COMP307/AIML420 INTRODUCTION TO ARTIFICIAL INTELLIGENCE

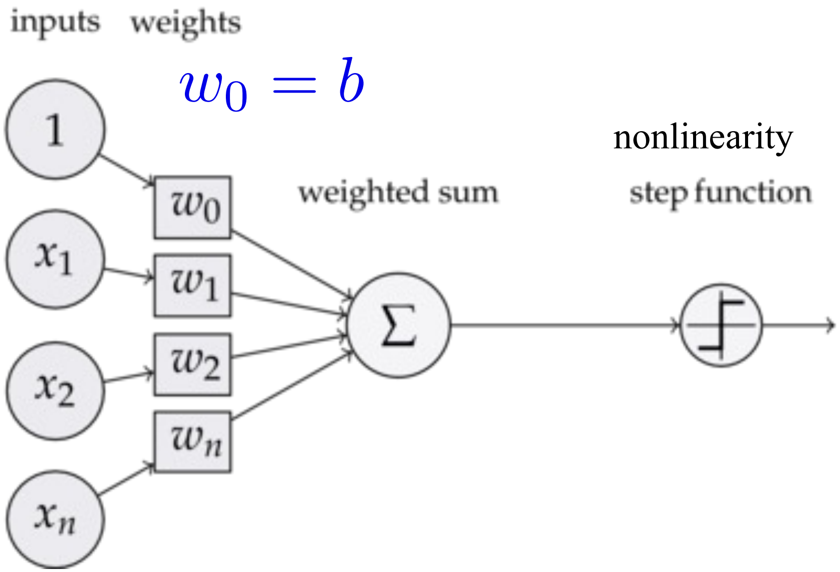
Neural Networks 2: Backpropagation

Outline

- Feed forward neural network
- Back propagation to train neural network

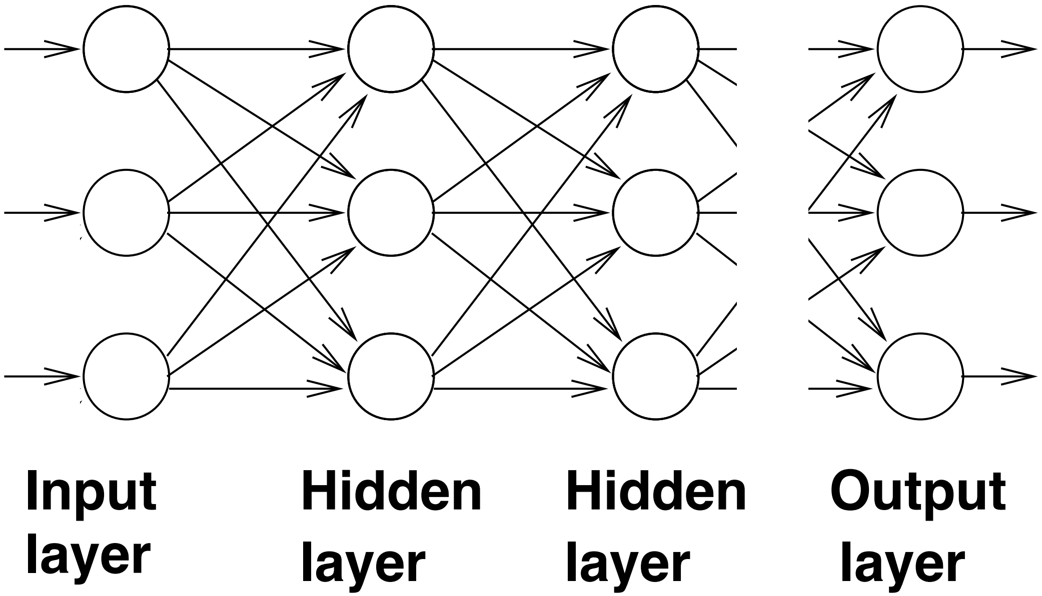
Neuron

- Generally real-valued input (denoted s_i for input i)
- Generally real-valued output (denoted y)
- Weights w_i
- Activation function $a(\cdot)$
- $y = a(\sum_{i=1}^{n} w_i s_i + b)$



