Introduction to Artificial Intelligence

VICTORIA UNIVERSITY OF WELLINGTON
TE HERENGA WAKA

COMP307/AIML420

Neural Networks 3: Neural Engineering

Dr Andrew Lensen

Andrew.Lensen@vuw.ac.nz

Outline

- Back propagation algorithm to train neural network
- Other considerations when designing (engineering) NNs

- **NB:** We are perhaps *slightly* behind in content – any slides that we do not finish today will become *readings*: ***still examinable***, but more "advanced knowledge" for A-level questions.

Training a Neural Network

- **Initialise** the weights (randomly)

- **Feedforward**
  - For each example, calculate the predicted outputs o_z using the current weights
  - Calculate the total *error* $\sum_z (d_z - o_z)^2$

- If the error is small enough, we can stop.

- Otherwise, we use **back propagation** to adjust the weights to make the error *smaller*.
  - Uses gradient descent (GD)

Back Propagation (BP) Algorithm

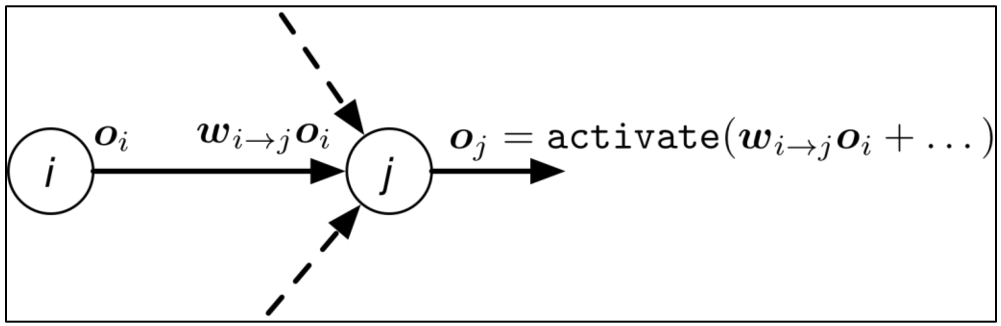
- Estimate the **contribution (gradient)** of each weight to the *error*, i.e. how much the error will be reduced by changing the weight by the gradient.

- Change each weight (simultaneously) proportional to its contribution to reduce the error as much as possible
  - Move in the direction of the **steepest gradient**

- We calculate the contribution/gradient backwards (from the last/output layer to the first hidden layer)
- Error of a single **output** node is $d_z - o_z$
  - d_z means "*desired*"
  - o_z means "output" (i.e. what we *actually* got)

Back Propagation (BP) Algorithm

- How big a change should we make to weight $w_{i \to j}$?
  - Make a big change if will improve error a lot (big contribution)
  - Make a small change if little effect on error (small contribution)

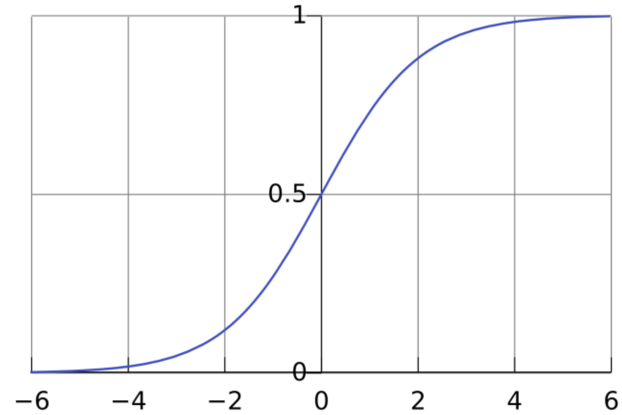


$w_{i \to j} \qquad w_{i \to j} o_i \qquad o_j = act(w_{i \to j} o_i + \cdots) \qquad \ldots \qquad error$

- β_j is how "*beneficial*" a change is for node j ("error term")
- When changing $w_{i \to j}$, the error change should be:
  - Proportional to the output: o_i (larger output = more effect)
  - Proportional to the *slope* of the activation function at node j: $slope_j$
  - Proportional to error term of j (β_j)

