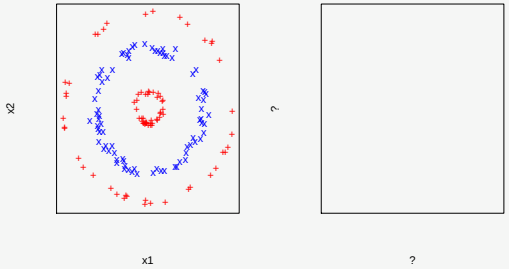
1. Minsky and Papert (1969) declared perceptrons dead when showing it couldn't solve XOR problem.

Sample	x_1	x_2	t
$\mu = 1$	0	0	0
$\mu = 2$	0	1	1
$\mu = 3$	1	0	1
$\mu = 4$	1	1	0

2. Need for multiple layers to provide non-linear transformation

The importance of features

- Find the right features to make the task solvable:



- Engineering features by hand is hard.
- Neural networks learn features that they find important.

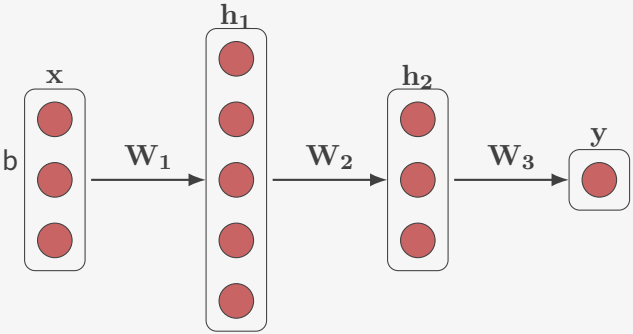
How many layers of features do you need?

One hidden layer is all you need **in theory** to make a “universal approximator” (Cybenko 1989; Hornik 1991).
Online example:
`http://neuralnetworksanddeeplearning.com/chap4.html` shows how to approximate 1-d and 2-d function with one layer of (many) hidden units.
With linear transfer functions how many layers do we need?

Neural networks and linear algebra / 1

- Activation of a layer of neurons stored in a **vector**.
- Synapses from one layer to another stored in a **weight matrix**: W_{ji} is strength of connection from unit i in one layer to unit j in next layer.
- "The single key fact about vectors and matrices is that each vector represents a point located in space, and a matrix moves that point to a different location. Everything else is just details." (Stone 2019, Appx. C).

Neural networks and linear algebra / 2



$$\mathbf{h_1} = g(\mathbf{W_1 x})$$
$$\mathbf{h_2} = g(\mathbf{W_2 h_1})$$
$$\mathbf{y} = g(\mathbf{W_2 h_2})$$
$$\mathbf{y} = g(\mathbf{W_3} g(\mathbf{W_2} g(\mathbf{W_1 x})))$$

If $g(x) = x$ then this reduces to

$$y = \mathbf{W_3 W_2 W_1 x} = \mathbf{M x}$$

Origins of back propagation?

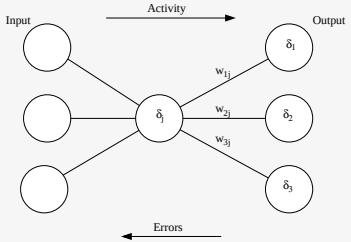
Amari used it in 1960s. Popularised in 1980s by Hinton et al.

Learning in multi-layer perceptrons

How to solve the **credit assignment problem**? i.e. what is the "delta" for hidden units, that have no desired output?

$$\delta_j = g'(h_j) \sum_i w_{ij} \delta_i$$

h_j is total input to unit j; g' is 1st derivative of activation function.



No guarantee (unlike PCT) that this will converge due to local minima.

Some terms

1. **Online learning**: learn after every input. (each **iteration**). Sometimes called **stochastic gradient descent** as approximating gradient with one sample at a time.
2. **Batch learning**: wait until all training samples have been presented (each **epoch**).
3. **epoch**: one presentation of all inputs. Training consists of a variable number of epochs until (hopefully) convergence is reached.
4. **Mini batch**: break data into several groups (make sure each is balanced).

Some common alternatives for learning

1. Alternative loss functions, e.g. cross-entropy, useful for 1-of-N (one-hot) classification:

$$E = \sum_{k=1}^{K} t_k \log \frac{1}{y_k} + (1 - y_k) \log \frac{1}{(1 - y_k)}$$

2. Weight regularisation terms on loss functions.

$$E = \sum_{k=1}^{K} (t_k - y_k)^2 + \mathbf{w}^\mathsf{T} \mathbf{w}$$

3. Other aspects, such as momentum, are quite well-established but will be discussed later.

Automatic Differentiation

Introduction to automatic differentiation

Back-propagation is nice, but error-prone to implement. Would be nice to have an easier approach. What alternatives are there?