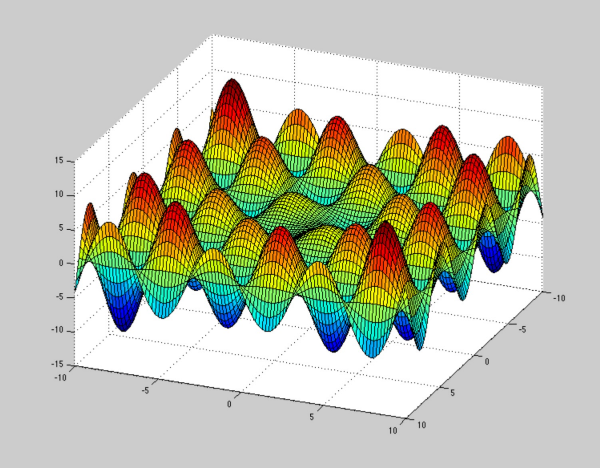
Feedforward Neural Network

- MLP / feedforward network
  - Referred to as "fully connected" layer / neural network
  - Multiple (hidden) layers, many nodes in each layer
  - No jump connections
  - Each node connects to all nodes of adjacent layers
    - Very many weights (parameters): one per link



Learning Network Parameters

- Have a database of inputs and outputs
- Find MLP parameters that mimic input / output relation
- A complex optimisation problem



$$W^* = \underset{W}{\operatorname{argmin}} f(W)$$

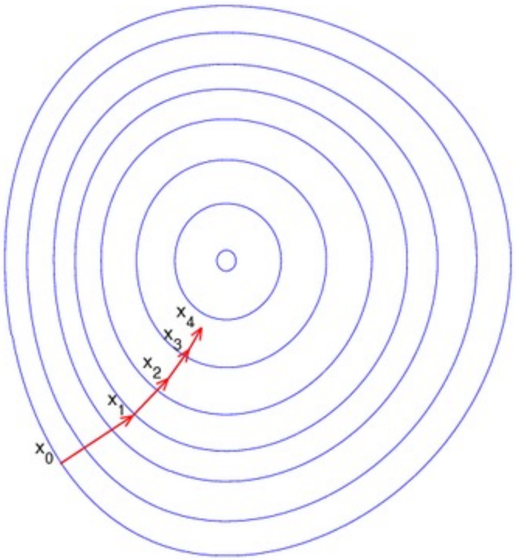
- f is objective function = loss function
- W^* is the W that minimizes $f(W)$
- f usually non-convex (many local optima)
- Extremely high dimensional
- Impossible to solve using analytic methods
- Must use numerical methods

Optimisation=Learning ANN Weights

- Gradient descent
  - Compute gradient $\nabla_W f(W)$
  - Small steps in direction $-\nabla_W f(W)$
  - Example visualization

$$\nabla_W f(W) = \begin{pmatrix} \frac{\delta\, f(W)}{\delta W_0} \\ \vdots \\ \frac{\delta f(W)}{\delta W_n} \end{pmatrix}$$

- Stochastic gradient descent
  - Divide database into *batches*
  - Define surrogate $f(W)$ for each subsequent batch
  - Jittery descent for “true” $f(W)$
  - Quicker and perhaps also better