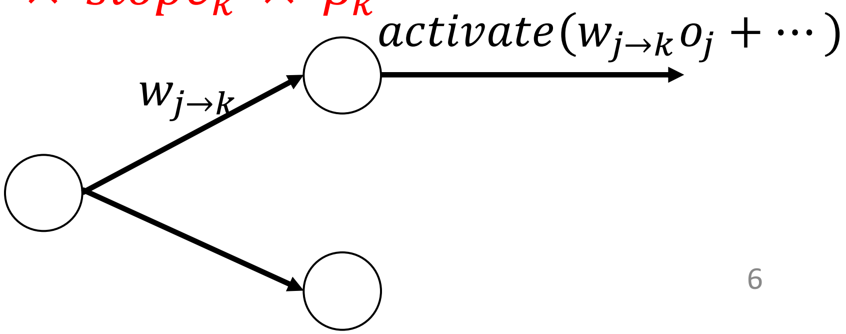
Back Propagation (BP) Algorithm

- How to calculate $slope_j$?
  - Some calculus knowledge: derivative of the activation function
  - Steeper (larger) the slope, larger the effect of changing the weight
  - We don't expect calculus in this course!



- How to calculate β_j?
  - *Back-propagated* from later layer

  - The **output layer**: the error $\beta_z = d_z - o_z$
  - **Other layers**: error is $\beta_j = \sum_k w_{j\to k} \times slope_k \times \beta_k$



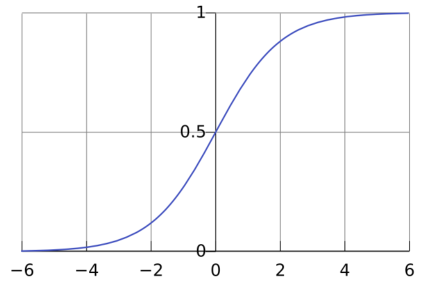
Back Propagation (BP) Algorithm

- Assume a neural network with:
  - Activation function: sigmoid

$$slope_j = o_j(1 - o_j)$$

  - Target: minimise total sum squared error

$$error = \frac{1}{2} \sum_{s \in examples} \sum_{c \in classes} (d_{sc} - o_{sc})^2$$



$$o_j = \frac{1}{1+e^{-ws_j}}$$

Makes the maths easier! (differentiation)

- Output node:

$$\beta_z = d_z - o_z$$

- Hidden node:

$$\beta_j = \sum_k w_{j \to k} o_k (1 - o_k) \beta_k$$

BP Algorithm Implementation

- Let η be the learning rate ("eta"…)

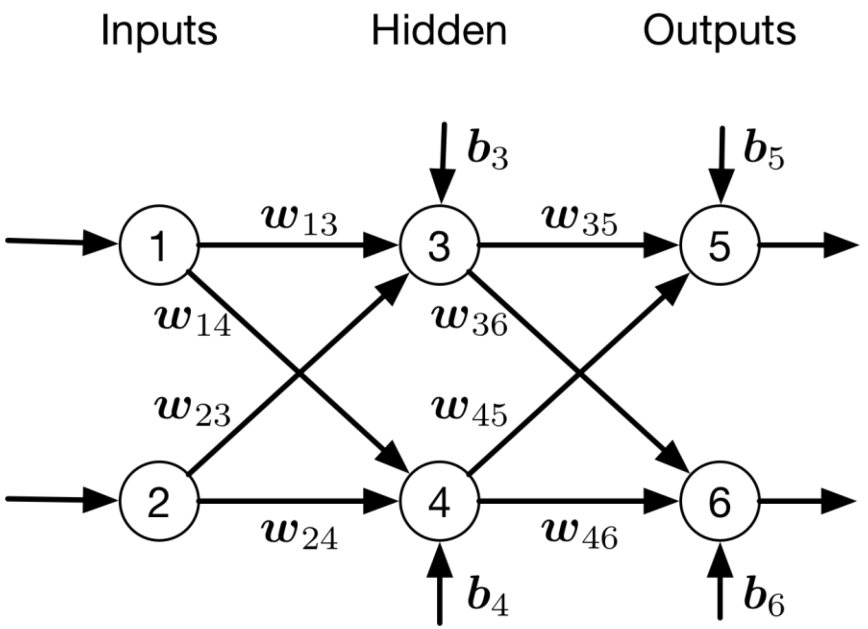
- Initialise all weights (+bias) to small random values

