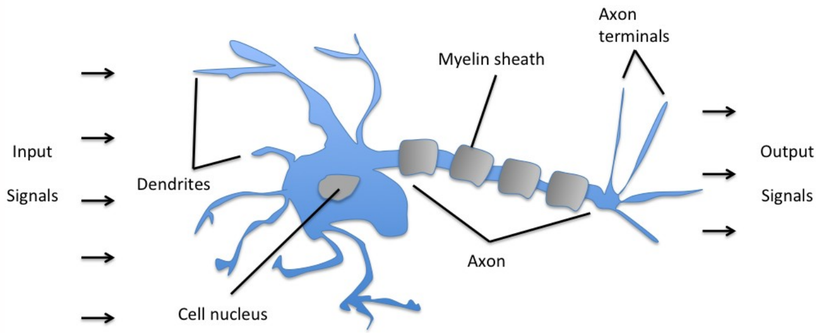
Ollscoil na Gaillimhe
University of Galway

EE4104
MLP Regression

University *of* Galway.ie

Mimicking Biological Neurons

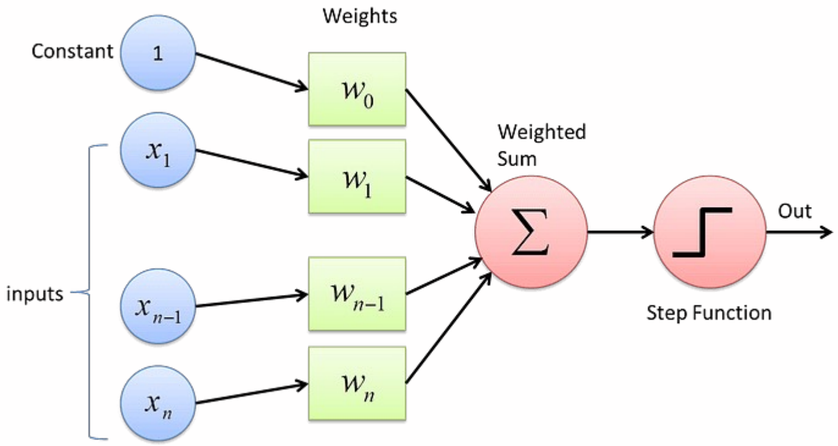
- The story of neural networks has its roots in work completed by McCulloch and Pitts in the 1940s and their *peceptron* model (for 2-class classification)
- This model mimicked some of this basic operations of a biological neuron
  - Electrochemical signals of various strengths arrive at the biological neuron (from other neurons) via dendrites
  - These signals are then accumulated in the body of the neuron
  - If the accumulated signals exceed some threshold then the cell emits an output electro-chemical signal via the axon towards other neurons



Schematic of a biological neuron.