Option 1: finite difference numerical approximation

Recall the definition of a gradient:

$$\frac{\mathrm{d}y}{\mathrm{d}x} = \lim_{h \to 0} \frac{y(x+h) - y(x)}{h}$$

so in the context of a network

$$\frac{\partial E}{\partial w_j} = \lim_{h \to 0} \frac{E(\mathbf{w} + h\mathbf{e_i}) - E(\mathbf{w})}{h}$$

Then we can numerically approximate this for each weight in the network. Expensive (recalculate for each weight) and error-prone: what step size h to take?

Option 2: analytical calculation

If we have an explicit expression for E, why not just take the derivative of it?

May work on restricted cases, but needs computer algebra system and there can be the problem of **expression swell**, when repeated derivatives get longer (Baydin et al., 2018).

$$f(x) = xe^x$$
$$f'(x) = e^x + xe^x$$

Exact but cannot cope with flow-control in programs (think about $\frac{\mathrm{d}|x|}{\mathrm{d}x}$).

Option 3: complex-step derivative

Like finite step, but in complex domain, where h is an imaginary number (Lyness and Moler, 1967).
Taylor expand $f(\cdot)$ around x:

$$f(x+ih) = f(x) + ihf'(x) - \frac{h^2 f''(x)}{2} + O(h^3)$$

$$\text{Re}: f(x+ih) = f(x) \qquad + O(h^2)$$

$$\text{Im}: f(x+ih) = \qquad hf'(x) \qquad + O(h^3)$$

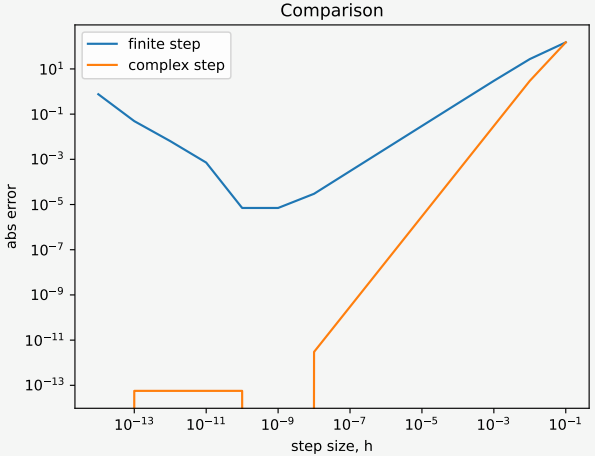
For small h ignore $O(h^2)$ terms; one eval of $f(x+ih)$ produces value and derivative:

$$f(x) \approx \text{Re} f(x+ih)$$

$$f'(x) \approx \text{Im} \frac{f(x+ih)}{h}$$

Comparison of finite-step and complex-step approaches

Example function $y(x) = x^2 + \frac{3}{x} + 4$ evaluated at $x = 0.1$.



Automatic differentiation

Two key approaches:

1. Forward mode.
2. Reverse mode.

Forward mode

Good when we have fan-out networks (not typical unfortunately).

Use of dual numbers defined by $u = a + \epsilon b$ where $\epsilon > 0, \epsilon^2 = 0$.
So, for two dual numbers, $u = a + \epsilon b, \quad v = c + \epsilon d$ we find:

1. $u + v = (a + c) + \epsilon(b + d)$

2. $u * v = (a + \epsilon b)(c + \epsilon d) = ac + \epsilon(bc + ad) + \epsilon^2 bd = ac + \epsilon(bc + ad)$

3. $\frac{u}{v} = \frac{a+\epsilon b}{c+\epsilon d} = \frac{(a+\epsilon b)}{(c+\epsilon d)}\frac{(c-\epsilon d)}{(c-\epsilon d)} = \frac{ac+\epsilon(bc-ad)}{c^2} = \frac{a}{c} + \epsilon\frac{(bc-ad)}{c^2}$

Reverse mode

1. Good for most networks where number of input units much larger than output units.
2. However, no simple implementation. Must store up intermediate results on forward pass; gradients then propagate backwards.

Micrograd

Worth watching the “building micrograd” video (2 1/2 hours) by Andrej Karpathy:
`https://www.youtube.com/watch?v=VMj-3S1tku0`
This requires graphviz installation. It draws the diagrams beautifully.

Another good one is Joel net, but fast-paced “live coding madness”
`https://www.youtube.com/watch?v=o64FV-ez6Gw`

Computational environments

1. Pytorch `http://www.pytorch.org`
2. Micrograd `https://github.com/karpathy/micrograd`
3. Flux.jl `https://fluxml.ai/Flux.jl/stable/`

A catalogue of techniques to improve learning

A catalogue of techniques to improve learning

- Most of the theory was around in the 70-80's. Why only so popular now?
- Computers are much more powerful
- (Labelled) data sets are much bigger
- There have been a host of pragmatic advances which have also come along, some of which are now discussed.

Initialising weights

How to choose initial weights?
The choice of initial weights influences where we converge to.