- Until total error is small enough, repeat:
  - For each input example:
    - Feed forward pass to get predicted outputs
    - Compute $\beta_z = d_z - o_z$ for each output node
    - Compute $\beta_j = \sum_k w_{j\to k} o_k (1 - o_k) \beta_k$ for each hidden node (working backwards from last to first layer)
    - Compute (+store) the weight changes for all weights
      $\Delta w_{i\to j} = \eta o_i o_j (1 - o_j) \beta_j$ (proportional to all 3 factors)
  - Sum up weight changes for all input examples
  - Change weights!

BP Algorithm Example

- Calculate one pass of the BP algorithm given the example (feedforward + back propagation)

Inputs		Outputs	
I_1	I_2	d_5	d_6



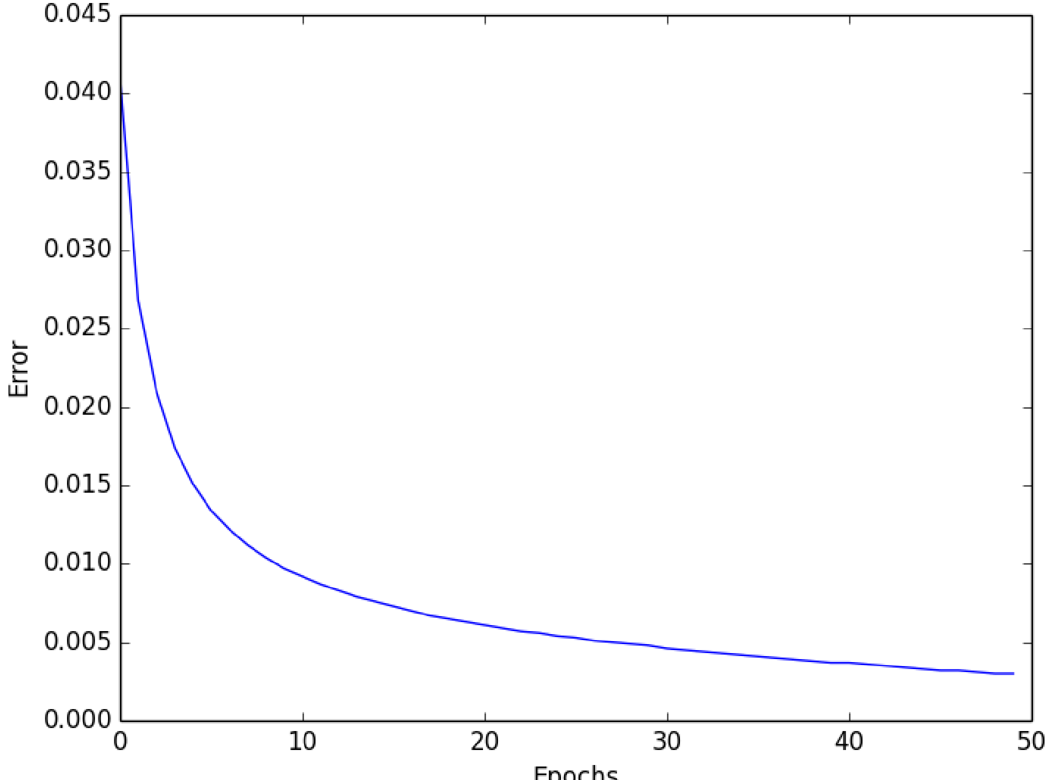
Walking through an example of feedforward and backpropagation from class

Mathematica

PDF Video

Notes on BP Algorithm

- *1 Epoch*: all input examples (entire training set, batch, …)
- A target of 0 or 1 cannot be reached in reasonable time. Usually interpret an output > 0.9 or > 0.8 as '1'
- Training may require *thousands* of epochs. A convergence curve can help to decide when to stop (overfitting?)