Rosenblatt's Perceptron

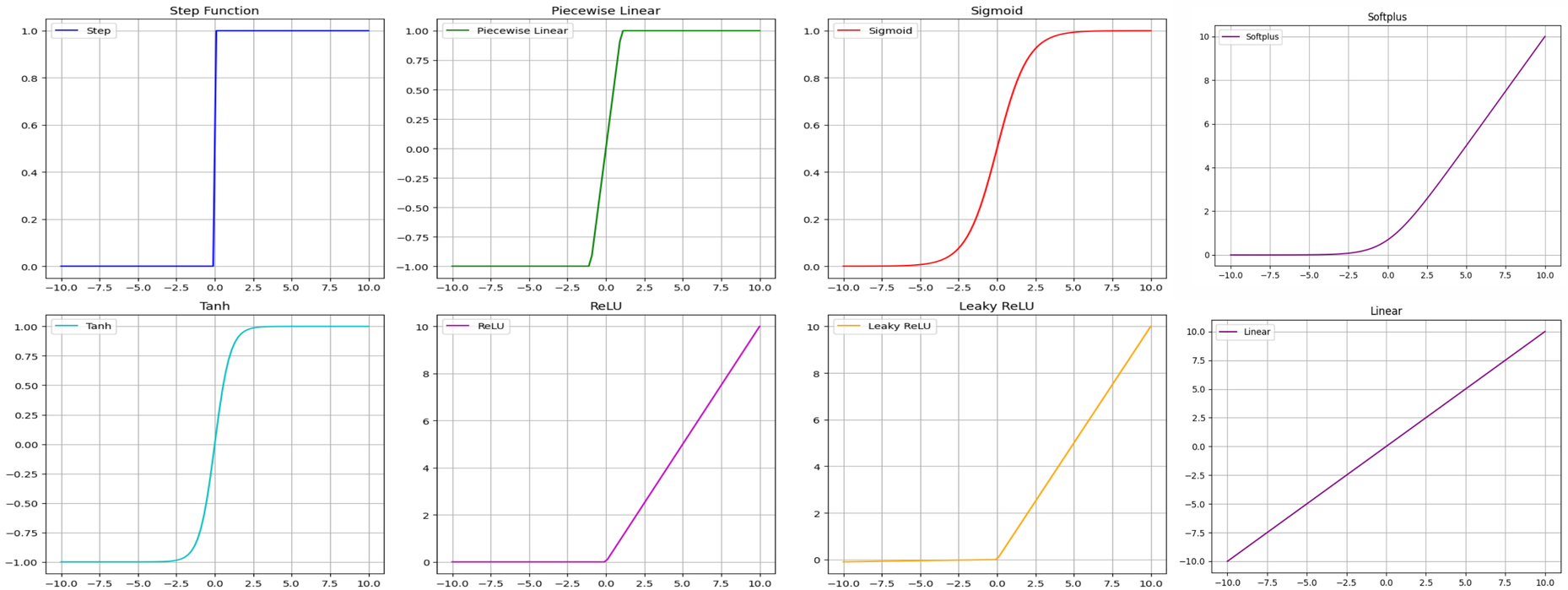
- In the late 1950s, Rosenblatt built on this idea to develop a concept of a perceptron which is closer to that which we use today
- The key contributions of his work were:
  - Introducing the concept of input weights which allow the contributions of the individual inputs to the neuron to be different
  - A supervised learning algorithm by which these weights could be adjusted so that they would "learn" by means of training examples



- Accumulated signal thresholding implemented as an additional constant input with the threshold controlled by the weight (w_0)
- Output controlled by an **Activation function** which implemented a (Heaveside) step function modelling the "All or None" behaviour of the biological cell output

Contemporary Perceptron Models

- Particularly over the last 40 years, the main change in the perceptron model has focussed on using a wide variety of different activation functions
  - Some more and some less biologically plausible
- Algorithms to implement “better” and more (computationally) efficient training of weights



Perceptron Operation - Mathematically

- Accumulation of weighted inputs
  - x_i are the input values to the perceptron
  - w_i is the weight associated with input x_i
- Comparison of the weighted accumulated input to a static threshold value:
  - b is a constant and termed the bias
- Generation of perceptron output by activation function:

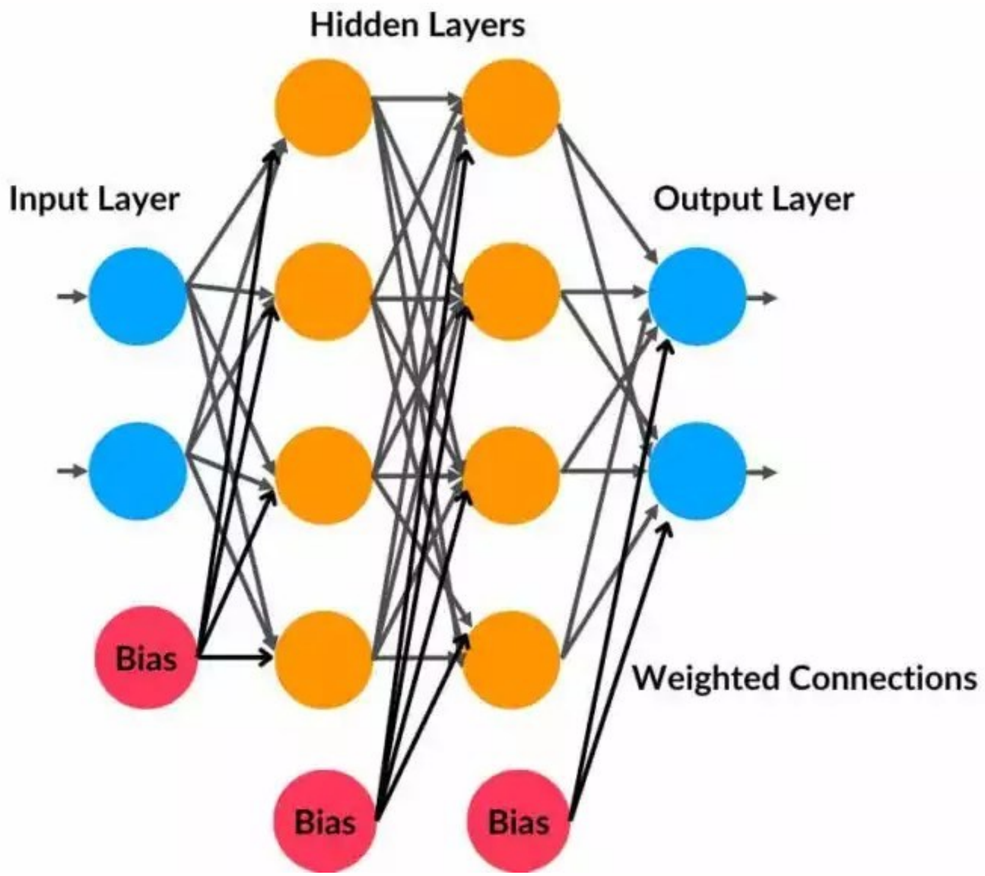
where $x_0=1$ (constant) and $w_0=-b$

Multi-Layer Perceptron (MLP) Neural Network

Ollscoil na Gaillimhe
University of Galway

- For regression (or classification) a single perceptron is not of much use as it does little more than what can be done with linear regression
  - In fact if a linear activation function is used then all it does is linear regression!!
- It is the use of these individually simple models in a dense layered architecture that facilitates their use in problems which implement highly non-linear behaviour
  - Such an architecture is known as a Multi-Layer Perceptron (MLP) Neural Network

MLP Network Structure



- Use of the term “layer” in “input layer” can be misleading as it only means the input values are applied there (i.e. no weighting, summation, bias, activation function)
- Notation

where :

$y_{j,k}$ is the output of the neuron j in layer k

$w_{i,k}$ is the i^{th} input to neuron j in layer k

$x_{i,j,k}$ is the i^{th} input to neuron j in layer k

$f_{j,k}(u)$ is the activation function implemented in neuron j of layer k

Back Propagation Training Algorithm

Ollscoil na Gaillimhe
University of Galway

- The real revolutionary algorithm in unlocking the potential for training MLP models was the supervised learning based back-propagation algorithm proposed in 1986 by Rumelhart, Hinton and Williams
- In the case of using MLPs for regression, a training dataset would consist of examples where each example would contain
  - A set of values applied to the input “layer” of the MLP
  - The desired output value for the single perceptron in the output layer (y)
- For each example in the training set we can complete the “feed forward” step where the input values of that example are applied to the MLP input and by calculating the output of each neuron in each hidden layer we will eventually calculate the output (for that input example) predicted by the MLP ()

Back Propagation Training Algorithm (2)

- The backpropagation step is then implemented which works backwards using the error between the desired (y) and predicted output () summed across all the examples in the training set and this error is used to “back-propagate” values through the network resulting in the adjustment of all the weights (and biases) in the network
  - This constitutes one ***epoch*** of training and training continues for many Rinehart epochs until it is deemed (by some criterion which we will later discussed) that the network has adequately ***learned*** the input/output relationship from the training examples

Steepest Descent in Back Propagation

Ollscoil na Gaillimhe
University of Galway

- We have already encountered the key mechanism that is used within the back propagation algorithm, namely steepest descent