- Context:
  - Simulated annealing
  - Tabu search
  - Evolutionary computation

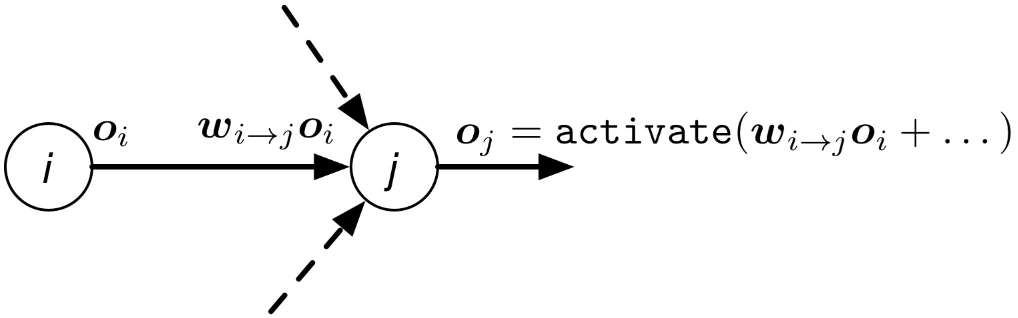
Back Propagation (BP) Algorithm

- Gradient descent on network parameters

- **Input:** data (input-output pair examples) $\{(s, d^{(s)})\}$
- **Initialise** network weights (parameters) W
- **Repeat** until stop condition:
  - **Feedforward**
    - For each example z, calculate the network output o_z with current weights
    - Calculate the average loss (objective) function as a function of parameters, $J(W)$, over batch of z
  - **Back propagation**
    - Estimate the **gradient** of the loss to each individual weight w_i
      - How much the loss will be reduced by changing the weight
    - Change each individual weight proportional to minus its gradient
      - *The gradients are computed backwards (from the last layer to the first layer) in one sweep for all weights, tp reduce computational effort*
- **Output:** *updated network weights W*

Back Propagation (BP) Algorithm

- How to calculate contribution of $w_{i\to j}$ to the loss function?



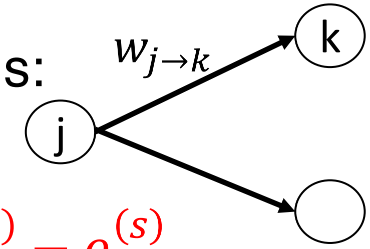
- When changing $w_{i\to j}$ by tiny amount $dw_{i\to j}$, the loss change d_loss should be proportional to $o_i \times slope_j \times \beta_j \times dw_{i\to j}$:
  - Proportional to $dw_{i\to j}$ itself
  - Proportional to the previous-neuron output: o_i
  - Proportional to slope of the activation function at node j: $slope_j$
  - Proportional to slope of the error as a function of neuron output o_j: β_j

$$\begin{tikzcd}
w_{i\to j} \arrow[r, bend right] & w_{i\to j}o_i \arrow[r, bend right] & o_j = act(w_{i\to j}o_i + \cdots) \arrow[rr, bend right] & \ldots & loss
\end{tikzcd}$$

Back Propagation (BP) Algorithm

- Previous slide says $d\_loss = \ o_i \times slope_j \times \beta_j \times dw_{i\rightarrow j}$
  - Hence gradient is $\frac{\partial\, loss}{\partial w_{i\rightarrow j}} = o_i\ slope_j\ \beta_j$
- $slope_j$ is derivative of activation function
- β_j, slope of the error as a function of neuron output o_j is recursive in the layers:
  - Recursion adding contributions of downstream neurons:

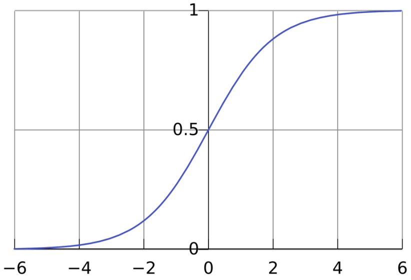
$$\beta_j = \sum_k w_{j\rightarrow k} \times slope_k \times \beta_k$$



  - Output layer: for squared error loss/objective: $\beta_k = d_k^{(s)} - o_k^{(s)}$

- Wikipedia page has nice explanation (requires calculus)