Weight Update Frequency

- All the weights are updated **after** one feedforward pass and one backward propagation/pass
- Frequency of weight update = Frequency of passes

- **Online learning**: a pass for each training instance
- **Batch learning**: a pass for a batch (a subset of training instances)
  - weight change is the sum of the changes for all the instances in the batch
- **Offline learning**: a pass for all the training instances
  - Weight change is the sum of the changes for all training instances

- Online and batch learning are stochastic gradient descent
- Offline learning is "true" gradient descent

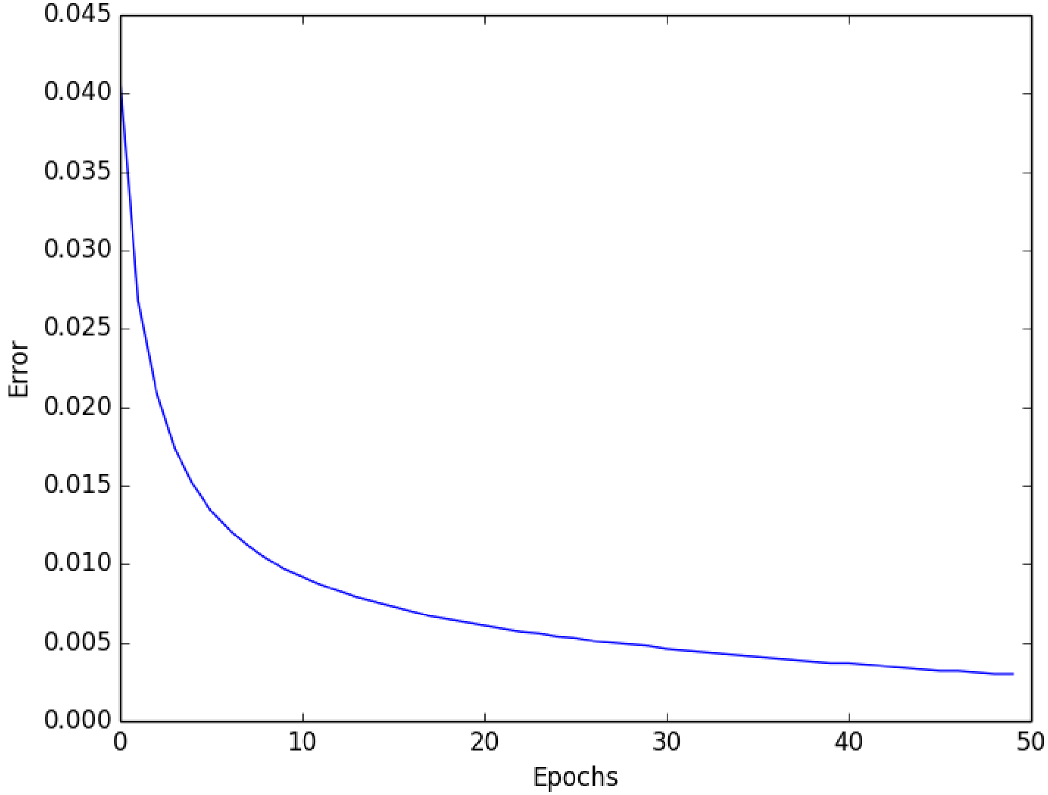
Weight Update Frequency

- Assuming a weight $w = 0.2$
- 4 training instances
- Online learning
  - Instance 1, $\Delta w = 0.1$, $w \rightarrow 0.3$
  - Instance 2, $\Delta w = 0.05$, $w \rightarrow 0.35$
  - Instance 3, $\Delta w = 0.03$, $w \rightarrow 0.38$
  - Instance 4, $\Delta w = 0.01$, $w \rightarrow 0.39$
- Offline learning
  - Instance 1, $\Delta w = 0.1$, $w = 0.2$ unchanged
  - Instance 2, $\Delta w = 0.08$, $w = 0.2$ unchanged
  - Instance 3, $\Delta w = -0.03$, $w = 0.2$ unchanged
  - Instance 4, $\Delta w = 0.05$, $w = 0.2$ unchanged
  - $w \rightarrow 0.2 + 0.1 + 0.08 - 0.03 + 0.05 = 0.4$

