DI501 Introduction to Data Informatics

Neural Networks: Review

Backpropagation Algorithm

Introduction

- Neural networks share much of the same mathematics as logistic regression.
- But neural networks are a more powerful classifier than logistic regression, and indeed a minimal neural network (technically one with a single ‘hidden layer’) can be shown to learn any function.

Introduction

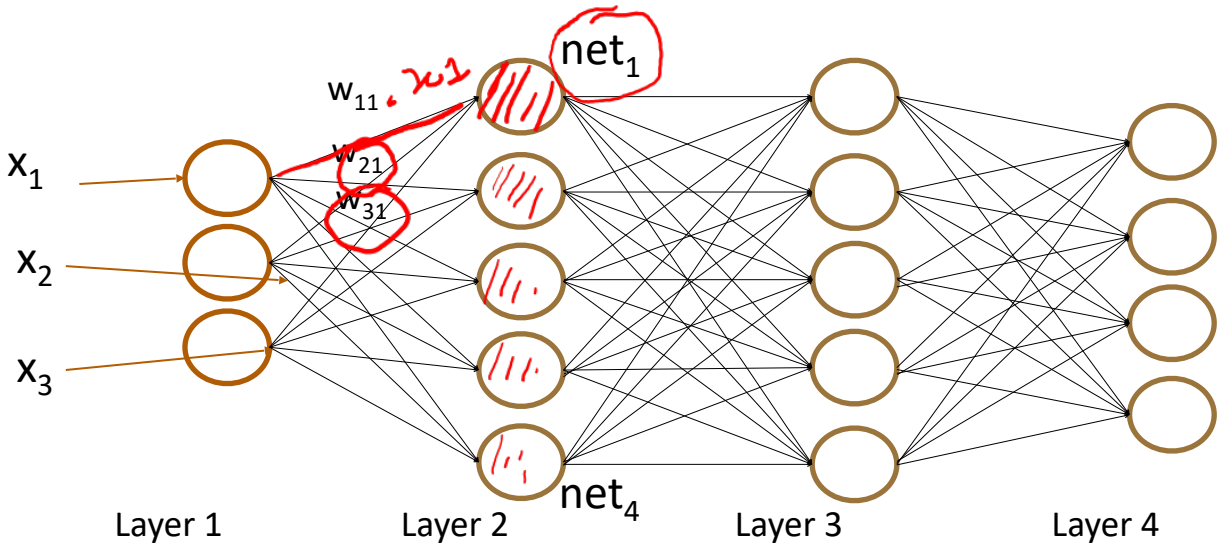
- In the 1970s Werbos developed a technique for adapting the weights.
- It was Rumelhart et al. (1986) who brought back neural networks to life.
- The weight adaption rule is known as backpropagation.
- It defines two sweeps of the network:
  - A forward sweep from the input layer to the output layer
  - A backward sweep from the ouput layer to the input layer
    - Error values are propagated back through the network to determine how the weights are to be changed during training.

Algorithm

Forward sweep

The standard summation of products is used to find the net total input.

$$net_j = \sum_{i=0} x_i w_{ij}$$

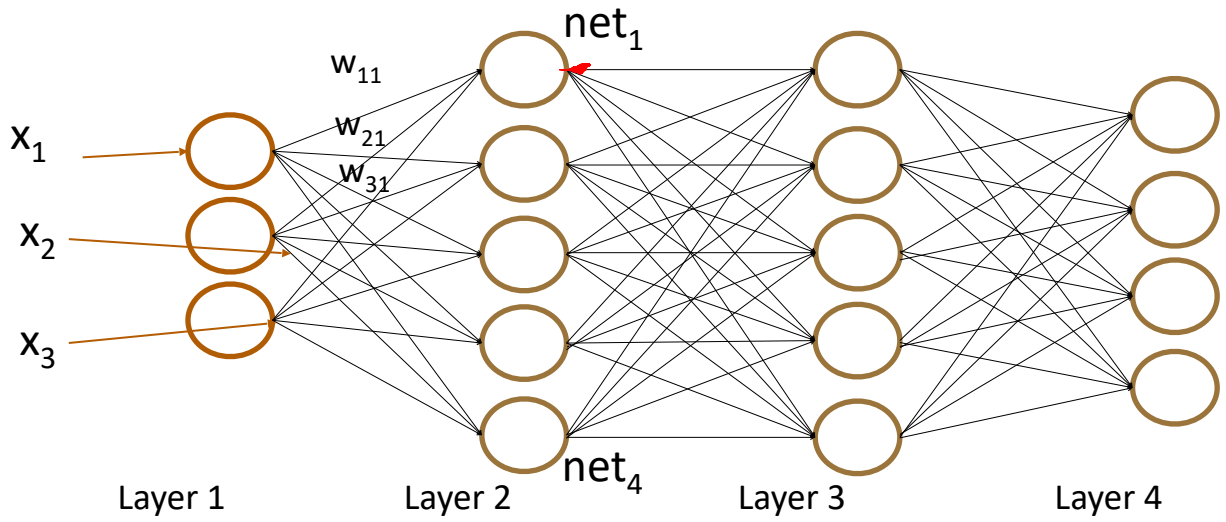


Algorithm

Forward sweep

For activation function *f* (such as logistic function) the output is:

$$o_j = f(net_j) = \frac{1}{1 + \exp(-net_j)}$$



Algorithm

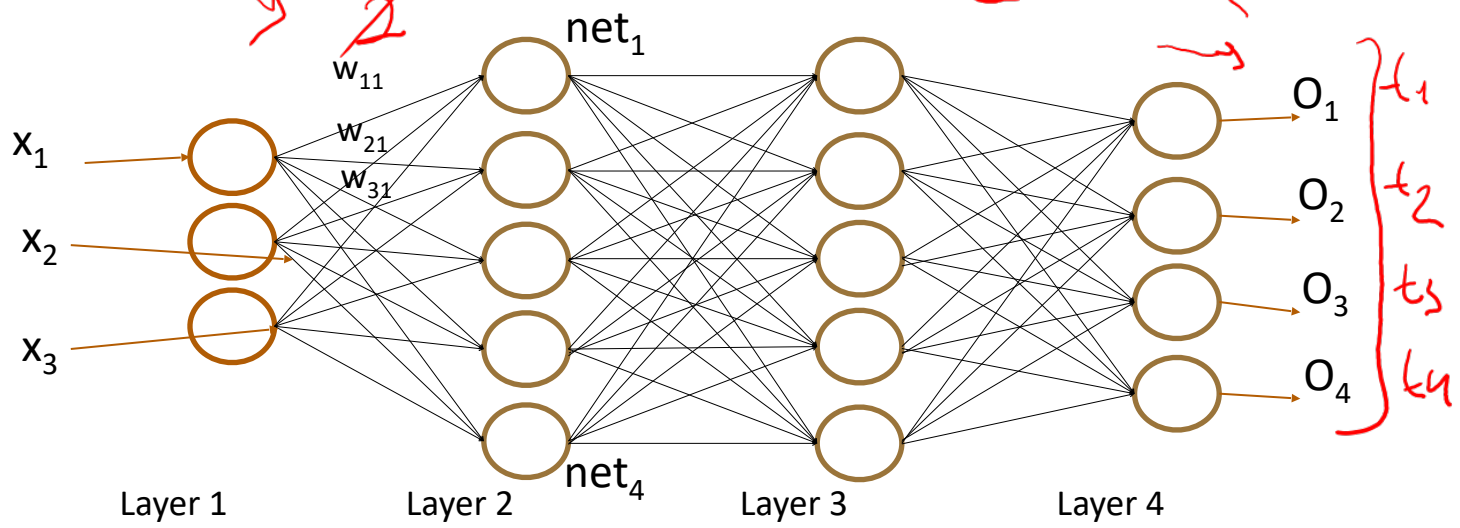
Backward sweep

The error is (t is the target value):

$$E = \frac{1}{2}\sum_j (t_j - o_j)^2$$

So we have:

$$\frac{dE}{do_j} = -(t_j - o_j)$$



Algorithm

Backward sweep

$\delta_j^{(l)}$ = "error" of node j in layer l.

