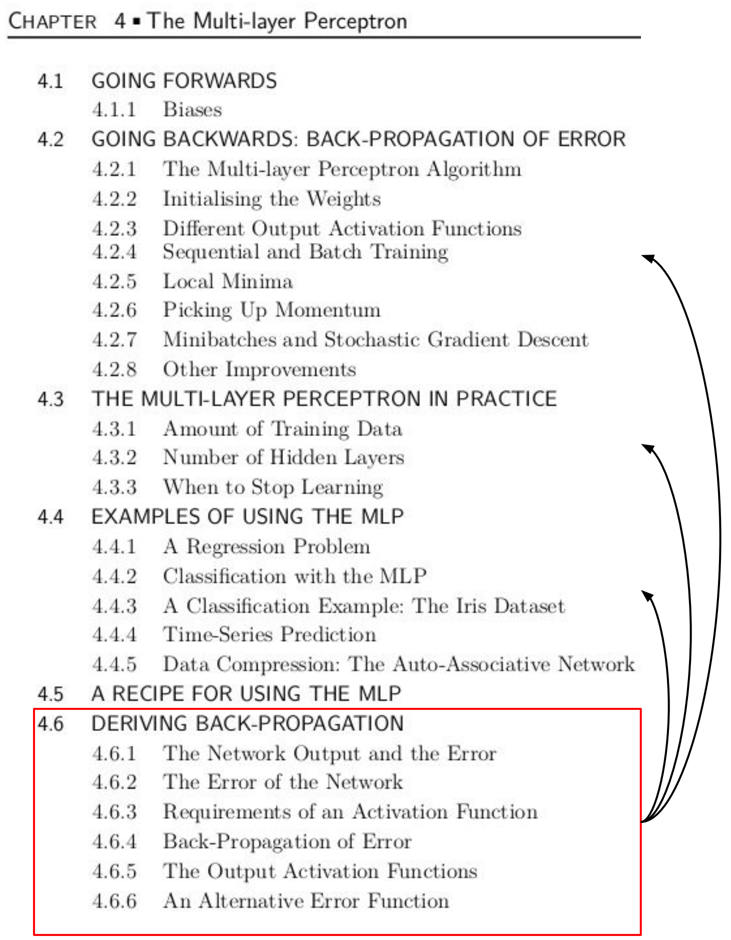
Chapter 4: The Multi-layer Perceptron

Connor Clarkson & Ben Lloyd-Roberts

Book Layout

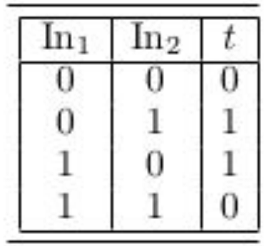
- Order of slide content varies a little from the textbook.
- Explanations of components toward last section
- Definitions scattered throughout their relevant sections



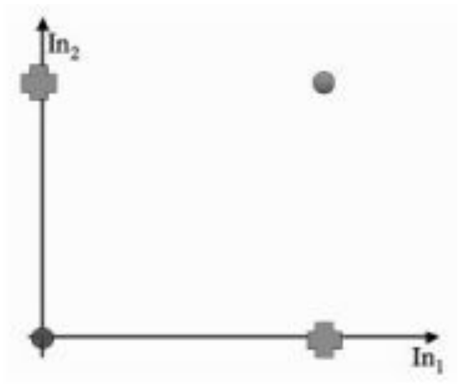
CHAPTER 4 ▪ The Multi-layer Perceptron

Content

Towards Multi-layer Perceptron: XOR Problem

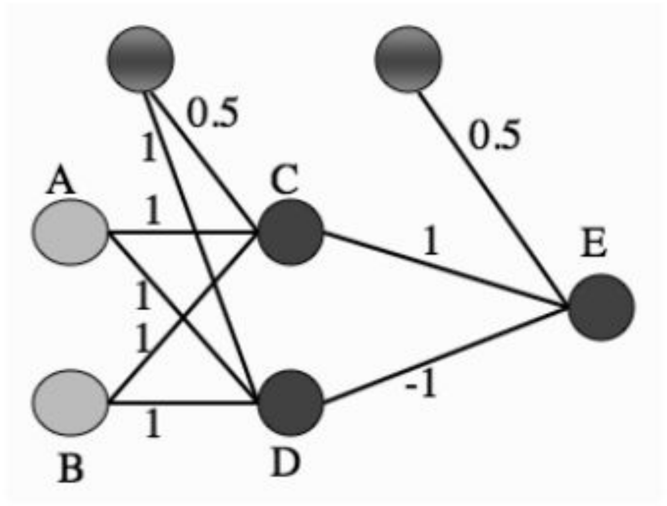


In_1	In_2	t
0	0	0
0	1	1
1	0	1
1	1	0



- Single perceptron fail under non-linearly separable data
- 3 different plans of attack:
  - Add more layers
  - Change bias
  - Change activation functions

Multi-layer Perceptron



$$\begin{cases} 1 & \text{if } \sum_{i=0}^{m} w_{ij}x_i > 0 \\ 0 & \text{if } \sum_{i=0}^{m} w_{ij}x_i \leq 0 \end{cases}$$				
A	**B**	C	D	**E**
0	0	0	0	**0**
0	1	1	0	**1**
1	0	1	0	**1**
1	1	1	1	**0**