Weight Update Frequency

- Epoch: period when all the training instances are used once
- #Iterations = #passes

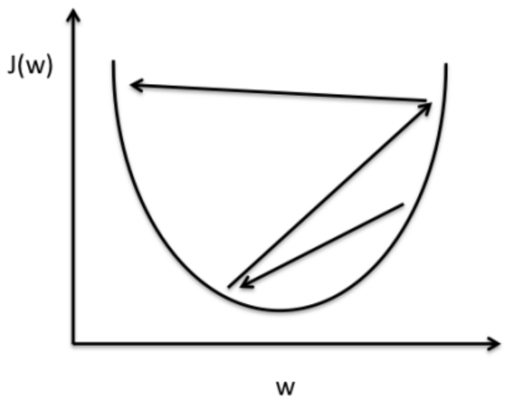
- 1000 training instances, batch size = 500, then need 2 iterations to complete **one** epoch



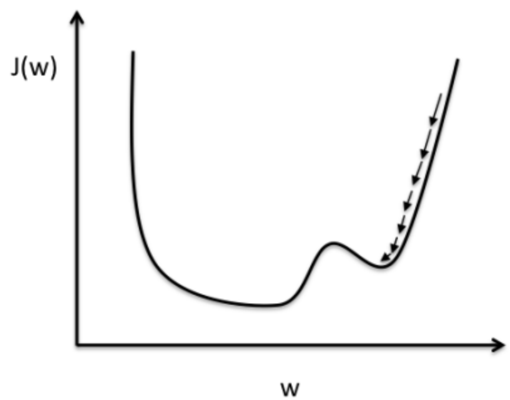
Learning Rate

- Large learning rate may cause oscillating behaviour
- Small learning rate may cause slow convergence
- 0.2 is a good starting point in practice

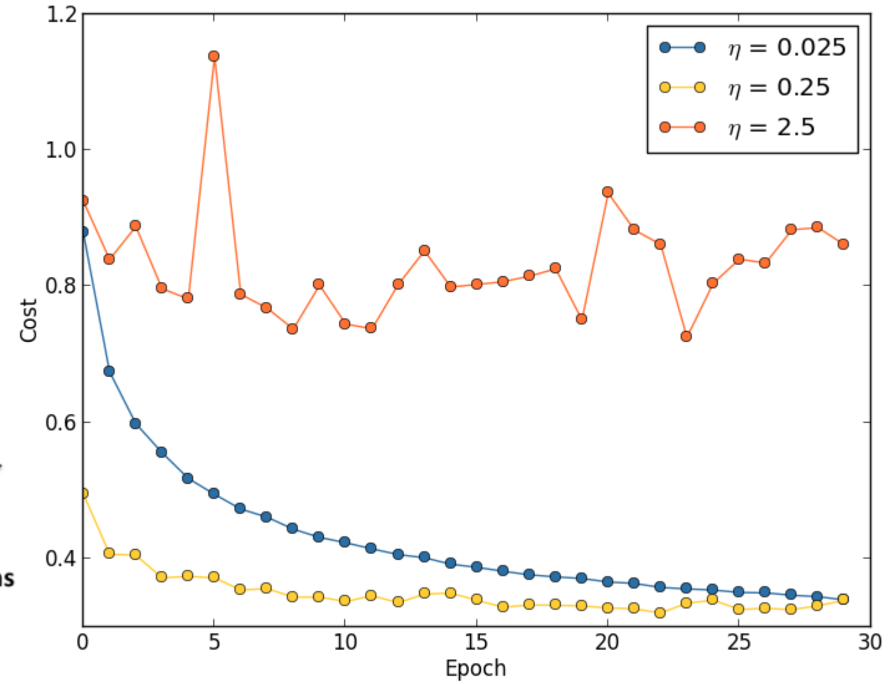
$$\boxed{\Delta w_{i \to j} = \boldsymbol{\eta} o_i o_j (1 - o_j) \beta_j}$$



Large learning rate: Overshooting.