Biases: can be zero. **Weights**: if all zero, would mean features are the same.
Need to avoid saturating neurons.
Some pragmatic approaches depends on the activation function (n is fan-in to a neuron):

1. (linear, sigmoid, tanh): $U[-\frac{1}{\sqrt{n}}, +\frac{1}{\sqrt{n}}]$ (Glorot and Bengio, 2010)
2. (ReLU): $N(0, \frac{2}{\sqrt{n}})$ (He et al., 2015)

Overcoming local minima

Momentum

$$\Delta w_{ij}(t+1) = -\eta \, \frac{\partial E}{\partial w_{ij}} + \alpha \, \Delta w_{ij}(t)$$

where $0 < \alpha < 1$.

Adam method

Compare with SGD.
Adam (“adaptive momentum”) and other methods of minimisation have improved upon basic SGD (Kingma and Ba, 2014). See overview (Ruder, 2016). Adam has emerged as default.
Adam = Adaptive movement estimation. Uses per-parameter step-size that can adapt. Combines two other successful methods (momentum + RMSprop). Best stick to defaults.
Take each parameter independently and calculate.

Adam: the algorithm

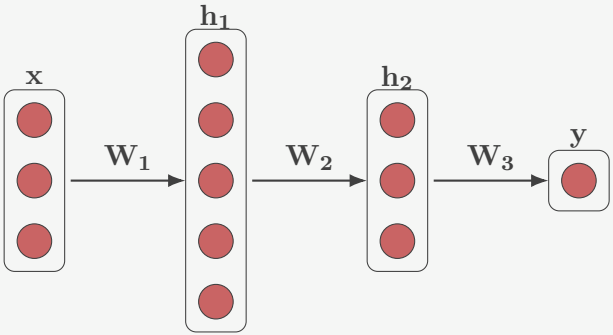
The following is applied independently to each weight (θ) in the network.
`https://arxiv.org/pdf/1412.6980.pdf` (Algorithm 1)

Default parameters: $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$. Init: $m_0 = v_0 = 0$.

$$
\begin{aligned}
g_t &\leftarrow \nabla_\theta f(\theta) \\
m_t &\leftarrow \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &\leftarrow \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &\leftarrow \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &\leftarrow \frac{v_t}{1-\beta_2^t} \\
\theta_t &\leftarrow \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}
\end{aligned}
$$

Transfer learning

Training large networks takes a long time...
Transfer learning allows us to re-use the early layer of networks.
Particularly useful with convolutional images.



Improving optimisation: batch normalisation

- Independently normalise activation of each unit (Ioffe and Szegedy, 2015) within a mini-batch.
- Take m activation values from a batch B for any one neuron $a_i, i = 1 \dots m$ with $\epsilon > 0$.

$$\mu = \frac{1}{m} \sum_{i=1}^{m} a_i, \qquad \sigma^2 = \epsilon + \frac{1}{m} \sum_{i=1}^{m} (a_i - \mu)^2$$

$$y_i = g(BN(a_i) + b) \qquad BN(a) = \frac{a - \mu}{\sigma}$$

- Keep running estimates of μ, σ during training to use during testing.
- Works by smoothing out energy landscape (Santurkar et al., 2018).
- Alternatives, e.g. layernorm, for normalisation of activity within each layer.

Dropout

- **Dropout** introduced as a way to reduce overfitting. Bank-teller analogy from Hinton.
- During training, on each iteration silence some fraction of units.
- Implemented during training:
  1. once the activation of a layer calculated, set output of half of neurons to zero.
  2. Double the activation of rest of the neurons.
- Avoid using together with batch normalisation.

Hyper parameters

How do you decide on the hyper-parameters of the model?

1. Network architecture: how many hidden units? how many hidden layers?
2. Learning rates (and other parameters)
3. Error functions
4. Feature selection
5. Data set size

One approach is to overfit then regularize. Sophisticated methods exist, but often more pragmatic approaches taken.
Caution: searching for the 'best' result can lead to over-concerns on performance in narrow settings. Depends on your question.
This is very much an empirical science. See: `https://paperswithcode.com/sota/image-classification-on-mnist`

References I

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. *J. Mach. Learn. Res.*, 18(153):1–43.

Dayan, P. and Abbott, L. F. (2001). *Theoretical Neuroscience: Computational and Mathematical Modeling of Neural Systems*. MIT Press.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, pages 249–256.

He, K., Zhang, X., Ren, S., and Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456. PMLR.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *Arxiv*, 1412.6980.

References II

Lyness, J. N. and Moler, C. B. (1967). Numerical differentiation of analytic functions. *SIAM J. Numer. Anal.*, 4(2):202–210.

Ruder, S. (2016). An overview of gradient descent optimization algorithms.

Santurkar, S., Tsipras, D., Ilyas, A., and Madry, A. (2018). How does batch normalization help optimization?

Copyright

Camera test