- where is the loss function of the MLP which depends on all of the weights\bias values forming the network
- Specifically in the case of MLP regression the widely used loss function is the Sum (SSE) or Mean Squared Error(MSE) function calculated at the one output layer node

Output Layer Weight Update

Ollscoil na Gaillimhe
University of Galway

- Let's first focus on the output layer which will contain a single output neuron
- Rather than referring to for this layer, we will refer to just as
  - As there is only a single neuron (j=1) in an MLP regression output layer (and we are only considering the output layer (k=output layer)
- Remember from earlier that for a given training example (m)

where

- N is the number of neurons in the layer directly before the output layer
- $X_{i,m}$ is the output of the i^{th} neuron in the layer directly before the output layer for the m^{th} training example
- w_i is the weight connect the i^{th} neuron in the layer directly before the output layer to the output layer neuron

Output Layer Weight Update (2)

- We can re-write out loss function as:

- We need to determine an expression for to implement the steepest descent algorithm
- Used the chain rule to help with this:

Output Layer Weight Update (3)

- The first term in the chain rule is

then

where is the error between desired and predicted output for the m^{th} training example

- The second term is but since then we can just write
- The final term is but and so

Output Layer Weight Update (4)

- Putting in all three terms:

- For MLP regression typically a linear activation is used in the output layer neuron (as we will typically not want to have any restrictions on the range of values that the output value can take)
  - Hence and hence
- In this case (of a weight connecting to the output node of a MLP regression with a linear activation function in the output node), the weight update rule for the m^{th} example in the training set becomes

What about weights connecting other layers?

Ollscoil na Gaillimhe
University of Galway

- For weights leading into the output node (as there will only be one when using an MLP for regression), the weight update equation is

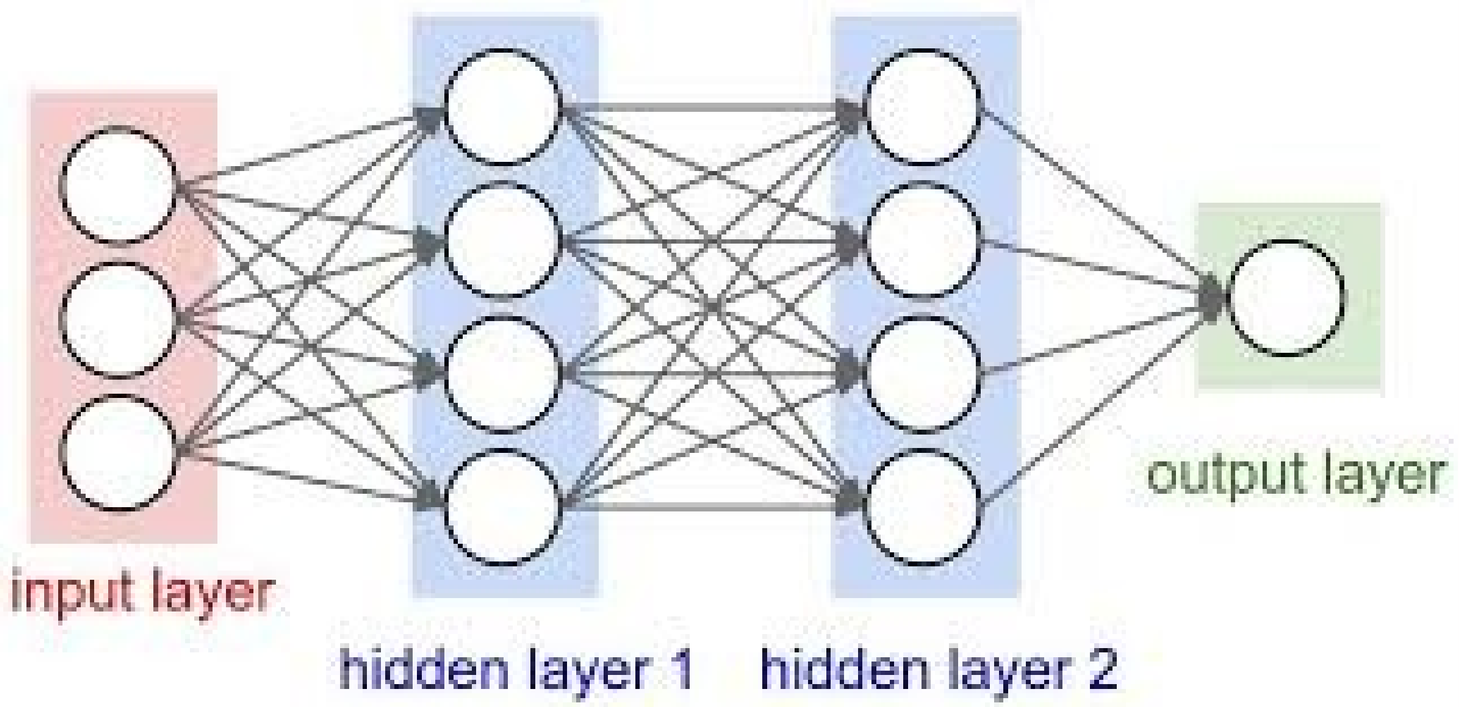
- We will define the term and hence the weight update equation becomes
  - Since the activation function for the output node of an MLP regression algorithm will typically be linear since