- We can add more neurons between the input nodes and the output nodes creating a more complex model
    - Intuitively more weights may increase capacity to learn
- New problems arise:
    - Which weights were wrong?
    - Number of hidden layers
    - Weight Initialisation

There and Back Again; Forward Pass

- Back-propagation allows us to find the weights of the neurons that got the output wrong. While gradient descent allows us to update the neurons in such a way that minimises the error. We first need to perform a forward pass then calculate the error of the network.
- The forward pass is same as the Perceptron except we compute the output layer by layer.
  - We start by filling in the values for the inputs then compute the weighted sum for each node in the layer. Each weighted sum is passed through an activation function to determine if the node should fire
  - We then use the output activations of the first layer to compute the weighted sum for each node in the second layer. After passing it through the activation functions we get our output

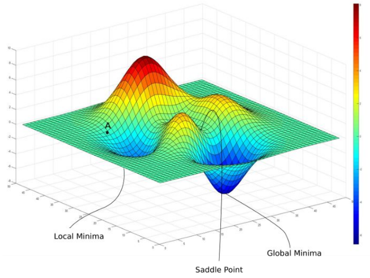
There and Back Again; Error of Network

- The output of the network is a function of three things:
    - The current input (x)
    - The activation function g(.)
    - The weights of the network
- In order for the network to improve its performance we need to change the weights of each neuron for each layer. But in order to know how much we need to change the weights by we must compute the error of the network.
- The chosen error function is sum-of-squares error - which calculates the difference between the prediction (y_k) and the target (t_k) for each node (k), squares them, and adds them together

$$\begin{aligned} E(\mathbf{w}) &= \frac{1}{2}\sum_{k=1}^{N}(y_k - t_k)^2 \\ &= \frac{1}{2}\sum_{k=1}^{N}\left[g\left(\sum_{j=0}^{M} w_{jk}a_j\right) - t_k\right]^2 \end{aligned}$$

There and Back Again; Gradient Descent

- We use gradient descent to adjust the weights in the direction of the negative gradient of the error function
- We want to find how the error function changes with respect to the different weights, then apply each change to each of the weights
- However there are 2 problems here
  - For the output neurons, we do not know the inputs
  - For the hidden neurons, we do not know the targets. For extra hidden layers, we know neither the input nor the targets
- To solve this we need to back-prop through the network to find what weight(s) caused the error.
- Back-prop uses the chain rule to calculate the derivatives of composite functions. Meaning we can compute the derivative of hidden neurons.



There and Back Again; Backward Pass

- In order to compute the derivative of the error function w.r.t weights we first need to compute the derivative of the error w.r.t weighted sum of each neuron then compute the derivative of the weighted sum w.r.t weights.

$$\frac{\partial E}{\partial w_{\zeta\kappa}} = \frac{\partial E}{\partial h_\kappa}\frac{\partial h_\kappa}{\partial w_{\zeta\kappa}}$$

E - error function
w - weights
h - weighted sum
ζ - hidden-layer neuron index
κ - output-layer neuron index

$$\delta_o(\kappa) = \frac{\partial E}{\partial h_\kappa} = \frac{\partial E}{\partial y_\kappa}\frac{\partial y_\kappa}{\partial h_\kappa}.$$

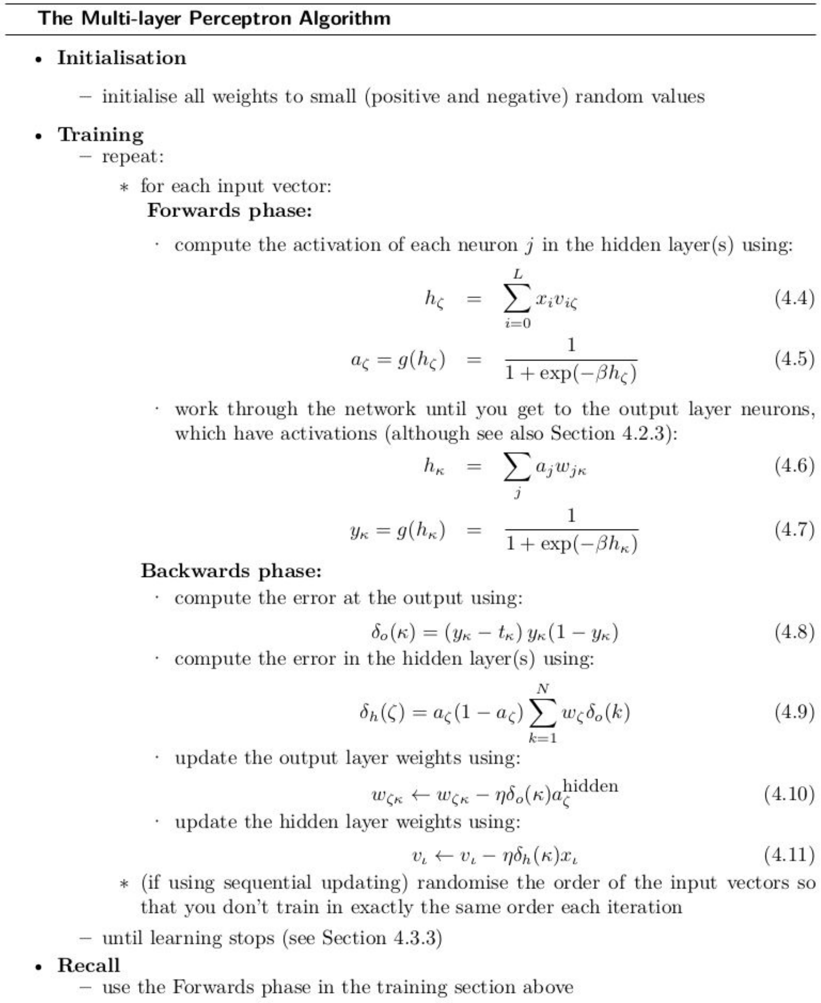
$$y_\kappa = g(h_\kappa^{\text{output}}) = g\left(\sum_{j=0}^{M} w_{j\kappa} a_j^{\text{hidden}}\right)$$