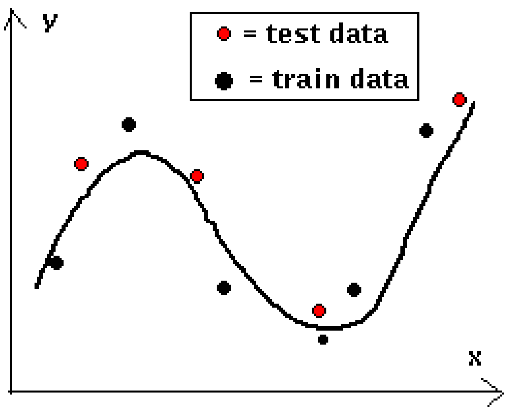


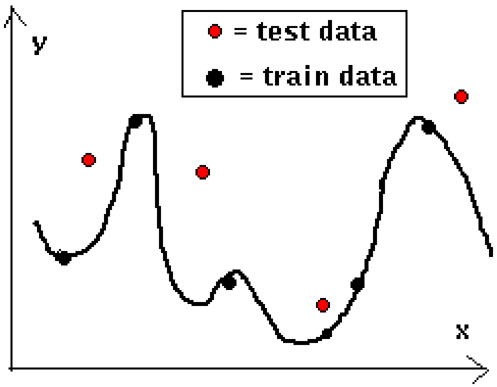
Small learning rate: Many iterations until convergence and trapping in local minima.

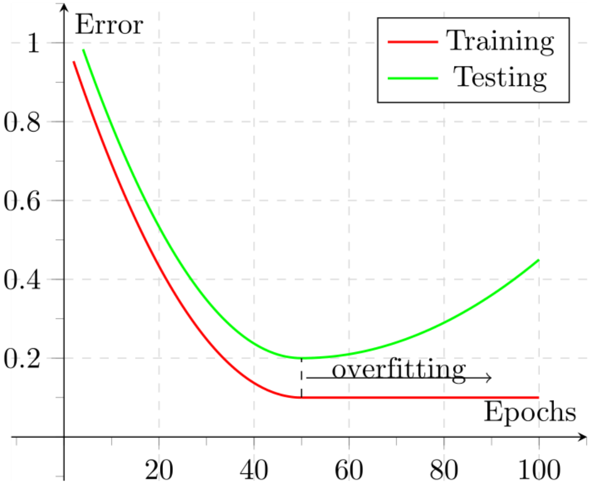


Overfitting

- Has a very high accuracy on the training set,
  but poor accuracy on the test set
- Caused by:
  - Training for too long
  - Too many weights (parameters) to train
  - Too few training instances
- The more parameters to train, the more data (training instances) we need to have an accurate estimation