Numeric Example of (output) weight update

- We have TWO (!!) examples in our training dataset for an MLP regression model formed by a single perceptron:
  - x={-0.21,0.45,-0.65,0.12} y=1.2
  - x={0,0.21,0.32,0.52} y=-0.5
- Assuming the initial weights and bias were randomly set to w={-0.43,0.86,0.56,-0.32,0.87} (where the first value is w_0), determine the updated weights after completion of one training epoch
  - Use a step size value of 0.1 and do not average the weight update values

Weights feeding into Hidden Layer Neurons

- Let's consider a MLP used for regression with TWO hidden layers



- We have already considered the rule to update the weights feeding into the output layer neuron

Hidden Layer Weight Update

Ollscoil na Gaillimhe
University of Galway

- To reduce confusion let's drop the "m" subscript from our notation but bear in mind that anything we derive refers to what is happening when the m^{th} example from the training set is being processed
- For a weight feeding into a **neuron in hidden layer 2**, the weight update equation remains the same for

- The chain rule expansion of the derivative\gradient term remains the same

- However, the difference this time is that is the output of the activation function for the j^{th} neuron in the k^{th} (hidden) layer
  - At the moment we are considering layer k=2

Hidden Layer Weight Update (2)

- The replacements that we determined for the MLP output neuron are still applicable
  - , the first derivative of the activation function of the neuron in the k^{th} hidden layer
  - , the input to the weight feeding into the i^{th} neuron in the k^{th} (hidden) layer

Hidden Layer Weight Update (3)

Ollscoil na Gaillimhe
University of Galway

- So the challenging term that we are trying to resolve or for this specific case and we can expand that (using chain rule) as

Since

Hidden Layer Weight Update (4)

- So for the weight update equation for a weight feeding into the neuron in that hidden layer

$$= \; =$$

$$-$$

$$=$$

- As we did with the output (layer) neuron, we will define

$$=$$

Hidden Layer Weight Update (5)

Ollscoil na Gaillimhe
University of Galway

- Now let's consider another preceeding layer of neurons (layer 1)
- For neurons in this additional layer, we also need some way of evaluating the derivative of the loss function **with respective to the output () of the j^{th} neuron in this (first) hidden layer**
- We first need to recall the chain rule which is used in the case of multivariate preceding calculus
  - If f(x,y) is a function of x and y and both x and y are functions of time (t) then
    or, in a more general case, if and all x_i variables are functions of time then

Hidden Layer Weight Update (5)

Ollscoil na Gaillimhe
University of Galway

- The key point is that the outputs of neurons in second hidden layer ($y_{i,hid-2}$) are functions of the outputs of neurons in the first hidden layer
- So by the multivariate chain rule

Hidden Layer Weight Update (5)

- This results in a weight update equation of these neurons in the first hidden layer

$$=$$

- As we have done at all other layers, we can define a

Final general weight update equation

- This results in a weight update equations for all 3 layers (in our example) which can then be generalised:

Output (neuron) layer (Layer 3)	=
Hidden Layer 2	
Hidden Layer 1	
We can generalise to: **Hidden Layer n**	

What about the activation function?