$$\begin{aligned}
\delta_o(\kappa) &= \frac{\partial E}{\partial g\left(h_\kappa^{\text{output}}\right)}\frac{\partial g\left(h_\kappa^{\text{output}}\right)}{\partial h_\kappa^{\text{output}}} \\
&= \frac{\partial E}{\partial g\left(h_\kappa^{\text{output}}\right)} g'\left(h_\kappa^{\text{output}}\right) \\
&= \frac{\partial}{\partial g\left(h_\kappa^{\text{output}}\right)}\left[\frac{1}{2}\sum_{k=1}^{N}\left(g(h_k^{\text{output}}) - t_k\right)^2\right] g'\left(h_\kappa^{\text{output}}\right) \\
&= \left(g(h_\kappa^{\text{output}}) - t_\kappa\right) g'(h_\kappa^{\text{output}}) \\
&= (y_\kappa - t_\kappa) g'(h_\kappa^{\text{output}}),
\end{aligned}$$

$$\begin{aligned}
\frac{\partial h_\kappa}{\partial w_{\zeta\kappa}} &= \frac{\partial \sum_{j=0}^{M} w_{j\kappa} a_j}{\partial w_{\zeta\kappa}} \\
&= \sum_{j=0}^{M} \frac{\partial w_{j\kappa} a_j}{\partial w_{\zeta\kappa}} \\
&= a_\zeta.
\end{aligned}$$

The Multi-layer Perceptron Algorithm

1. Input vector is put into the input nodes
2. The inputs are fed forward through the network
    a. Compute output of first hidden layer
    b. Compute output layer
3. The error is computed as the sum-of-squares difference between the network outputs and the targets
4. The error is fed backwards through the network
    a. Frist update the second layer weights
    b. Then update the first layer weights



The Multi-layer Perceptron Algorithm

- **Initialisation**
  - initialise all weights to small (positive and negative) random values
- **Training**
  - repeat:
    - ∗ for each input vector:

      Forwards phase:
      - · compute the activation of each neuron j in the hidden layer(s) using:

$$h_\zeta = \sum_{i=0}^{L} x_i v_{i\zeta} \tag{4.4}$$

$$a_\zeta = g(h_\zeta) = \frac{1}{1+\exp(-\beta h_\zeta)} \tag{4.5}$$

      - · work through the network until you get to the output layer neurons, which have activations (although see also Section 4.2.3):

$$h_\kappa = \sum_j a_j w_{j\kappa} \tag{4.6}$$

$$y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)} \tag{4.7}$$

      Backwards phase:
      - · compute the error at the output using:

$$\delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa(1-y_\kappa) \tag{4.8}$$

      - · compute the error in the hidden layer(s) using:

$$\delta_h(\zeta) = a_\zeta(1-a_\zeta)\sum_{k=1}^{N} w_\zeta \delta_o(k) \tag{4.9}$$

      - · update the output layer weights using:

$$w_{\zeta\kappa} \leftarrow w_{\zeta\kappa} - \eta\delta_o(\kappa) a_\zeta^{\text{hidden}} \tag{4.10}$$

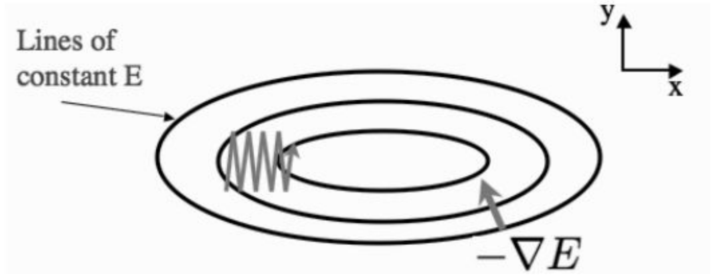
      - · update the hidden layer weights using:

$$v_\iota \leftarrow v_\iota - \eta\delta_h(\kappa) x_\iota \tag{4.11}$$

    - ∗ (if using sequential updating) randomise the order of the input vectors so that you don't train in exactly the same order each iteration
  - until learning stops (see Section 4.3.3)
- **Recall**
  - use the Forwards phase in the training section above

Optimisation

- Error minimisation is the driving force of learning, meaning we are **optimising** the error function.
- Done by approximating the negative gradient of the error function (ideally) towards a **global minimum.**
- Following the negative gradient through a series of steps likely leads to a local minimum where a lower point may exists elsewhere which isn't 'visible' from the current point.
- Computationally less expensive to compute loss based on steps.