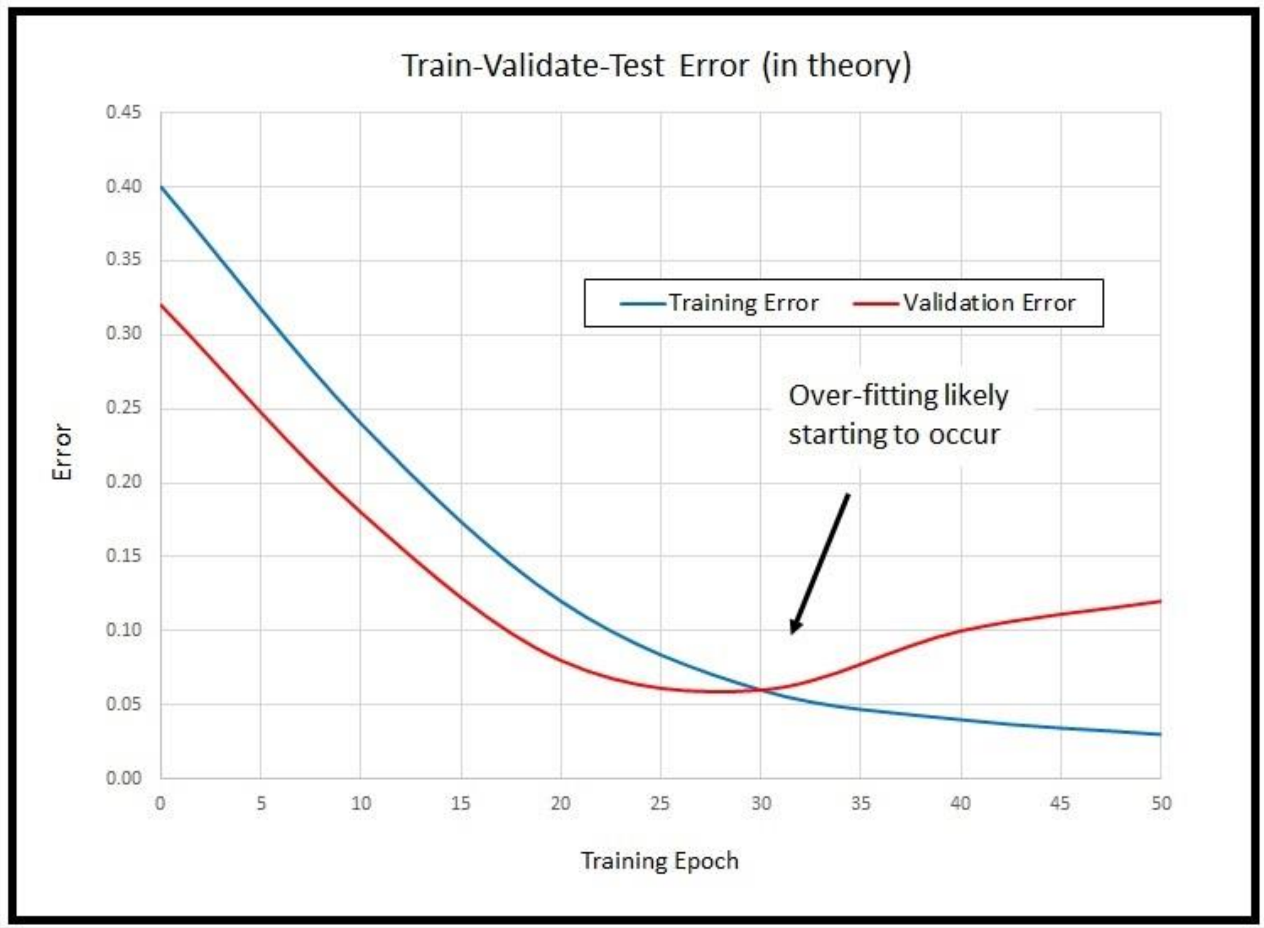


Stopping Criteria

- When a certain number of epochs is reached
- When the error (e.g. mean/total squared error) on the training set is smaller than some threshold **T**
- Proportion of correctly classified training instances (i.e. accuracy) is larger than a threshold
- Early stopping strategy
  - Validation control to **avoid overfitting**

Validation Control

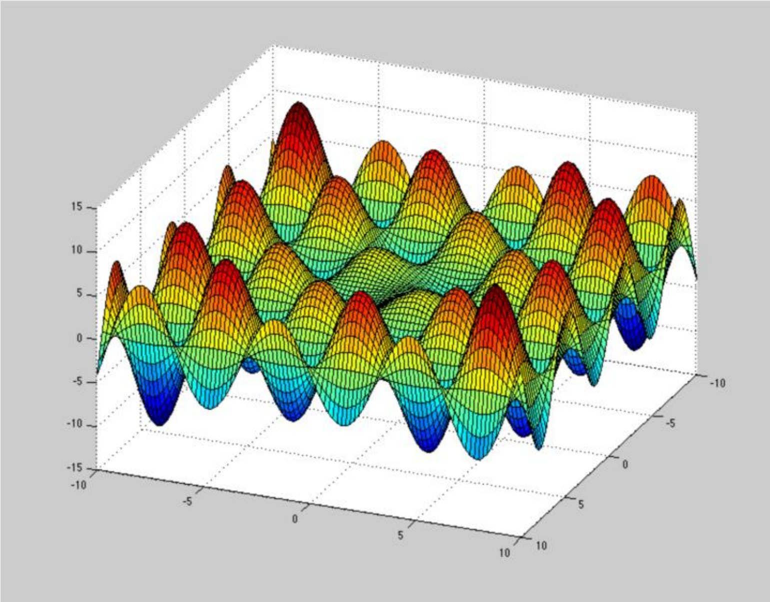
- Break the training set into two parts
- Use 1st part to compute the weight changes
- Every *m* (e.g. 10, 50, 100) epochs apply the current NN to the 2nd part (validation set) to calculate the validation error