Simple example backpropagation



- Assumptions for our case:
  - Activation function sigmoid

    (*not* so good, but analytically tractable)

    $$slope_j = o_j(1 - o_j)$$
  - Have data input-output data pairs $\{(s, d^{(s)})\}$
  - Squared error (L2) loss function

$$\text{Loss} = \sum_{s \in data} \left\| d^{(s)} - o^{(s)} \right\|^2 = \sum_{s \in data} \sum_{i \in vector\ elements} \left(d_i^{(s)} - o_i^{(s)} \right)^2$$

- Back propagation:
  - Output node k: $\beta_k = d_k^{(s)} - o_k^{(s)}$ (slope/derivative of loss)
  - Hidden node j: $\beta_j = \sum_k w_{j\to k} o\_k(1 - o_k)\beta_k$
  - Update: $\Delta w_{j\to k} \propto - o_j \times slope_k \times \beta_k = - o_j o_k (1 - o_k) \beta_k$

Simple BP example: Implementation

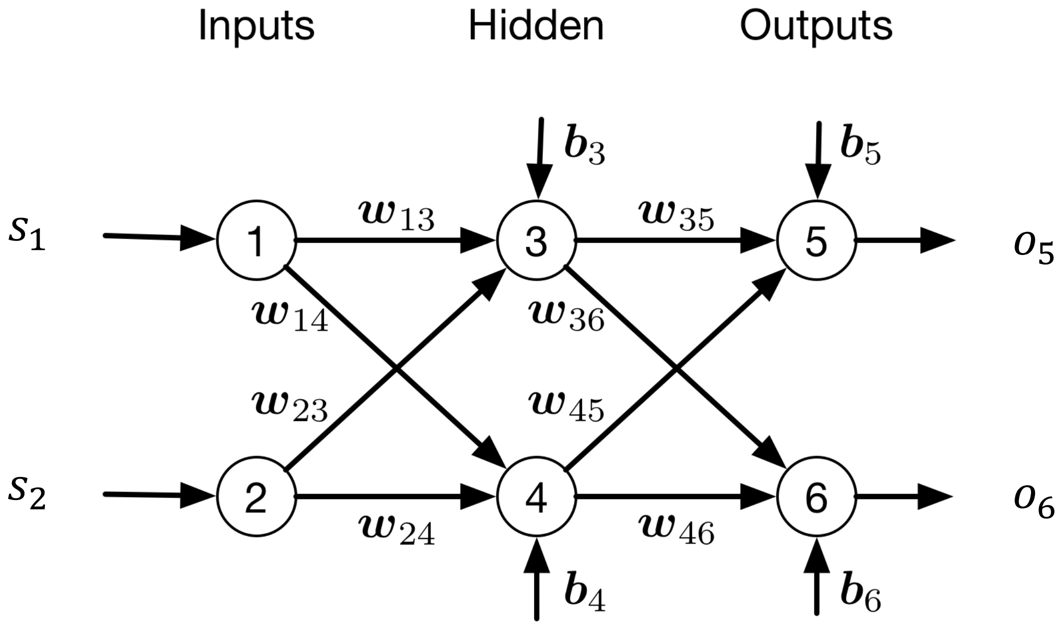
- For assumptions of previous slide (includes not-so-good sigmoid)

- Have data set $\{(s, d^{(s)})\}$
- Set learning rate η
- Set weights (parameters) W to random values
- Repeat until stop condition:
  - For all pairs in batch
    - Feed forward pass to get network outputs $o^{(s)}$
    - Backward pass:
      - Compute $\beta_k = d_k^{(s)} - o_k^{(s)}$ for each output node
      - Compute $\beta_j = \sum_k w_{j\to k} o_k (1 - o_k) \beta_k$
      - Compute the weight changes $\Delta w_{j\to k} = -\eta o_j o_k (1 - o_k) \beta_k$
  - Add weight changes for all data in batch
  - Change weights by scaled weight-change sum