- Step sizes must then be adjusted to reach the minimum

Small steps - slower training, less likely to miss significant changes in the error

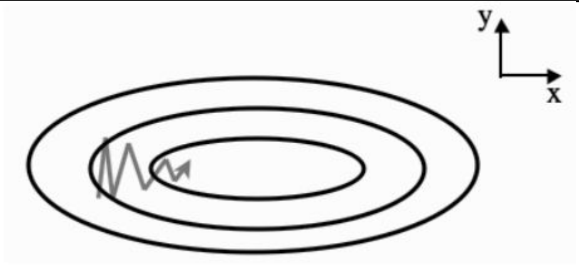
Momentum

- We may have discovered a local minimum where the global could be a few steps over some local maximum.
- Valley traversal analogy - a weighted object moving downwards means we can build **momentum**.

- Add some contribution from the previous weight to the updated weight. Two or more dimensions means more direct movement towards the minimum with reduced influence from the local changes.

$$w_{\zeta\kappa}^{t} \leftarrow w_{\zeta\kappa}^{t-1} + \eta\delta_o(\kappa)a_{\zeta}^{\text{hidden}} + \alpha\Delta w_{\zeta\kappa}^{t-1},$$

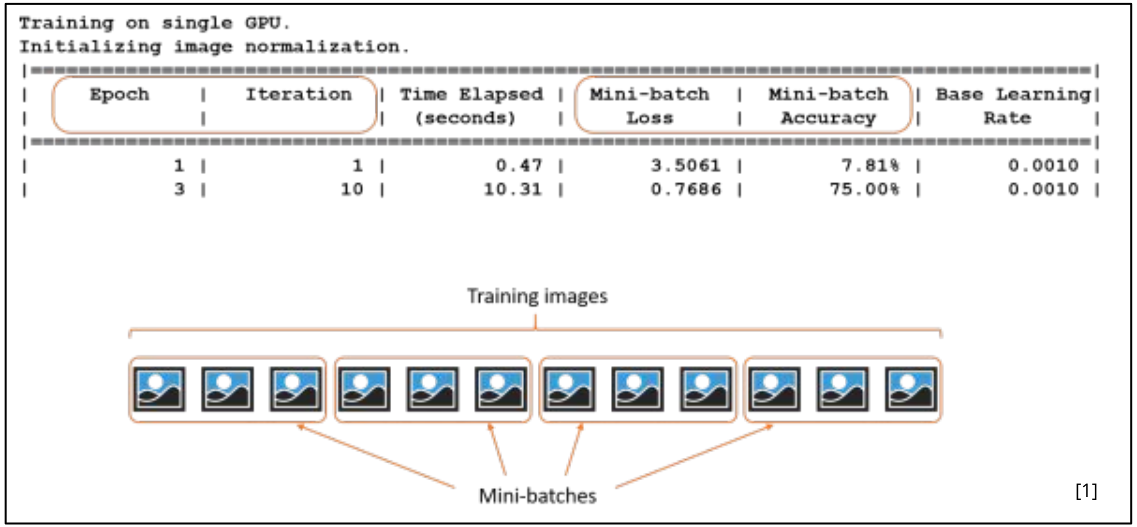
t	: the current update
t-1	: the previous update
$\Delta w_{\zeta\kappa}^{t-1}$	: the previous weight update
$0 < \alpha < 1$	: the momentum constant (typically 0.9)



- Can also include **weight decay** ($0 < \epsilon < 1$), to reduce the size of the weights as number of iterations increases.
- Value chosen experimentally, can make learning significantly worse as a result.
- Small weights are better since they produce near linear networks

Mini-batch Training

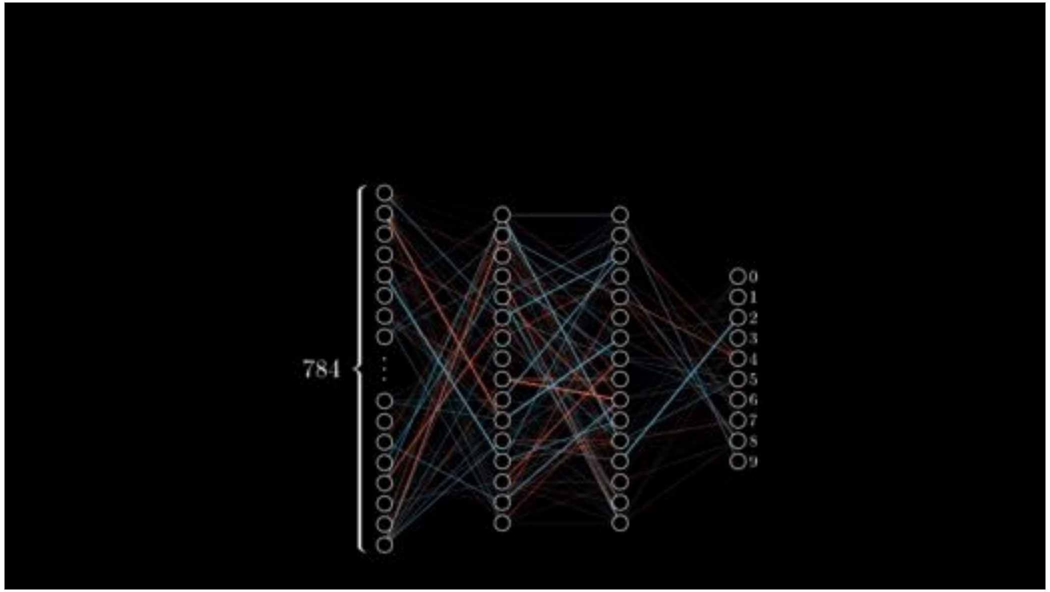
1. Split the training set into random batches
2. Estimate the gradient based on one of the subsets of the training set
3. Perform a weight update
4. Using the next subset, estimate a new gradient and use that for the weight update
5. Repeat until all of the training set has been used.



```
Training on single GPU.
Initializing image normalization.
|===========================================================================================|
|     Epoch     |   Iteration   | Time Elapsed |  Mini-batch  |  Mini-batch  | Base Learning|
|               |               |  (seconds)   |     Loss     |   Accuracy   |     Rate     |
|===========================================================================================|
|             1 |             1 |         0.47 |       3.5061 |        7.81% |       0.0010 |
|             3 |            10 |        10.31 |       0.7686 |       75.00% |       0.0010 |
```