We have:

$$\frac{dE}{do_j} = -(t_j - o_j)$$

Recall that:

$$o_j = f(net_j)$$

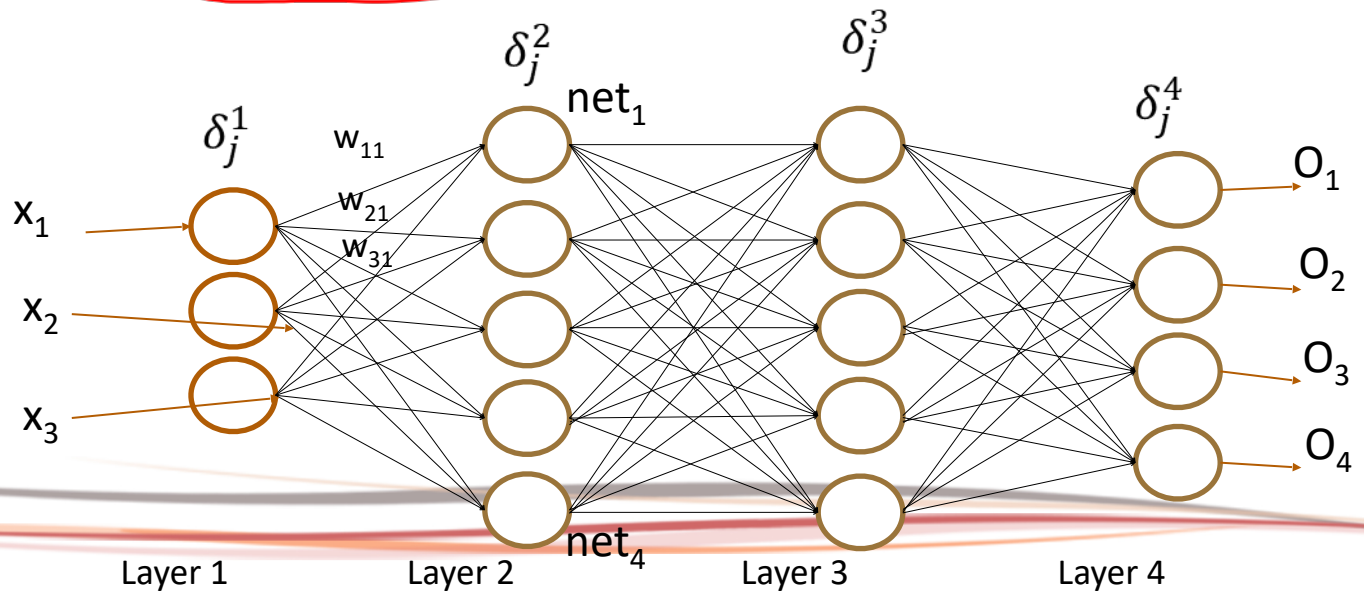
So we have:

$$\frac{do_j}{dnet_j} = f'(net_j)$$

$$\delta_j = \frac{dE}{dnet_j}$$

$$\delta_j = -\frac{dE}{do_j}\frac{do_j}{dnet_j}$$

$$\delta_j = (t_j - o_j) f'(net_j)$$



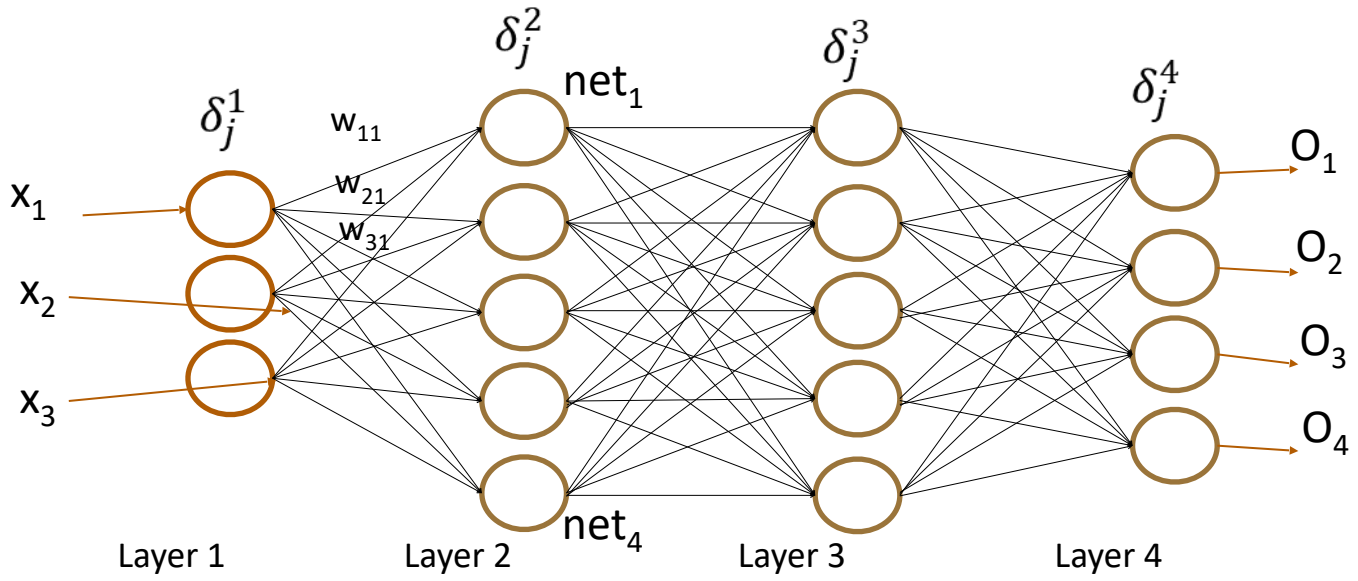
Algorithm

Backward sweep

$$\delta_j = (t_j - o_j) f'(net_j)$$

$$f(net_j) = \frac{1}{1 + \exp(-net_j)}$$

$$f'(net_j) = f(net_j)[1\text{-}f(net_j)]$$

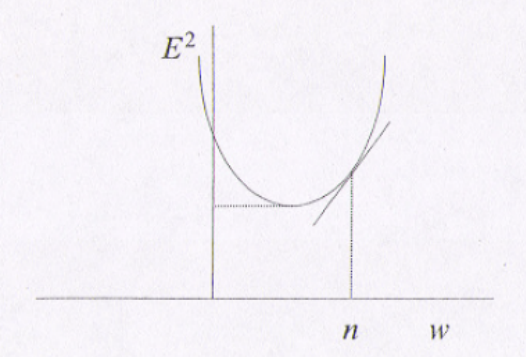


Algorithm

Backward sweep: propagation through weights

$$\frac{dE}{dw_{ij}} = \frac{dE}{do_j} \frac{do_j}{dnet_j} \frac{dnet_j}{dw_{ij}}$$

Figure at the left: the line represents the derivative of the error with respect to a weight at time n.



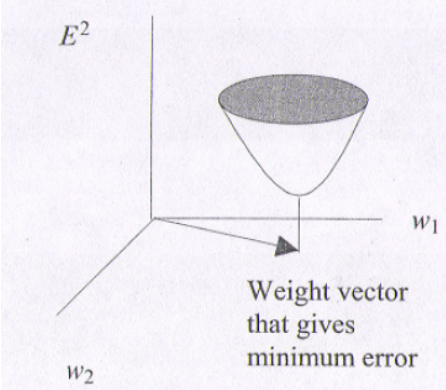


Figure at the right: the error can be plotted for different combinations of weights.

Algorithm

Backward sweep: propagation through weights

$$\frac{dE}{dw_{ij}} = \frac{dE}{do_j}\frac{do_j}{dnet_j}\frac{dnet_j}{dw_{ij}}$$

$$\frac{dE}{do_j} = -(t_j - o_j) \qquad \frac{do_j}{dnet_j} = f'(net_j)$$

$$net_j = \sum_{i=0} x_i w_{ij} \qquad \frac{dnet_j}{dw_{ij}} = x_i$$

$$\frac{dE}{dw_{ij}} = -(t_j - o_j) f'(net_j) x_i$$

$$\frac{dE}{dw_{ij}} = -(t_j - o_j) f(net_j)[1 - f(net_j)] x_i$$

The error in the output unit.

Algorithm

Backward sweep: propagation through weights

- Noting that the weight change should be in the direction opposite to the derivative of the error surface and applying a learning rate η, the weight change for a unit is:

$$\Delta w_{ij} = \eta \delta_j x_i \quad \text{where} \quad \delta_j = (t_j - o_j) f'(net_j)$$

  - For a hidden unit the error is:

$$\delta_j = f'(net_j) \sum_k \delta_k w_{kj}$$

Algorithm

Backward sweep: propagation through weights

- To reduce the likelihood of the weight changes oscillating, a momentum term, α, is introduced that adds in a proportion of the previous weight change:

$$\Delta w_{ij}(n+1) = \eta(\delta_j o_i) + \alpha \Delta w_{ij}(n)$$

Step 1. Read the first input pattern and associated output pattern. CONVERGE = TRUE

Step 2. For input layer – assign as net input to each unit its corresponding element in the input vector. The output for each unit is its net input.

Read next input pattern and associated output pattern.