Ollscoil na Gaillimhe
University of Galway

- The weight update equation (for all layers) still contains the term for the first derivative of the activation function (of the node that the weight feeds into)
- For practical implementations, we should only use activation functions where the value of this derivative for any value (of the result of the weight sum of the inputs to the neuron) can be easily (quickly) and accurately calculated
- Examples of activation functions where the calculation of the derivative at any value of u is trivial include:

  - Linear : We have already seen this one as means
  - Piecewise Linear: means

Other Common Activation Functions

Ollscoil na Gaillimhe
University of Galway

- ReLU : m
- Leaky ReLU : m with typically

- For all of the activation functions to date BOTH the function and its derivative are very simple to calculate!!!
  - This has a direct impact on how long training can take!!
- But what about the more complex activation functions that are quite commonly used?
  - Sigmoidal function
  - Hyperbolic Tan (tanh) function

Sigmoid and tanh activation functions

Ollscoil na Gaillimhe
University of Galway

- The sigmoid and tanh (hyperbolic tan) functions historically were very commonly used in neural networks (and MLP networks in particular)
  - Link to biological neuron “All or none” output firing mechanism
  - Less commonly used in some forms for contemporary neural network implementations (particularly “deep” networks) due to “vanishing gradient” problem
  - While activation functions like ReLU have become more common (as they are computationally simpler and do not suffer from the “vanishing gradient” problem) the use of sigmoid and tanh as activation functions is still common in general and essential for certain forms of networks

sigmoid:

tanh:

First derivative of sigmoid

- The first derivative of the sigmoid yields a result which can be very efficiently calculated bearing in mind that f(u) will be calculated for a given training example in the feed-forward step

- The (generalised) weight update equation now becomes:

First derivative of tanh

Ollscoil na Gaillimhe
University of Galway

- There is a similar result which allows the first derivative of tanh to be very efficiently calculated if you already have calculated

- The (generalised) weight update equation now becomes:

Batch gradient\steepest descent

Ollscoil na Gaillimhe
University of Galway

- As noted previously, the process of accumulating the weight updates across all examples in the training dataset BUT only updating the weights once every epoch based on this accumulated value is the truest form (relative to theory) of the steepest descent algorithm
- The general characteristics of **batch gradient\steepest descent**
  - Initial convergence can be slow as every weight is only updated once every epoch of training
  - On the positive side, this process of convergence while slow tends to be more "stable" which is important towards the end of the training process
  - Computationally\Memory intensive since the accumulated weight updates must be calculated and stored for ALL weights while processing ALL examples in the training dataset
  - For problems which have a convex loss function (i.e. where there is a single minimum) then this approach is guaranteed to converge towards that solution