[1] https://matlab4engineers.com/wp-content/uploads/2017/10/minibatches-600x276.png

Properties of MLP



Initialising Weights

- Weights randomly initialised to small positive/negative values

- Random initialisation and similarly sized weights supports **uniform learning;**

    - *Intention to finish updating our weights at the same time*

- Learning begins at different points for each initial run of the network

- If inputs have uniform variance - typically input to neuron will be:

  $$w\sqrt{n},$$

  Where n is the number of input neurons.

- Therefore, a general range for weights:

  $$-1/\sqrt{n} < w < 1/\sqrt{n},$$

  Making the total input to a single neuron approx. 1

Activation Functions

An activation function determines whether a neuron will fire according to some threshold θ, given the weighted sum from the previous layer.

Activations must be:

- *Differentiable* - to compute the gradient
- *Saturates* - at both ends of the output range
- *Sensitive* - to small changes between saturation points

Sigmoid Functions

- Sigmoid, a family of S-shaped activation functions, satisfy the previous criteria.
- Introduces neuron fire/don't fire pattern

Generally of the form: $a = g(h) = \frac{1}{1 + \exp(-\beta h)}$

Where β is some positive parameter.

- Input domain of all real numbers
- Return value (commonly) in the range [0,1] or [-1,1]

$$\begin{aligned}
g'(h) = \frac{dg}{dh} &= \frac{d}{dh}(1 + e^{-\beta h})^{-1} \\
&= -1(1 + e^{-\beta h})^{-2}\frac{de^{-\beta h}}{dh} \\
&= -1(1 + e^{-\beta h})^{-2}(-\beta e^{-\beta h}) \\
&= \frac{\beta e^{-\beta h}}{(1 + e^{-\beta h})^2} \\
&= \beta g(h)(1 - g(h)) \\
&= \beta a(1 - a)
\end{aligned}$$

Output Activations

- Previously seen sigmoid used for activation in hidden and output layers
    - Fine for classification, not so much for regression
- Need outputs within a continuous range, between 0 and 1.
- Replace sigmoid neurons at output layer with linear nodes to use the sum of their input

$$a = g(h) = \frac{1}{1 + \exp(-\beta h)} \longrightarrow g(h) = h$$

- Output nodes no longer represent neuron fire/don't fire pattern. This alteration allows us alter the MLP according to regression problems.

Softmax

- Commonly used activation for output neurons in classification problems using the 1-of-*N* encoding.
- Rescales outputs by calculating the exponential of the inputs, dividing by the sum of the inputs to all of the neurons so the activations sum to 1 and all lie in [0,1].
- Changes in the activation function means a change in its derivative, therefore the learning rule will be different.

Activation

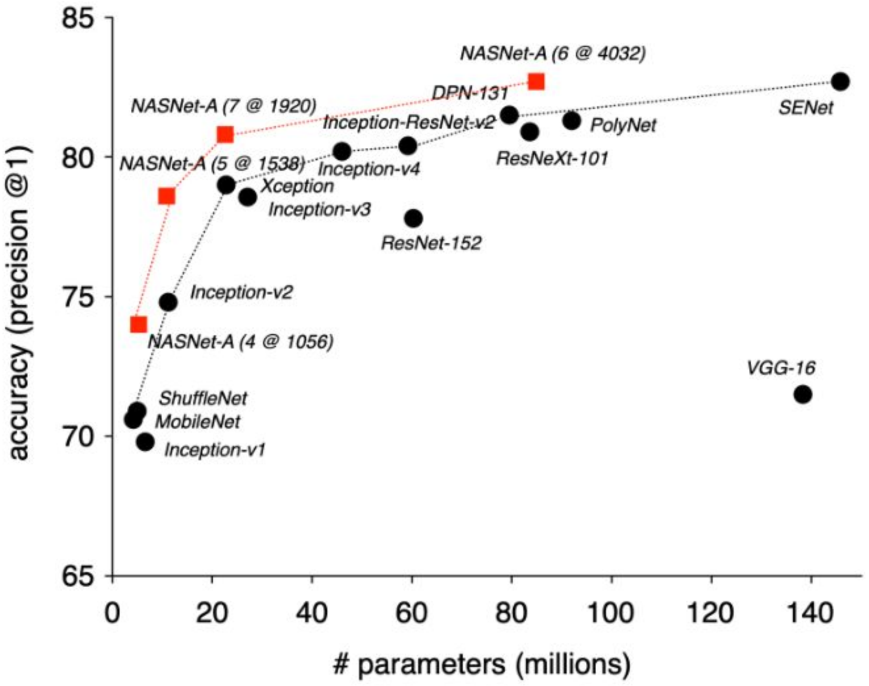
$$\left\{ y_\kappa = g(h_\kappa) = \frac{1}{1+\exp(-\beta h_\kappa)} \longrightarrow y_\kappa = g(h_\kappa) = \frac{\exp(h_\kappa)}{\sum_{k=1}^{N} \exp(h_k)}. \right\}$$