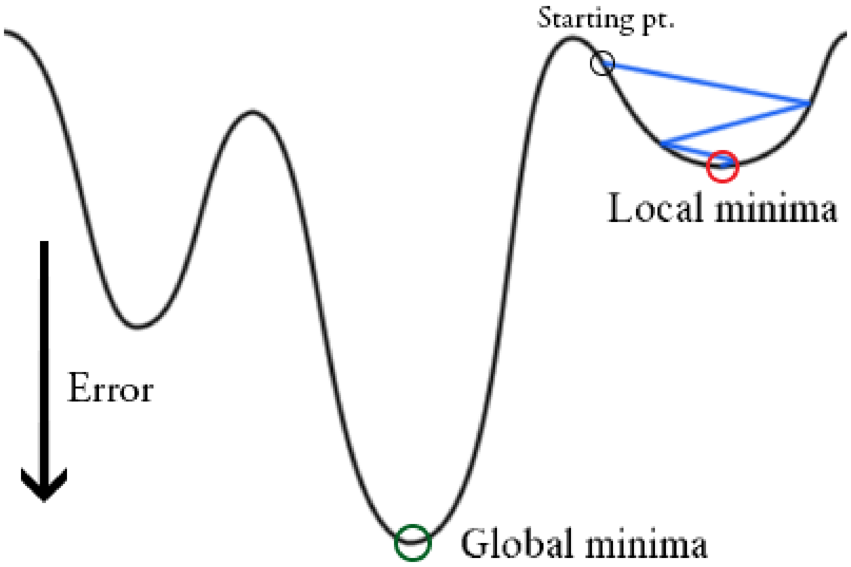
- Stop when the error on the validation set is minimum

Local Minima

- For each weight vector, we can calculate the error of the NN
- The (weight vector, error) surface/landscape can be very rugged: many local minima
- Search: a trajectory of points leading to the global minima
- A bad trajectory leads to poor local minimum



Surface with 2 weights



Surface with 1 weight

ANN Architecture

- How many input and output nodes?
  - Usually determined by the problem and data

- Number of input nodes equals the number of features

- Number of outputs
  - 1 output nodes for binary classification (true/false)
  - N output nodes for N-class classification
  - Example: (1, 0, 0) = class 1; (0, 1, 0) = class 2; (0, 0, 1) = class 3

ANN Architecture

- How many hidden layers/nodes?
  - Theorem: **one** hidden layer is enough for any problem
  - But training is significantly faster with several layers
  - Best to have as few hidden layers/nodes as possible: better *generalisation*, fewer weights to optimise (easier to solve)

  - Make the best guess you can
  - If training is unsuccessful try more hidden nodes
  - If training is successful try fewer hidden nodes
  - Observe weights after training:
    nodes with small weights can probably be eliminated

Momentum

- Normal to have huge ANNs take days/weeks/months to train
- Speeding this up is crucial!

- Momentum is a widely used approach
  - Use the gradient from last step(s)

$$\Delta w_{i\to j}(t) \leftarrow \eta o_j o_j (1 - o_j)\beta_j \boxed{+ \alpha \Delta w_{i\to j}(t-1)}$$

- Does momentum always help?
- Have you used/seen momentum before?
- How do we choose η and α?

Design Questions

- How to properly arrange the data for network training and for measuring the results?
- Number of input/output nodes?
- How many hidden layers are needed and how many nodes in each hidden layer?
- Values for the parameters and variables for controlling the training process, for example, learning rate, initial weights, momentum and number of epochs?
- Stopping criteria (validation control)?
- How often are the weights changed (batch size)?

Summary

- Different frequencies for weight update
  - Online, offline, batch learning
- Learning rate
  - Not too large nor too small, 0.2 is a good starting point
- Overfitting
- Stopping criteria
  - Validation control to avoid overfitting
- Local minima
- ANN architecture: #input/output/hidden nodes
- Momentum to speed up training