Step 3. For the first hidden layer units – calculate the net input and output.

$$net_j = \sum_{i=0} x_i w_{ij}$$

With sigmoid transfer function:

$$o_j = f(net_j) = \frac{1}{1 + \exp(-net_j)}$$

Repeat **Step 3** for all subsequent hidden layers.

Step 4. For the output layer units- calculate the net input and output:

$$net_j = \sum_{i=0} x_i w_{ij} \qquad o_j = f(net_j)$$

Step 5. Is the difference between target and output pattern within tolerance? IF No THEN CONVERGE = FALSE

Step 6. For each output unit calculate its error:

$$\delta_j = (t_j - o_j) o_j [1 - o_j]$$

Step 7. For the last hidden layer calculate error for each unit:

$$\delta_j = o_j (1 - o_j) \sum_k \delta_j w_{kj}$$

Repeat **Step 7** for all subsequent hidden layers.

CONVERGE= =TRUE

no

yes

STOP

Step 8. For all layers update weights for each unit:

$$\Delta w_{ij}(n+1) = \eta(\delta_j o_i) + \alpha \Delta w_{ij}(n)$$

Example

EXAMPLE 2.3

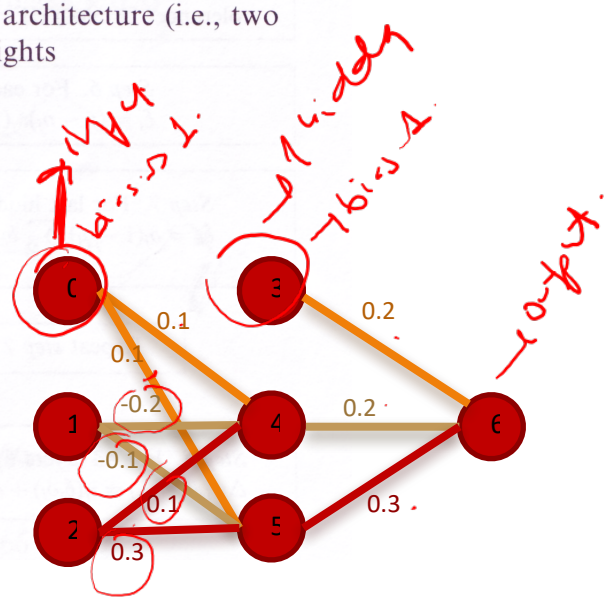
Perform a complete forward and backward sweep of a feedforward network using the backpropagation algorithm for an input pattern of [0.1 0.9] with a target output of 0.9 and a 2-2-1 architecture (i.e., two input, two hidden and one output unit) with weights

$$\begin{bmatrix} 0.1 & 0.1 \\ -0.2 & -0.1 \\ 0.1 & 0.3 \end{bmatrix}$$

for the first layer and

$$\begin{bmatrix} 0.2 \\ 0.2 \\ 0.3 \end{bmatrix}$$

for second layer.

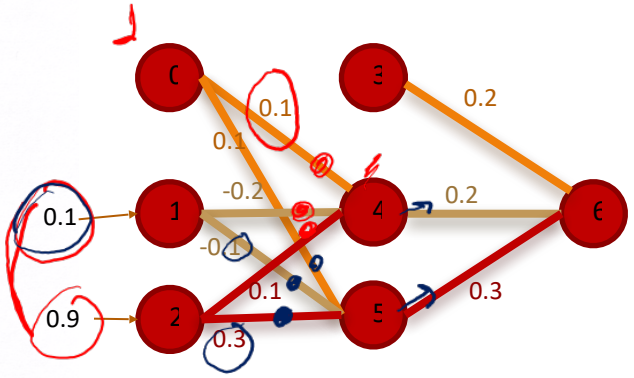


Example

SOLUTION 2.3

The output for the input units is simply the input pattern. The first row of either weight matrix defines the bias terms for each layer which, remember, are attached to a unit with an activation of 1. The units are numbered as {0, 1, 2} for the input layer, {3, 4, 5} for the hidden layer and {6} for the output layer, with units 0 and 3 being the bias units for the input and hidden layer, respectively.

$$net_4 = (1.0 \times 0.1) + (0.1 \times -0.2) + (0.9 \times 0.1)$$
$$= 0.170$$
$$o_4 = \frac{1}{1 + \exp(-0.17)}$$
$$= 0.542$$
$$net_5 = (1.0 \times 0.1) + (0.1 \times -0.1) + (0.9 \times 0.3)$$
$$= 0.360$$
$$o_5 = \frac{1}{1 + \exp(-0.36)}$$
$$= 0.589$$



Example

Similarly, from the hidden to output layer:

$$net_6 = (1.0 \times 0.2) + (0.542 \times 0.2) + (0.589 \times 0.3)$$
$$= 0.485$$
$$o_6 = 0.619$$

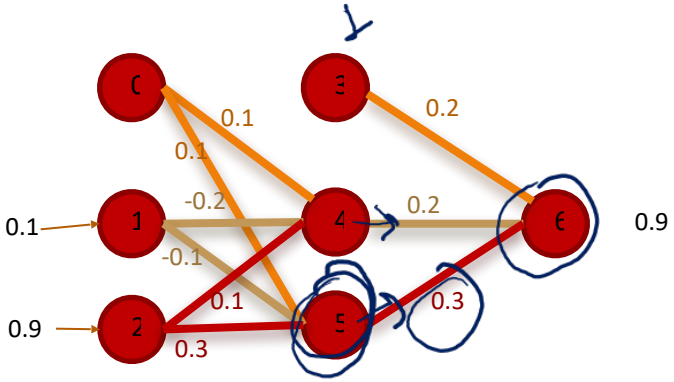
The error for the output node is: $\delta_j = (t_j - o_j) o_j [1 - o_j]$

$$\delta_6 = (0.9 - 0.619) \times 0.619 \times (1 - 0.619)$$
$$= 0.066$$

The error for the hidden nodes is: $\delta_j = o_j (1 - o_j) \sum_k \delta_j w_{kj}$

$$\delta_5 = 0.589 \times (1.0 - 0.589) \times (0.066 \times 0.3)$$
$$= 0.005$$
$$\delta_4 = 0.542 \times (1.0 - 0.542) \times (0.066 \times 0.2)$$
$$= 0.003$$

Note that the hidden unit errors are used to update the first layer of weights. There is no error calculated for the bias unit as no weights from the first layer connect to the hidden bias.



Example

The rate of learning for this example is taken as 0.25. There is no need to give a momentum term for this first pattern since there are no previous weight changes.

$$\Delta w_{5,6} = 0.25 \times 0.066 \times 0.589 = 0.01$$

$$\Delta w_{ij}(n+1) = \eta(\delta_j o_i) + \alpha \Delta w_{ij}(n)$$

$$\delta_6 = (0.9 - 0.619) \times 0.619 \times (1 - 0.619) = 0.066$$

$o_5 = 0.589$

The new weight is:

$$0.3 + 0.01 = 0.31$$

$$\Delta w_{4,6} = 0.25 \times 0.066 \times 0.542 = 0.009$$

Then:

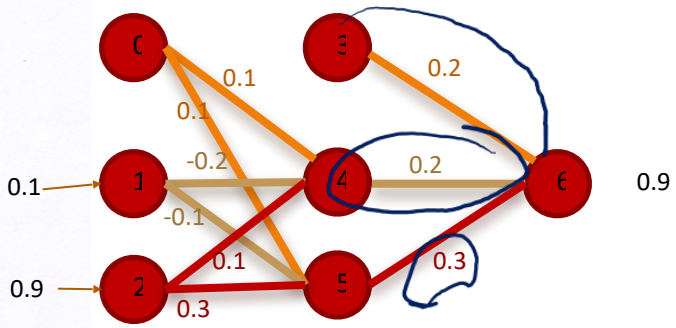
$$0.2 + 0.009 = 0.209$$

$$\Delta w_{3,6} = 0.25 \times 0.066 \times 1.0 = 0.017$$

Finally:

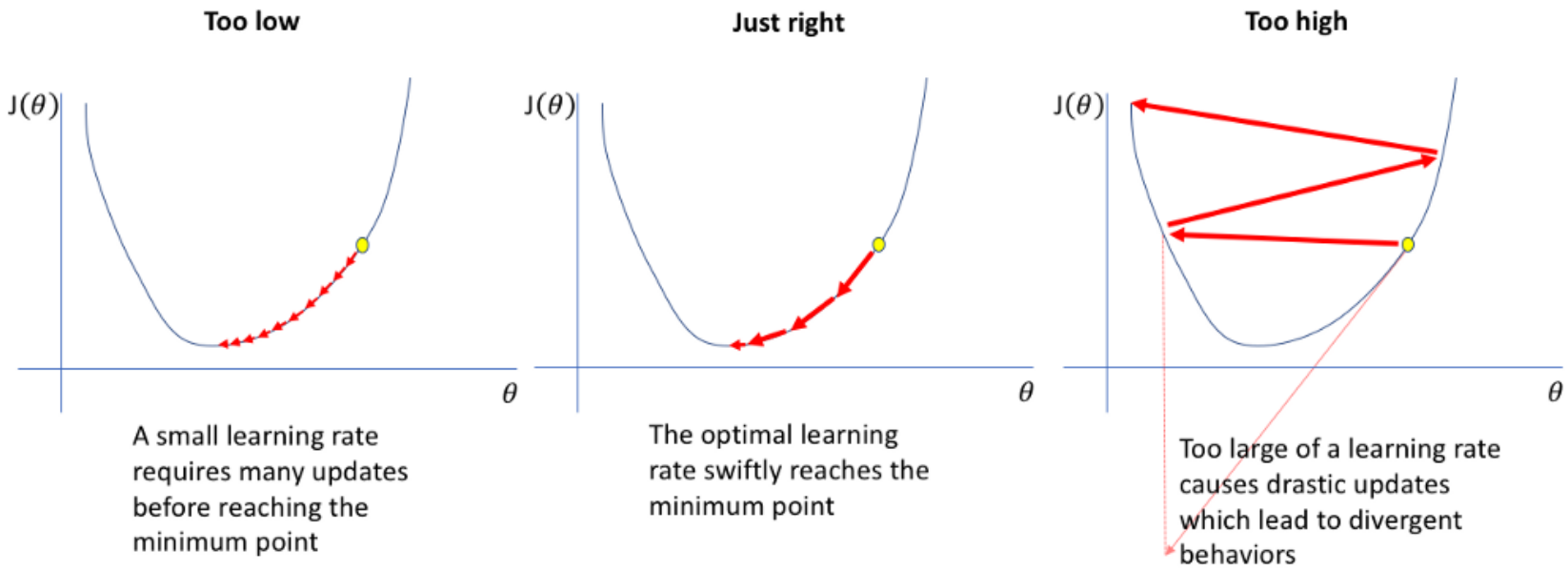
$$0.2 + 0.017 = 0.217$$

The calculation of the new weights for the first layer is left as an exercise.



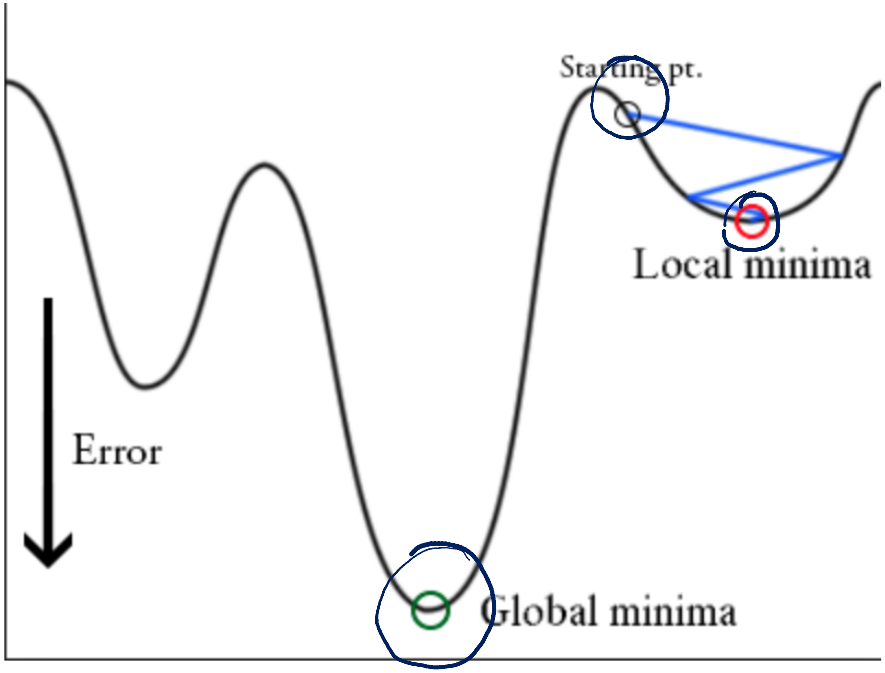
Parameter Tuning

Learning Rate



Parameter Tuning

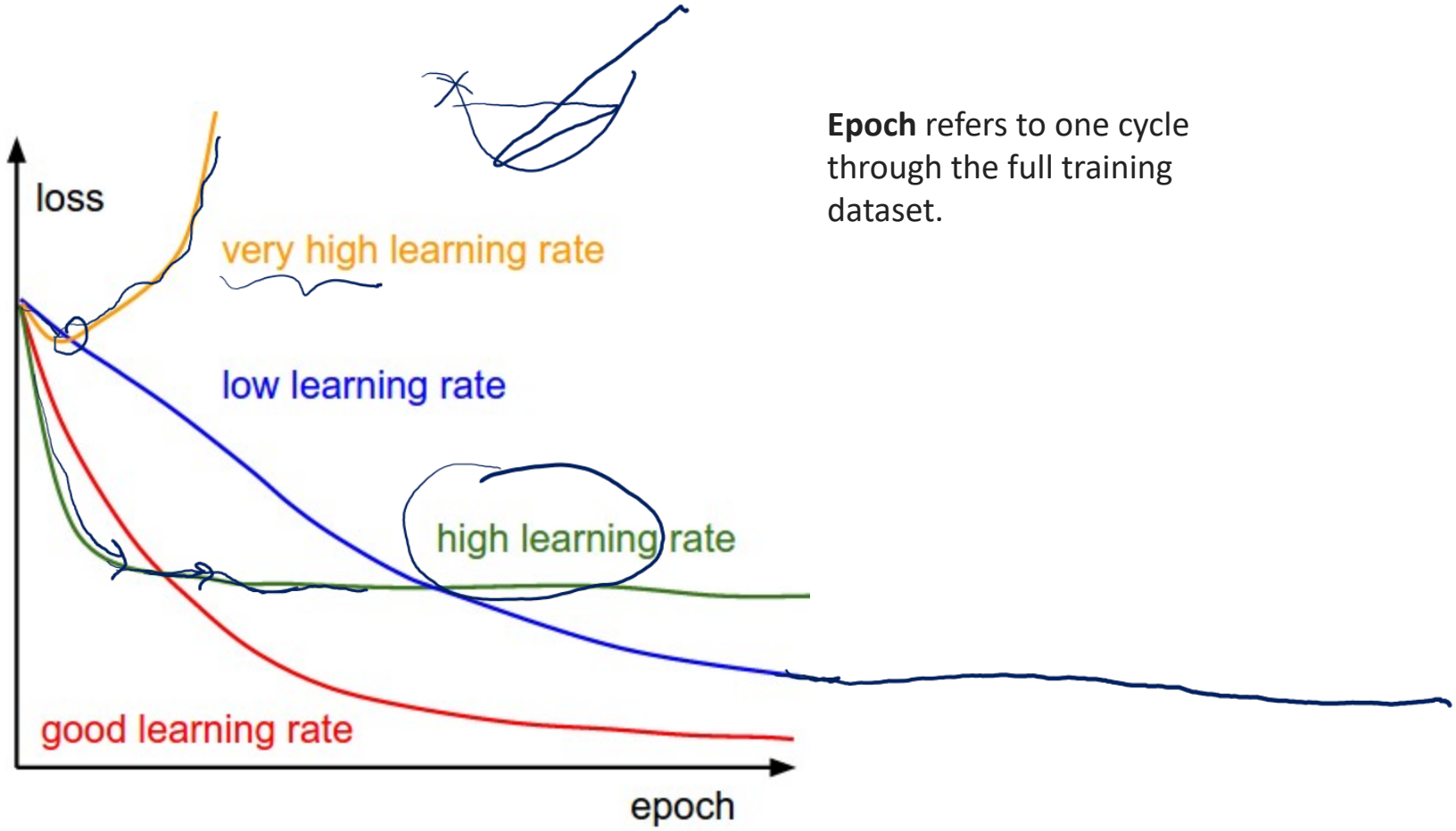
Learning Rate



Local minima vs. global minima

Parameter Tuning

Epochs



Epoch refers to one cycle through the full training dataset.

Parameter Tuning

Learning rate

- When training neural networks, it is common to use "weight decay" where after each update, the weights are multiplied by a factor slightly less than 1.
  - This prevents the weights from growing too large, and can be seen as gradient descent on a quadratic regularization term.
  - Weight decay is an example of a regularization method.