Output Error

$$\left\{ \delta_o(\kappa) = (y_\kappa - t_\kappa)\, y_\kappa (1 - y_\kappa) \longrightarrow \delta_o(\kappa) = (y_\kappa - t_\kappa) y_\kappa (\delta_{\kappa K} - y_K) \right\}$$

- Where δ□κ = 1 if □ = κ and 0 otherwise

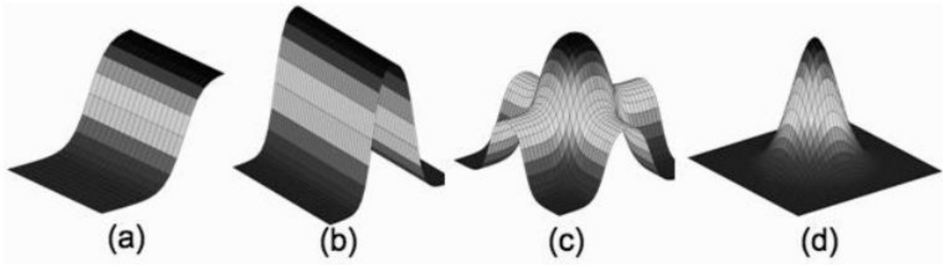
Amount of Training Data



- Number of weights for a MLP with one hidden layer:
  - (L +1) x M + (M + 1) x N, where L, M, N are the number of nodes in the input, hidden, and output layers
- Rule of thumb: Use 10 times as many training examples as their are weights

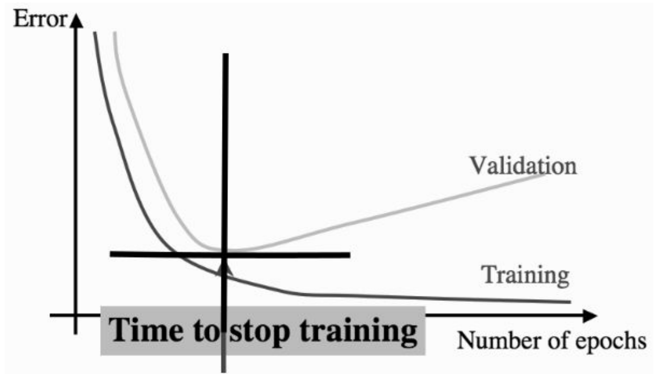
Number of Hidden Layers

- Under the Universal Approximation Theorem we can show that we never normally need more than 2 layers.
  - Its possible to approximate any smooth functional mapping using only linear combinations of localised sigmoidal functions
- The idea is that we combine sigmoid functions (a) to generate ridge-like functions (b). We combine ridge-like functions to generate functions with a unique maximum (c). Transforming these functions with another layer allows use to obtain a localized response (d). We combine these responses in the output layer, resulting in learning local representation of individual inputs



Early Stopping Criteria

- Training requires running over the entire dataset many times, changing the weights as MLP makes errors in each iteration. How many times do we iterate over the dataset?
- if we train for N iterations we run the risk that the MLP overfits or the MLP did not sufficiently learn by the N'th iteration. Instead we can stopping training when some predefined minimum error is reached.



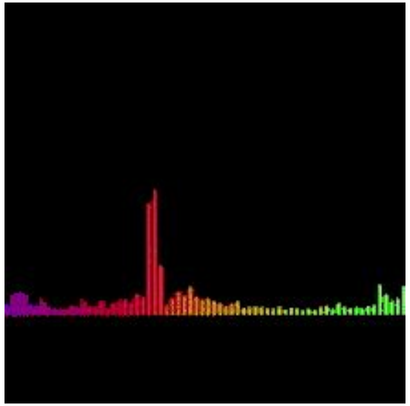
Summary of MLP

- Select inputs and outputs for your problem
  - Input vectors and target outputs from data related to the problem.
  - Feature selection
  - Output encodings
- Normalise inputs
  - Rescale the data by subtracting the mean value from each element in the input vector and divide by the variance.
- Split the data into training, testing and validation sets
  - Train the network, validate parameters, test network
- Select a network architecture
  - Inputs and outputs should be known - number of hidden layers/nodes to be selected
- Train a network
  - Apply the MLP algorithm to data - early stopping, etc
- Test the network
  - Show unseen samples to the network and measure the performance

Using MLP in Different Contexts

Regression Problems

Classification Problems


Time-series Problems

Auto-associative Problems

Regression Problems

- Generate a dataset by some mathematical function then learn that function so that we can find values for any input not just in the dataset

```
x  = np.ones((1,40))*np.linspace(0,1,40)
t  = np.sin(2*np.pi*x) + np.cos(4*np.pi*x) + np.random.randn(40)*0.2
x  = x.T
t  = t.T
```