BP Algorithm Example

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

Inputs		Outputs	
s_1	s_2	o_5	o_6



Automatic differentiation

- In the real world, no-one does backpropagation “manually”
- Automatic differentiation evaluates the gradient of a function:
  - Applies the chain-rule and evaluates the result
  - Provides gradient (in numbers) for current input-output batch
  - JAX example (differentiate to W and b)
    - Let loss function be: loss_fn(W, b, s)
    - We select the first two arguments (0,1) to differentiate to:

      W_grad, b_grad = grad(loss_fn, argnums=(0,1))(W, b, s)

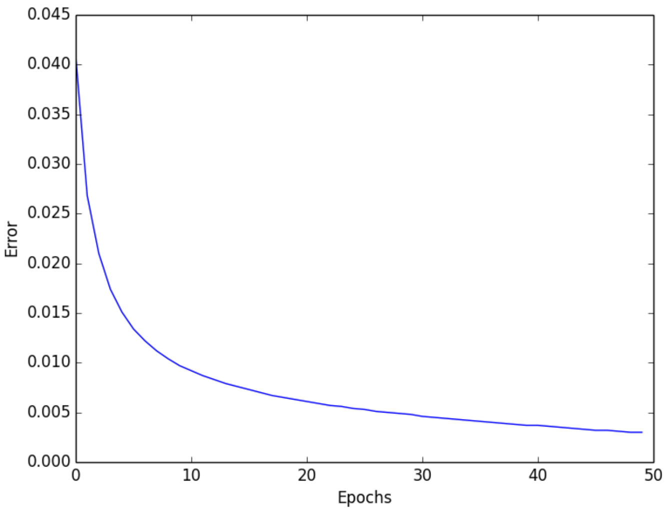
- Related but different:
  - *Symbolic differentiation* is aimed at obtaining explicit symbolic expressions (e.g., Mathematica or Maple)
  - *Numerical differentiation / finite-difference methods* do not require explicit derivatives, but are less exact

Backpropagation: automatic differentiation

```
# Example using JAX
def loss_fn(params, s, target):
  y = fcnn(params, s)
  loss = jnp.mean( jnp.square(target-y))
  return loss, loss

# main program (network to approximate multiplication of input given matrix)
rng = random.PRNGKey(3)
matrix = jnp.array([[10.,0.],[7.,10.]])  # for data generation
layerDims = jnp.array((2,3,2))
batchno = 1001
batchsize = 10

params, rng = init_fcnn(layerDims, rng)
optimiser = optax.adam(learning_rate=0.01)
opt_state = optimiser.init(params)

for batch in range(batchno):
  s, target, rng = gendata2(batchsize, matrix, rng)  # generate matrix multiply input-output data
  grads, loss = jax.grad(loss_fn, argnums=0, has_aux=True)(params, s, target)     # differentiate to params
  updates, opt_state = optimiser.update(grads, opt_state, params)
  params = optax.apply_updates(params, updates)

  if (batch % 100) == 0:
      print('batch', batch, 'training loss', loss)
```

Notes on BP Algorithm

- *Epoch*: all input examples (entire training set)
- Training may require thousands of epochs. A convergence curve will help to decide when to stop
  - Split data into *training data, validation data, test data*
  - Use validation data to decide when to stop
  - Don't use test data during ablation studies



Notes on BP Algorithm

- Squared error is just one objective function
  - Good choice for regression (but not only choice)
  - *Not* a good choice for classification
- Stochastic gradient descent: optimise over *batches* of data
  - Faster and better
- Automatic differentiation
  - Works for any architecture
- Data:
  - Training data: examples you train on (divide into batches)
  - Validation data: data not part of training data that you use to make the convergence curve.
  - Test data: what you use at the end to evaluate performance; keep these separate so as not to bias your methodology

Summary

- MLP = fully connected neural network
- Back propagation
  - Gradient descent
  - Feedforward then error back propagation -> weight update
  - In practice we always use automatic differentiation
  - Wikipedia page
  - History (not unbiased)