Parameter Tuning

Termination Criteria

- By epochs count: max number of iterations along all dataset
- By value of gradient: when gradient is equal to zero but small gradient->very slow learning
- When cost didn't change during several epochs
  - If error is not changing significantly, stop training
- Early stopping:
  - Stop when validation score starts increasing even when training score continue decreasing

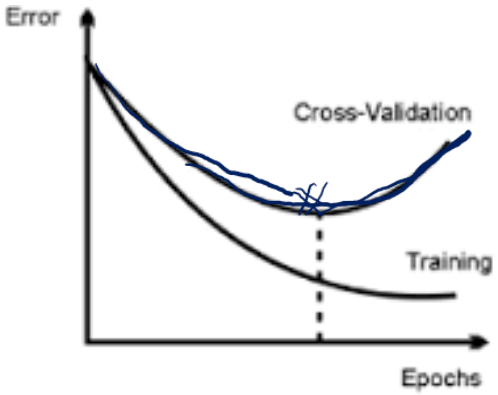
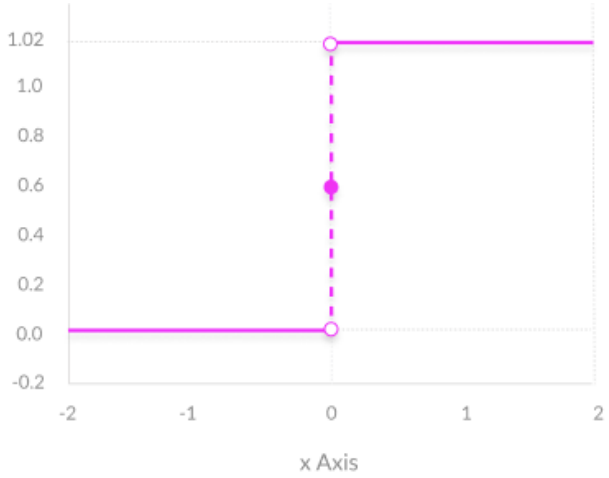


Figure 2: Profiles for training and cross-validation errors.

Parameter Tuning

Activation Functions

- Binary Step Function: A binary step function is a threshold-based activation function. If the input value is above or below a certain threshold, the neuron is activated and sends exactly the same signal to the next layer.

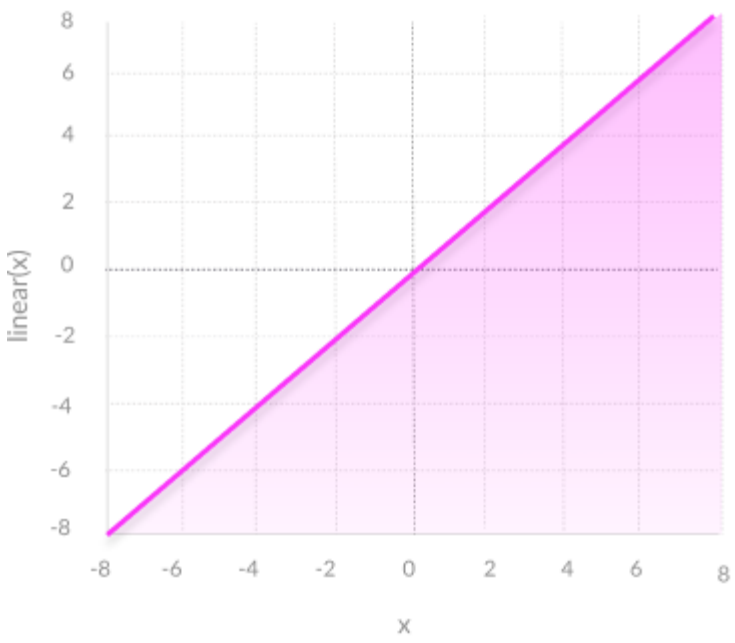


The problem with a step function is that it does not allow multi-value outputs—for example, it cannot support classifying the inputs into one of several categories.

Parameter Tuning

Activation Functions

- Linear Activation Function: A neural network with a linear activation function is simply a linear regression model. It has limited power and ability to handle complexity varying parameters of input data. It is better than a step function because it allows multiple outputs, not just yes and no.



It has two major problems:

1. Not possible to use backpropagation (gradient descent) to train the model—the derivative of the function is a constant, and has no relation to the input, *X*. So it's not possible to go back and understand which weights in the input neurons can provide a better prediction.

2. All layers of the neural network collapse into one —with linear activation functions, no matter how many layers in the neural network, the last layer will be a linear function of the first layer.

Parameter Tuning

Activation Functions

- Nonlinear Activation Functions: They allow the model to create complex mappings between the network's inputs and outputs.

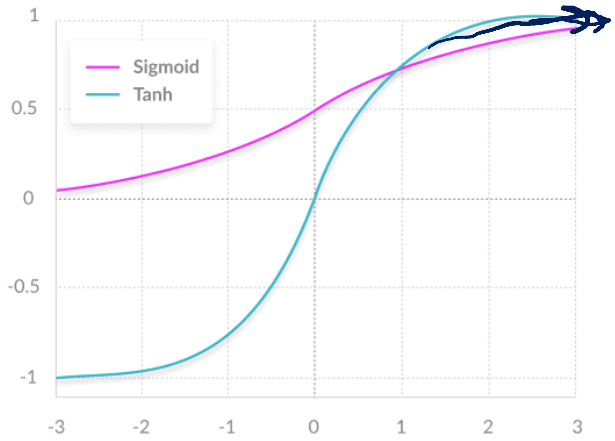
- Non-linear functions address the problems of a linear activation function:
  - They allow backpropagation because they have a derivative function which is related to the inputs.
  - They allow "stacking" of multiple layers of neurons to create a deep neural network.

Parameter Tuning

Activation Functions

- Advantages of sigmoid function: nonlinear, differentiable, has real-valued outputs, and approximates the sgn function.

- Disadvantages:
  - Vanishing gradient—for very high or very low values of X, there is almost no change to the prediction, causing a vanishing gradient problem. This can result in the network refusing to learn further, or being too slow to reach an accurate prediction.
  - Outputs not zero centered.
  - Computationally expensive



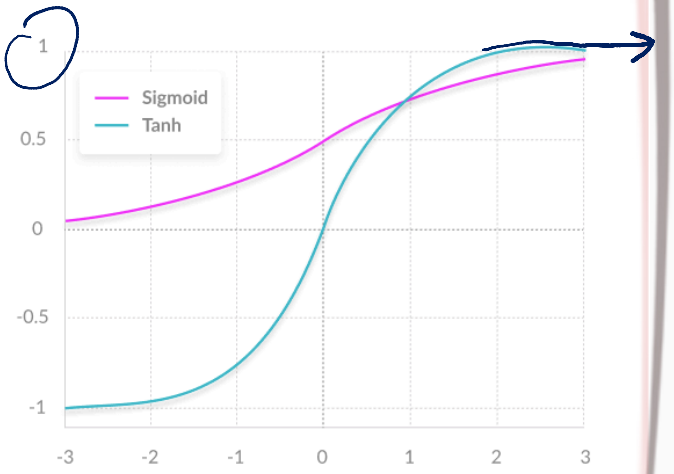
Two common neural network activation functions - Sigmoid and Tanh

Parameter Tuning

Activation Functions

- Advantages of tanh function:
  - Zero centered—making it easier to model inputs that have strongly negative, neutral, and strongly positive values.
  - Otherwise like the Sigmoid function.

- Disadvantages:
  - Like the Sigmoid function



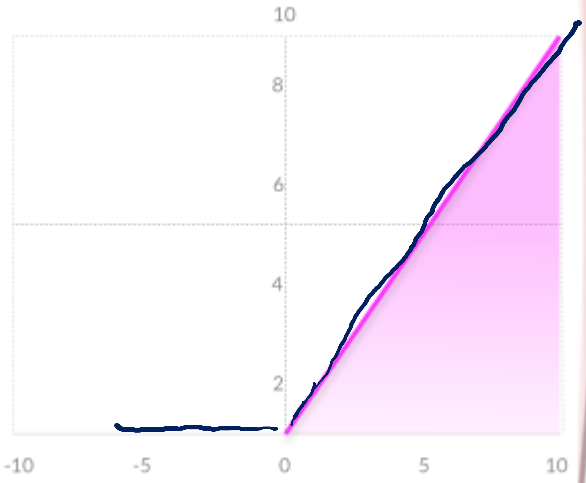
Two common neural network activation functions - Sigmoid and Tanh

Parameter Tuning

Activation Functions

- Advantages of ReLU (Rectified Linear Unit) function:
  - Computationally efficient—allows the network to converge very quickly
  - Non-linear—although it looks like a linear function, ReLU has a derivative function and allows for backpropagation

- Disadvantages:
  - The Dying ReLU problem—when inputs approach zero, or are negative, the gradient of the function becomes zero, the network cannot perform backpropagation and cannot learn.