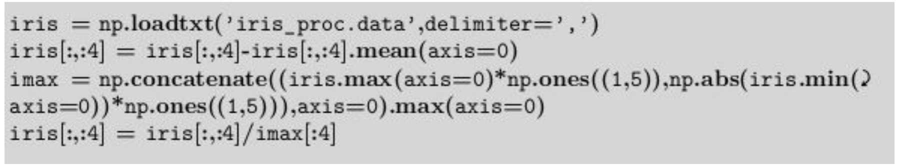
- Split the data into train:test:validation sets and start training

```
train  = x[0::2,:]
test  = x[1::4,:]
valid  = x[3::4,:]
traintarget  = t[0::2,:]
testtarget  = t[1::4,:]
validtarget  = t[3::4,:]
```

```
>>> import mlp
>>> net  = mlp.mlp(train,traintarget,3,outtype='linear')
>>> net.mlptrain(train,traintarget,0.25,101)
Iteration:  0  Error:  12.3704163654
Iteration:  100  Error:  8.2075961385
```

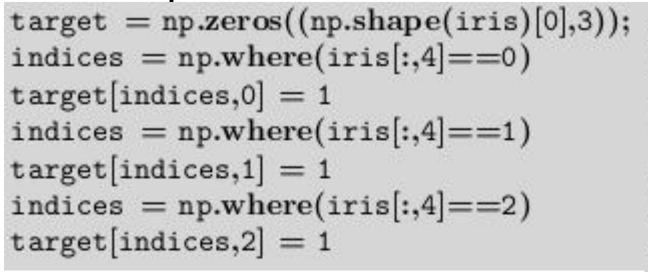
Classification Problems

- Our goal is to correctly predict the correct iris flower out of a possible 4 types using the Iris Dataset.
- Read dataset and preprocess data:

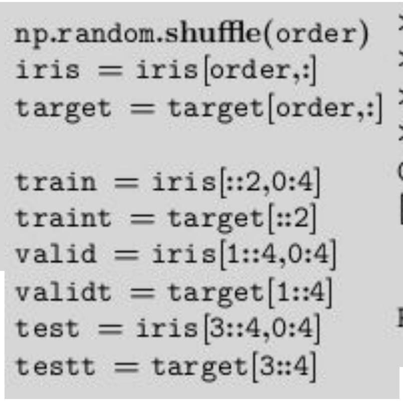


```
iris = np.loadtxt('iris_proc.data',delimiter=',')
iris[:,:4] = iris[:,:4]-iris[:,:4].mean(axis=0)
imax = np.concatenate((iris.max(axis=0)*np.ones((1,5)),np.abs(iris.min(
axis=0))*np.ones((1,5))),axis=0).max(axis=0)
iris[:,:4] = iris[:,:4]/imax[:4]
```

- Split data into train:test:validation sets then shuffle and finally train

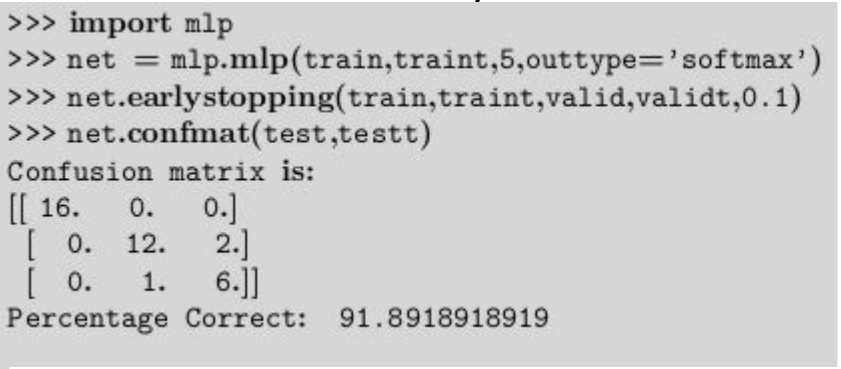


```
target = np.zeros((np.shape(iris)[0],3));
indices = np.where(iris[:,4]==0)
target[indices,0] = 1
indices = np.where(iris[:,4]==1)
target[indices,1] = 1
indices = np.where(iris[:,4]==2)
target[indices,2] = 1
```



```
np.random.shuffle(order)
iris = iris[order,:]
target = target[order,:]

train = iris[::2,0:4]
traint = target[::2]
valid = iris[1::4,0:4]
validt = target[1::4]
test = iris[3::4,0:4]
testt = target[3::4]
```



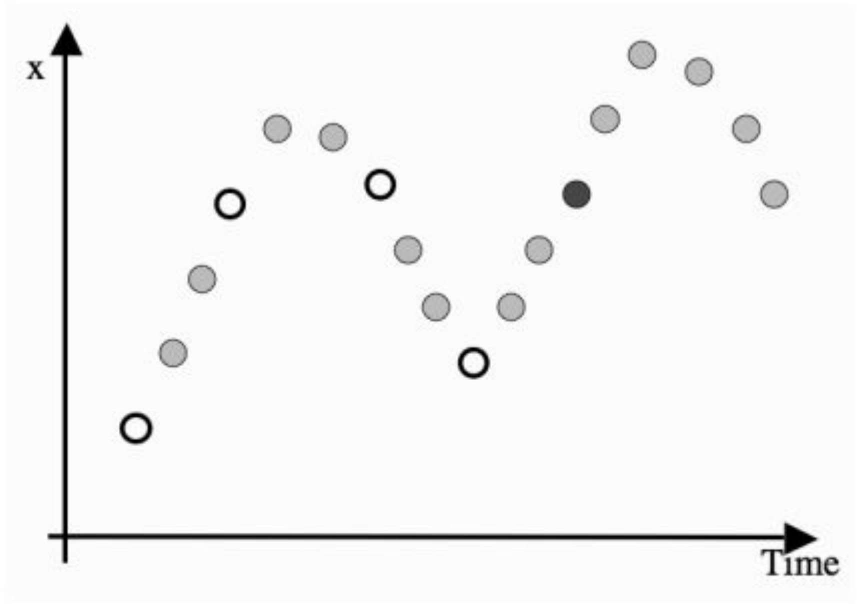
```
>>> import mlp
>>> net = mlp.mlp(train,traint,5,outtype='softmax')
>>> net.earlystopping(train,traint,valid,validt,0.1)
>>> net.confmat(test,testt)
Confusion matrix is:
[[ 16.   0.   0.]
 [  0.  12.   2.]
 [  0.   1.   6.]]
Percentage Correct:  91.8918918919
```

Time-Series Problems

- In time-series problems we have a set of data that shows how something varies over time.