Stochastic gradient\steepest descent

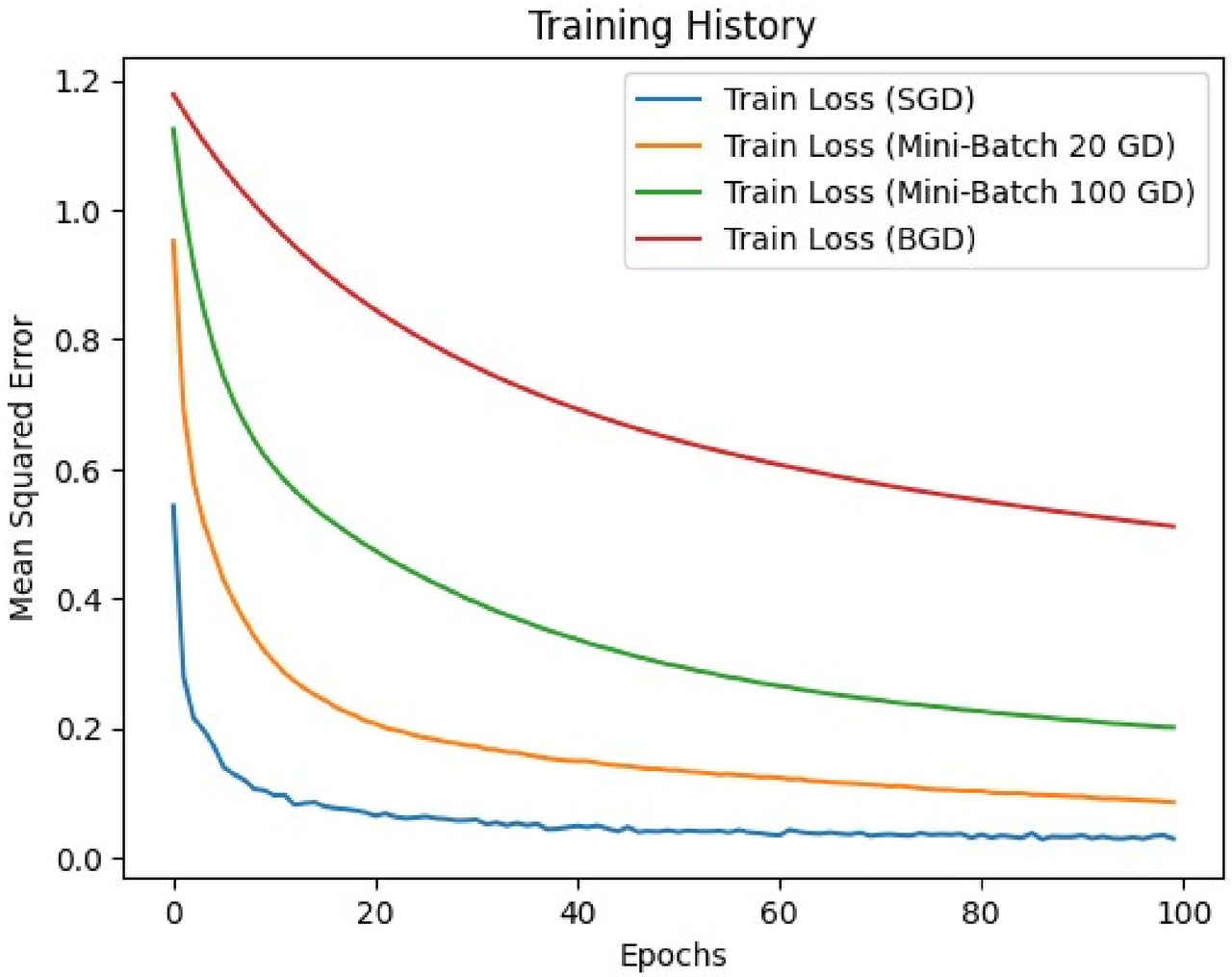
Ollscoil na Gaillimhe
University of Galway

- Whilst batch gradient descent is closest to theory, an alternative variant is **Stochastic Gradient Descent (SGD)** where weight updates are completed after every training example
- Some general comments on SGD:
  - Less memory requirements as no need to store an accumulated weight update for all the weights
  - Whilst not strictly true to theory, SGD behaves like BGD with a small amount of "randomness"/"noise" introduced to the weight updates procedure
  - Such "noise" may be useful in complex problem for the algorithm to "escape" local minima in the loss surface
  - As there are now one weight update for every training example, the convergence process can be expected to be (significantly) faster than BGD but it will be "noisier" (which can be problematic particular towards the end of training process)
  - SGD cannot be implemented in a parallelised form whereas BGD could be
  - SGD should be implemented with a randomised order of using training examples in each epoch