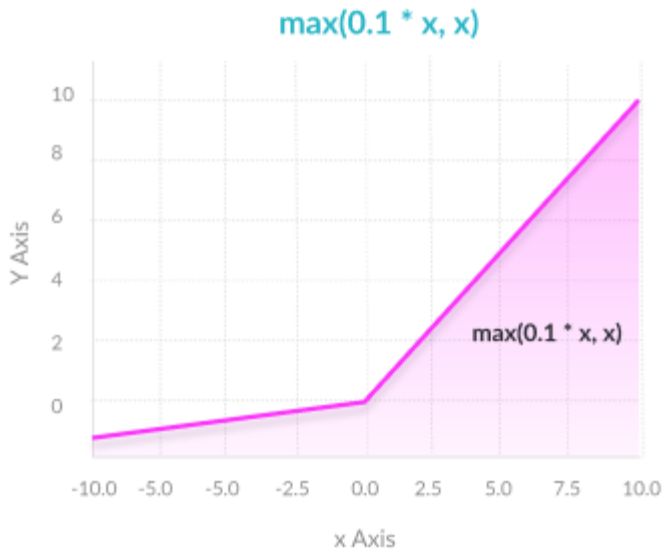


Parameter Tuning

Activation Functions

- Advantages of Leaky ReLU function:
  - Prevents dying ReLU problem—this variation of ReLU has a small positive slope in the negative area, so it does enable backpropagation, even for negative input values
  - Otherwise like ReLU

- Disadvantages:
  - Results not consistent—leaky ReLU does not provide consistent predictions for negative input values.



Parameter Tuning

Activation Functions

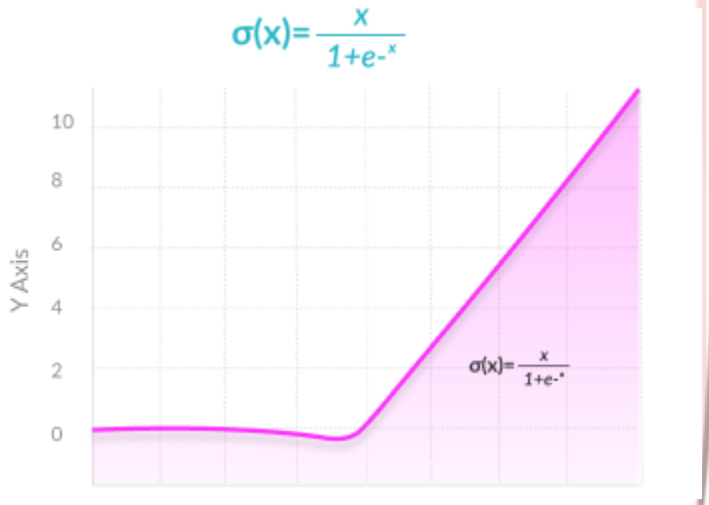
- Advantages of Softmax function:
  - **Able to handle multiple classes** only one class in other activation functions—normalizes the outputs for each class between 0 and 1, and divides by their sum, giving the probability of the input value being in a specific class.
  - **Useful for output neurons**—typically Softmax is used only for the output layer, for neural networks that need to classify inputs into multiple categories.

$$Softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)}$$

Parameter Tuning

Activation Functions

- Advantages of Swish function:
  - Swish is a new, self-gated activation function discovered by researchers at Google.
  - According to their paper, it performs better than ReLU with a similar level of computational efficiency. In experiments on ImageNet with identical models running ReLU and Swish, the new function achieved top -1 classification accuracy 0.6-0.9% higher.



Ramachandran, P., Zoph, B., & Le, Q. V. (2017). Swish: a self-gated activation function. *arXiv preprint arXiv:1710.05941*, *7*.

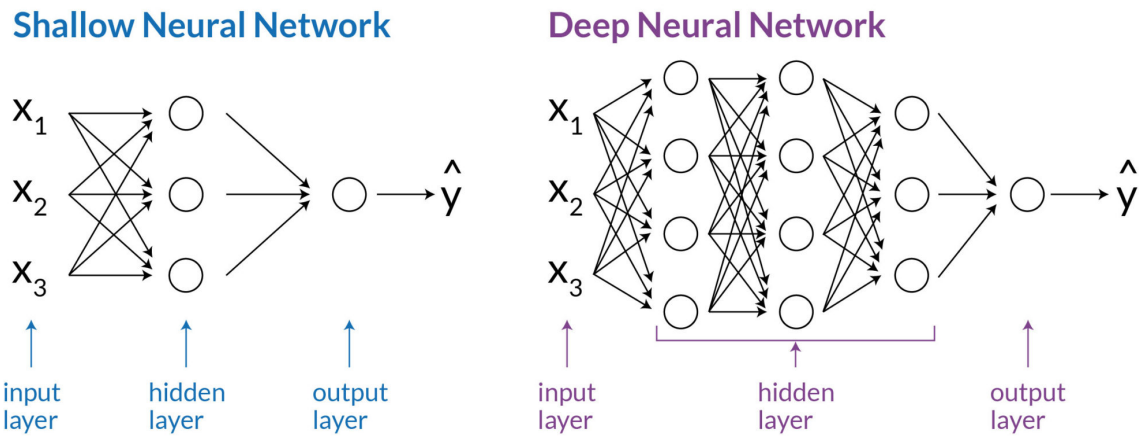
Parameter Tuning

Network Topology

- A neural network can be
  - “shallow”:1, 2 hidden layers
  - “deep”: 3..10 hidden layers
  - “very deep”: More than 10 layers

- Research from Goodfellow, Bengio and Courville and other experts suggests that neural networks increase in accuracy with the number of hidden layers.

- Hidden unit:
  - Too few – can’t represent target function
  - Too many- leads to overfitting
  - Use cross validation to decide on the number of hidden units



Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep learning*. MIT press.

Parameter Tuning

Network Topology

- Universal approximation theorem: One layer of hidden unit suffices to approximate any function with finitely many discontinuities to arbitrary precision, if the activation functions of the hidden units are nonlinear.

- However, stationarity in the mean and variance for time series data is essential. Although theoretically they should be able to model any nonstationary process without requiring any transformation (due to universal approximation theorem).

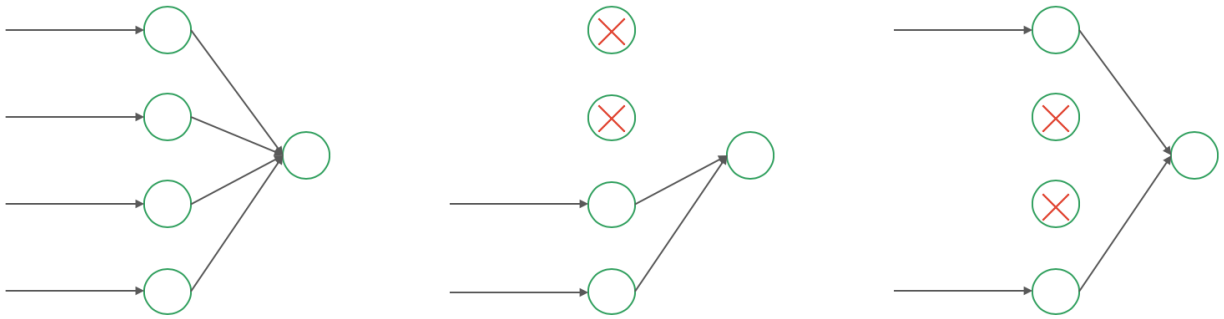
- It has been shown [1] that the logistic activation function restricts the model to a certain data range during training process. If the testing data is out of the training data range, the network is unable to generate an accurate estimate. In such cases, a trend removal process is mandatory prior to modeling.

[1] Cottrell, M., Girard, B., Girard, Y., Mangeas, M., & Muller, C. (1995). Neural modeling for time series: a statistical stepwise method for weight elimination. *IEEE transactions on neural networks*, *6*(6), 1355-1364.

Parameter Tuning

Drop out

- Various forms of regularization are used to prevent overfitting. One of the most important is dropout:
  - randomly dropping some units and their connections from the network during training.



a. Full network

b. Partial learning of weights over iterations

Co-adaptation: refers to when different hidden units in neural networks have highly correlated behavior.

It is better for computational efficiency and the model's ability to learn a general representation if hidden units can detect features independently of each other.