- Using this data we want to predict how the data will vary in the future

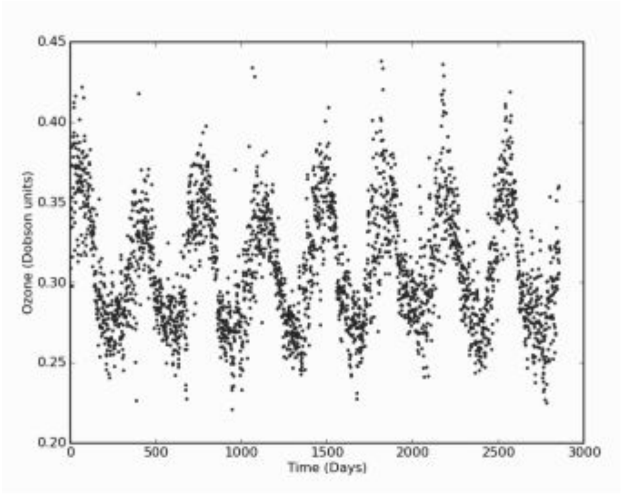
- Challenges:
  - Lots of variation between samples
  - How many data points should we look at to make a prediction (size of input)?
  - How far apart in time should we space the input (size of regain of interest)?

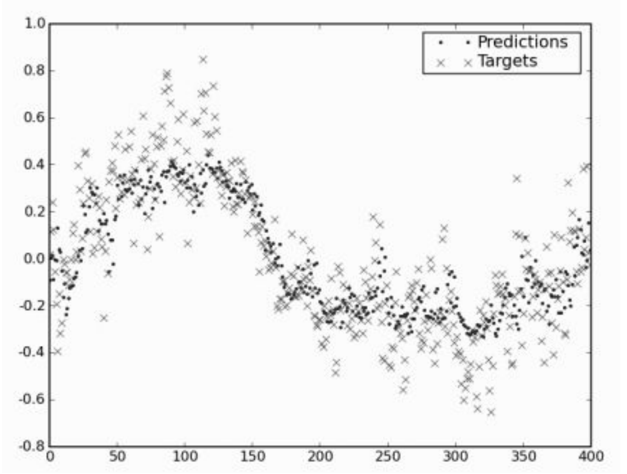


Time-Series Problems

- Our goal is to predict the ozone levels in the future and see if we can detect an overall drop in the mean ozone level
- The dataset consists of 4 elements for each sample: the year, the day of the year, the ozone level and the sulphur dioxide level. There are 2855 samples

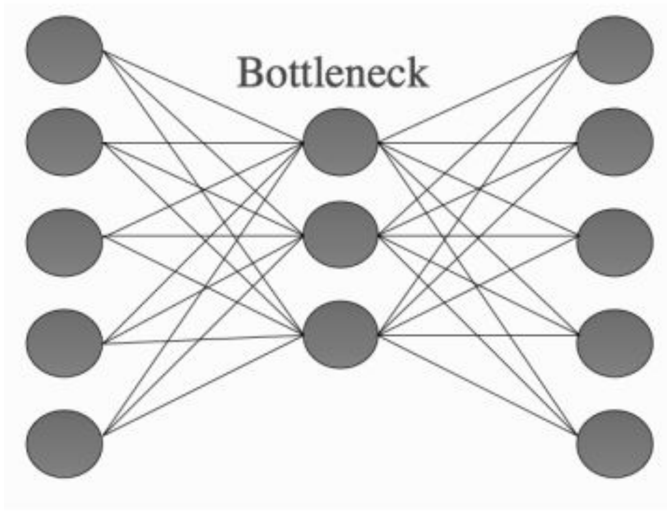
- To address the problems of how many inputs and how large sampling region should be, we write it as the following function:
  - y = x(t + r) = f(x(t), x(t - r), ..., x(t - kr))
  - r is space between samples and k is the number of datapoints. y is the prediction
- From here we treat time-series as a regression problem; the output nodes have a linear activation and we aim to minimise the sum-of-squares error





Auto-associative Problems

- The goal is to train a network so that it reduces the shape of the input then return it back to its original shape. Resulting in a network that can reproduce the input at the output layer.
- The idea is that we use a bottleneck hidden layer to represent all the information of the input in lower dimensional representation, thus we need to only keep the most important components of the data and ignore the noise (compression of the data).



Auto-associative Problems

- We can use Autoencoders to compress images into a 1D vector
- The network learns to reproduce the same image at the output, and the activations of the hidden nodes are recorded for each image.
- Use cases:
  - Denoising Images
  - Clustering
  - Generating new images
  - Dimensionality Reduction