Mini-batch gradient\steepest descent

Ollscoil na Gaillimhe
University of Galway

- Mini-batch gradient descent is a compromised between BGD and SGD
- Training examples are processed in batches of K examples and updates done once after all examples in a batch are processed
- Characteristic concerning convergence speed and stability fall somewhere between SGD and BGD



Extensions of Gradient Descent

Ollscoil na Gaillimhe
University of Galway

- There are many extensions of the basic gradient descent algorithm in common use particular for training contemporary “deep” networks
- There are a number of extensions on the basic concept which are particularly noteworthy
  - Inclusion of a “momentum” term
  - Adaptation of “learning rate”

Momentum Term

Ollscoil na Gaillimhe
University of Galway

- The inclusion of an additional “momentum” term in the weight update rule dates back a considerable time
- The weight update rule used is of the form:

$\alpha << 1$ is a small constant controlling the contribution of the momentum term

- Often expressed in this form:

$0 < \beta < 1$ is the “decay” term

- Main justifications for the inclusion of a momentum term are faster convergence, reduction in occurrence oscillations and “escaping” local minima or saddle points in loss surface

Adaptive Learning Rate\Step Size

Ollscoil na Gaillimhe
University of Galway

- A number of widely used algorithms (e.g. RMSprop, AdaGrad, ADAM) include a process of learning rate adaptation
- The weight update rule used is of the form:

- In such algorithm, the step size parameter is updated using various algorithms (e.g. division by averaged recent gradients)
- Some of the advantages of using an adaptive learning rate include:
  - Faster convergence in many cases
  - Less need to select appropriate learning rate as the algorithm will adapt it
  - Convergence is less impacted by the fact that gradients for different weights may be at different scales
- In essence it means that every weight in the network is being updated during training using a learning rate optimised for that weight

Criteria for terminating training

Ollscoil na Gaillimhe
University of Galway

- There are many criteria with can be used to terminate the training process
  - Maximum number of training epochs completed (should be used in conjunction with other criteria)
  - Training loss (e.g. SEE error) falls below a threshold value for a number of consecutive epochs
    - Challenging to select a suitable threshold value
    - Oscillations which are common in the training loss during convergence (particularly with some implementations of gradient descent) may cause a problem
  - Change in training loss between epochs falls below a threshold
    - Same potential problems as above!
  - Monitor performance of the algorithm with a separate **validation dataset** and terminate training when it exhibits (sustained) signs of stagnating or increasing
    - Particularly suitable for avoid **overfitting** to the training dataset

Performance Measures

Ollscoil na Gaillimhe
University of Galway

- Once trained, the performance of the trained MLP regression implementation should be evaluated against a test dataset (unused in ANY way in the MLP design and training process)
- As noted previously there are quite a number of different measures that can be used
  - SSE
  - MSE
  - RMSE
  - MAFE
  - R^2
    - Proportion of variance in the predicted variable which is explained by the model

Model Hyperparameters

Ollscoil na Gaillimhe
University of Galway

- All supervised algorithms that we have examined to date (for classification or regression) have parameters which affect the design and performance of the system which the user needs to optimise
  - Nearest Neighbour : Value of k, Use of Distance based Weighting
  - Probabilistic classifier : Event Models and parameters thereof
- However, in the case of MLPs (and Neural Networks in general) the number of such “parameters” is on a much different scale
  - Number of hidden layers
  - Number of nodes in each layer
  - Activation functions in each layer
  - Training termination criteria
  - Rubric for weight initialisation
  - Training rule and parameters thereof
- These are the **hyperparameters** of the model and ideally they need to be optimally selected

Tuning Hyperparameters

Ollscoil na Gaillimhe
University of Galway

- The process of attempting to find an optimally set of hyperparameters is often one of the most challenging aspects of using a neural network
- There are just too many hyperparameters to consider attempting to optimise all of them together and hence they are often optimised individually
- As previously discussed when we looked at optimising the k value for a nearest neighbour classifier, the process needs to be unbiased allowing for a fair comparison of the performance of the trained network for different values of a particular hyperparameter
- The use of K-fold cross validation is a very suitable approach to use when attempting to complete a process of optimisation of such hyperparameters