Loss Functions

- **Cross-entropy loss** is simply the log of the output probability corresponding to the correct class. Let *y* be a vector over the C classes representing the true output probability distribution.
- The cross-entropy loss here:

$$L_{CE}(\hat{y}, y) = -\sum_{i=1}^{C} y_i \log \hat{y}_i$$

We can simplify this equation further. Assume this is a hard classification task, meaning that only one class is the correct one, and that there is one output unit in *y* for each class. If the true class is *i*, then *y* is a vector where $y_i = 1$ and $y_j = 0$ $\forall j \neq i$.

$$\begin{aligned} L_{CE}(\hat{y}, y) &= -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log \hat{y}_i \\ &= -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log \hat{p}(y = k|x) \\ &= -\sum_{k=1}^{K} \mathbb{1}\{y = k\} \log \frac{e^{z_k}}{\sum_{j=1}^{K} e^{z_j}} \end{aligned}$$

Gradient Descent Variants

- There are three variants of gradient descent.
  - Batch gradient descent
  - Stochastic gradient descent
  - Mini-batch gradient descent
- The difference of these algorithms is the amount of data.

Update equation

$$\theta = \theta - \eta * \nabla_\theta J(\theta)$$

This term is different with each method.

- $J(\theta)$: Objective function
- $\theta \in R^d$: Model's parameters
- η: Learning rate. This determines the size of the steps we take to reach a (local) minimum.

Gradient Descent Variants

Batch Gradient Descent

- This method computes the gradient of the cost function with the entire training dataset.

Update equation

$$\theta = \theta - \eta * \nabla_\theta J(\theta)$$

We need to calculate the gradients for the whole dataset to perform just one update.

Code

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Gradient Descent Variants

Batch Gradient Descent

- Advantage:
  - It is guaranteed to converge to the global minimum for convex error surfaces and to a local minimum for non-convex surfaces.
- Disadvantage:
  - It can be very slow.
  - It is intractable for datasets that do not fit in memory.
  - It does not allow us to update our model online

Gradient Descent Variants

Stochastic Gradient Descent (SGD)

- This method performs a parameter update for each training example $x^{(i)}$ and label $y^{(i)}$.

Update equation

$$\theta = \theta - \eta * \nabla_\theta J(\theta; x^{(i)}; y^{(i)})$$

We need to calculate the gradients for a single point to perform just one update.

Code

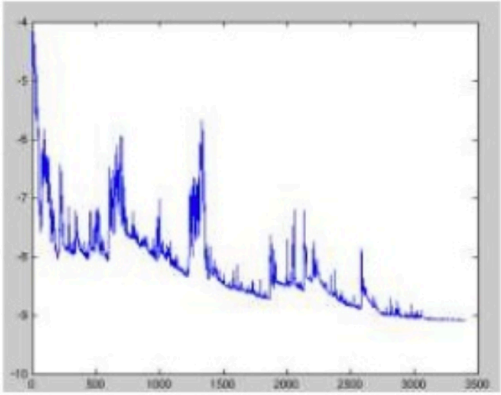
```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, example, params)
    params = params - learning_rate * params_grad
```

Note : we shuffle the training data at every epoch

Gradient Descent Variants

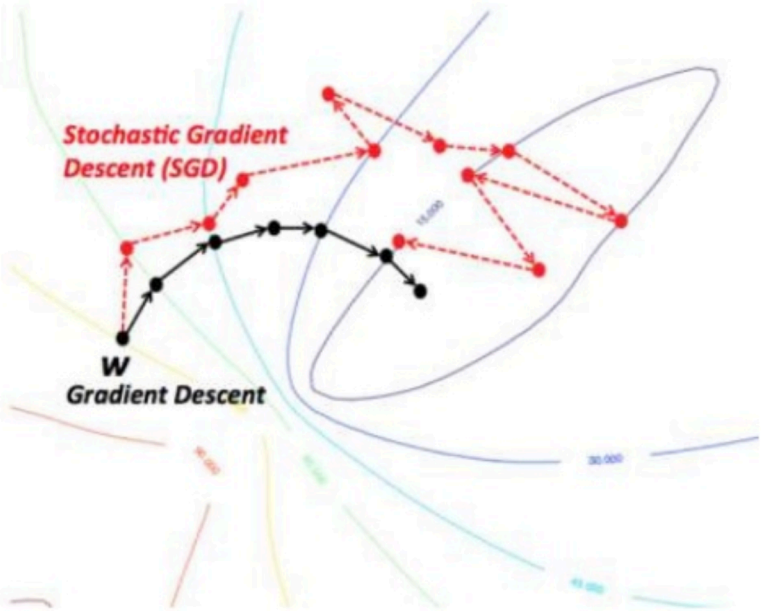
Stochastic Gradient Descent (SGD)

- Advantages:
  - It is usually much faster than batch gradient descent.
  - It can be used to learn online.
- Disadvantages:
  - It performs frequent updates with a high variance that cause the objective function to fluctuate heavily.



Gradient Descent Variants

The Fluctation: Batch vs SGD



https://wikidocs.net/3413

- Batch gradient descent converges to the minimum of the basin the parameters are placed in and the fluctuation is small.
- SGD's fluctuation is large but it enables to jump to new and potentially better local minima.
- However, this ultimately complicates convergence to the exact minimum, as SGD will keep overshooting.

Gradient Descent Variants

Mini-batch Gradient Descent

- This method takes the best of both batch and SGD, and performs an update for every mini-batch of *n*.

Update equation

$$\theta = \theta - \eta * \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

Code

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Gradient Descent Variants

Mini-batch Gradient Descent

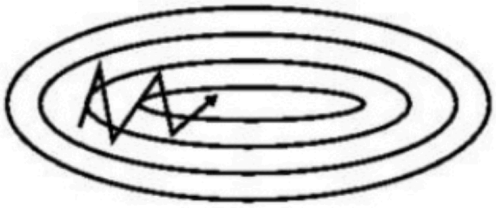
- Advantage:
  - It reduces the variance of the parameter updates.
    - This can lead to more stable convergence.
  - It can make use of highly optimized matrix optimizations common to deep learning libraries that make computing the gradient very efficiently.
- Disadvantage:
  - We have to set mini-batch size.
    - Common mini-batch sizes range between 50 and 256, but can vary for different applications.

Gradient Descent Variants

Momentum

- Momentum is a method that helps accelerate SGD.

(a) SGD without momentum



(b) SGD with momentum

The ball accumulates momentum as it rolls downhill, becoming faster and faster on the way.

- It does this by adding a fraction γ of the update vector of the past time step to the current update vector.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta = \theta - v_t$$

The momentum term γ is usually set to 0.9 or a similar value.

Gradient Descent Variants

Other Methods

- Nesterov accelerated gradient
- Adagrad
- Adadelta
- RMSprop
- Adam

Computation Graphs

- A computation graph is a representation of the process of computing a mathematical expression, in which the computation is broken down into separate operations, each of which is modeled as a node in a graph.
- Consider computing the function *L(a,b,c) = c(a+2b)*. If we make each of the component addition and multiplication operations explicit, and add names (*d* and *e*) for the intermediate outputs, the resulting series of computations is:

$$d = 2 \times b$$

$$e = a + d$$

$$L = c \times e$$

Computation Graphs

$d = 2 \times b$

$e = a + d$

$L = c \times e$

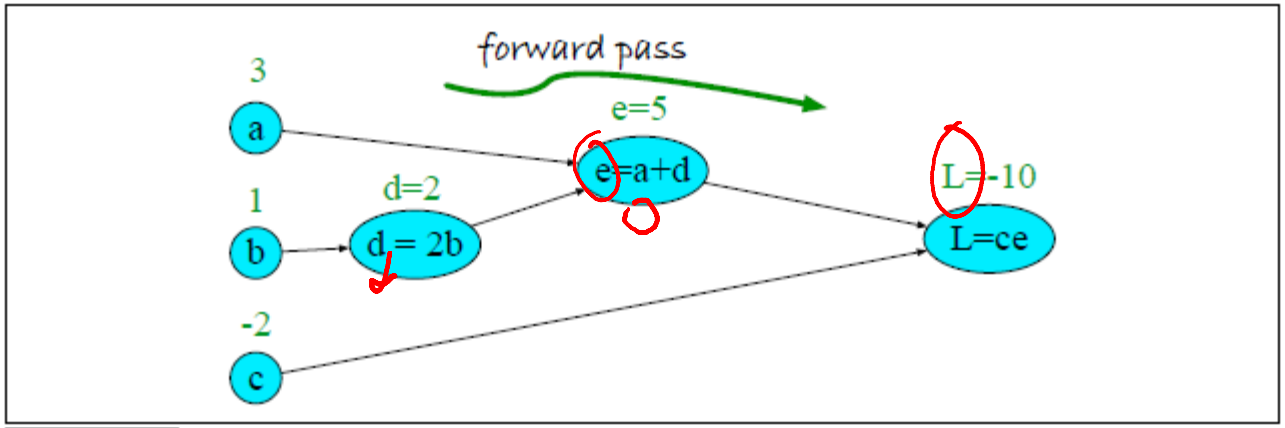


Figure 7.10 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

Computatio

$d = 2 \times b$

$e = a + d$

$L = c \times e$

$L = ce$: $\frac{\partial L}{\partial e} = c, \frac{\partial L}{\partial c} = e$

$e = a + d$: $\frac{\partial e}{\partial a} = 1, \frac{\partial e}{\partial d} = 1$

$d = 2b$: $\frac{\partial d}{\partial b} = 2$

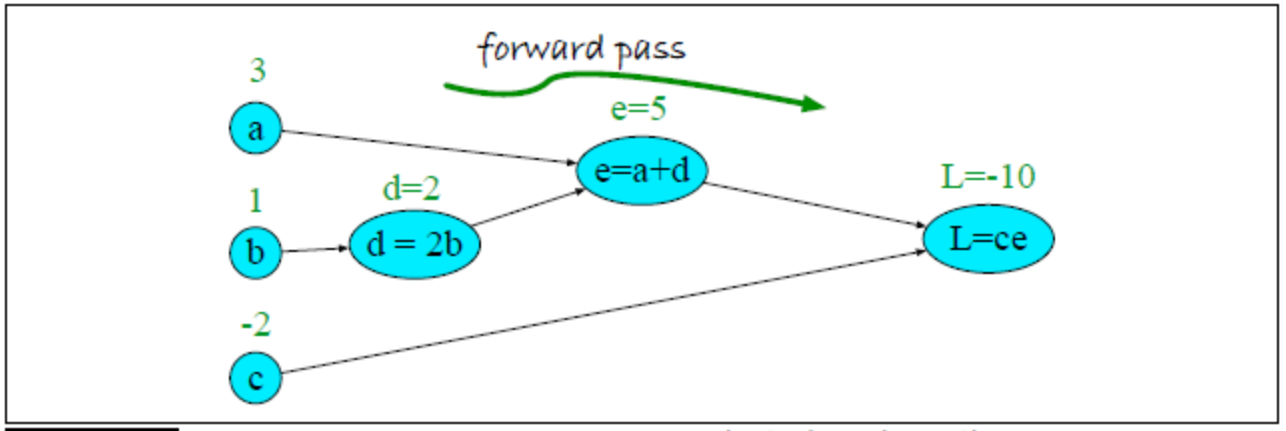


Figure 7.10 Computation graph for the function $L(a,b,c) = c(a+2b)$, with values for input nodes $a = 3$, $b = 1$, $c = -2$, showing the forward pass computation of L.

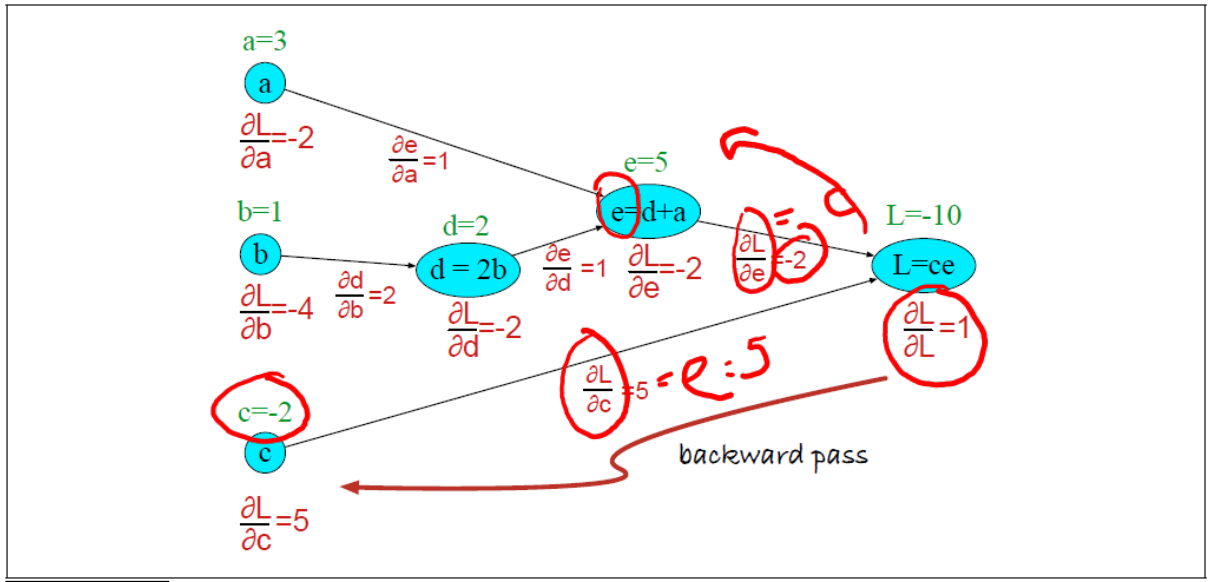


Figure 7.11 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

Computation Graphs

It is more complex for real neural networks.

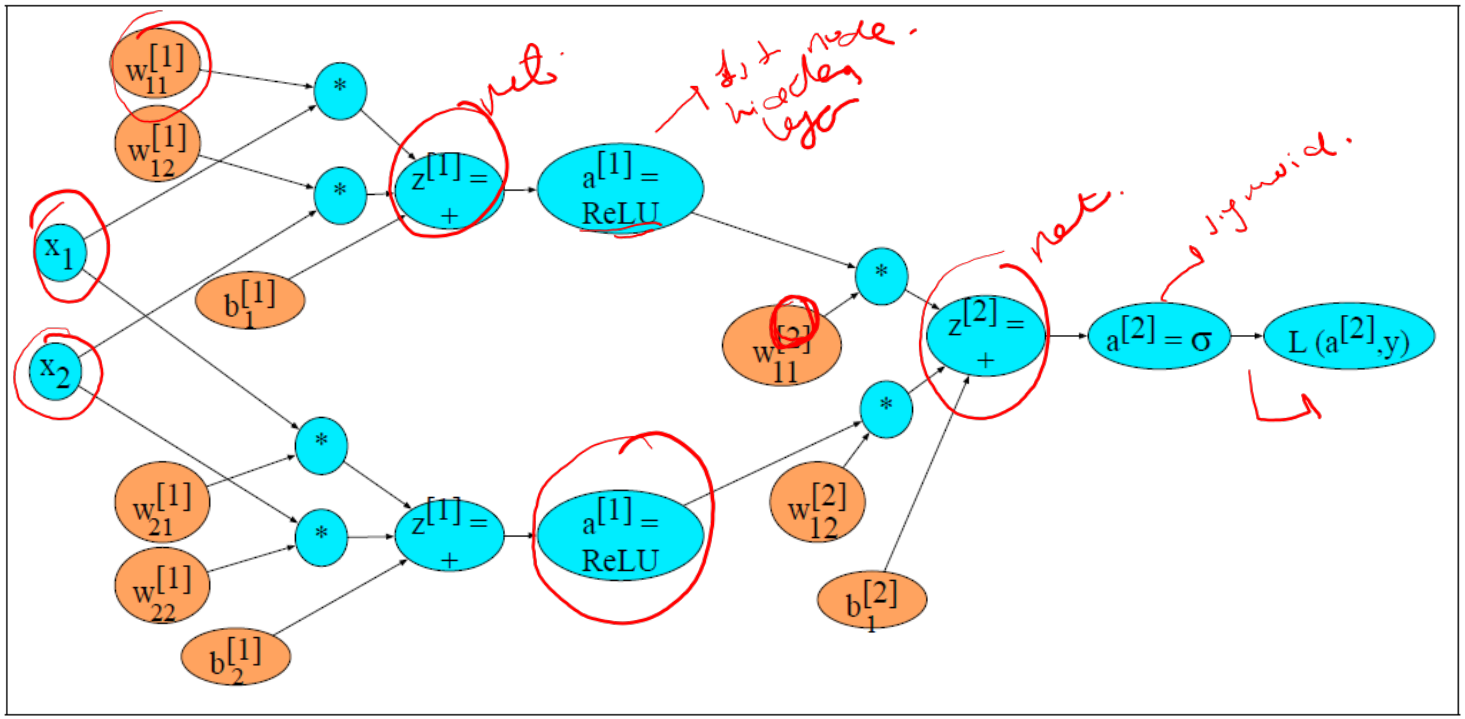


Figure 7.12 Sample computation graph for a simple 2-layer neural net (= 1 hidden layer) with two input dimensions and 2 hidden dimensions.