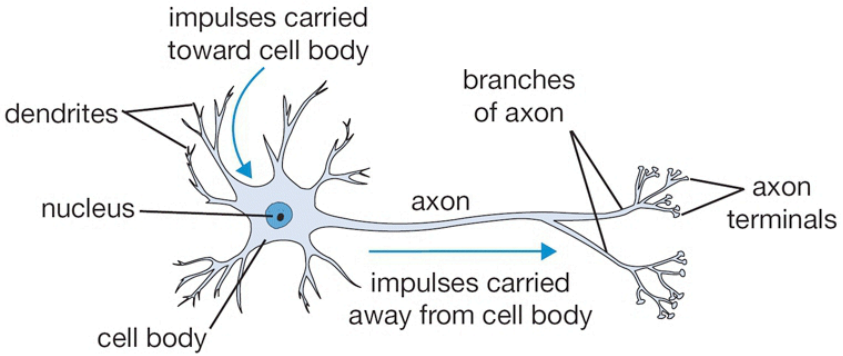
Neural Classifiers

Ishwar Sethi

Neural classifiers are built using artificial neurons. These classifiers are powerful and when integrated with automatic feature detectors as in deep learning can yield performance at par with humans. We will consider only single layer and multiple layer neural networks.

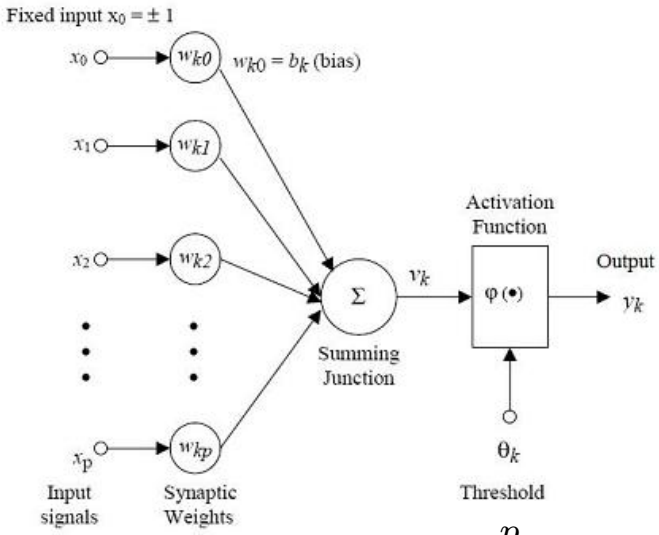
Biological Neuron

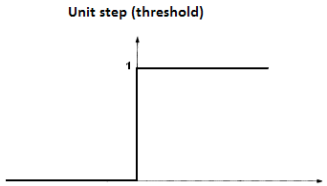
- A human brain has about 10 billion neurons, each connected to about 10,000 other neurons
- Each neuron receives electrochemical inputs from other neurons at dendrites
- A neuron will fire if the sum of its inputs exceeds certain level. The firing causes an electrochemical impulse that is carried to other connected neurons via axons



Artificial Neuron Model

- McCulloch-Pitts neuron model [circa 1945]





aka Net → $v_k = \sum_{j=0}^{p} w_{kj} x_j$

Artificial Neuron Model

- McCulloch-Pitts neuron model

Inputs

1

0.5

0.8

0.2

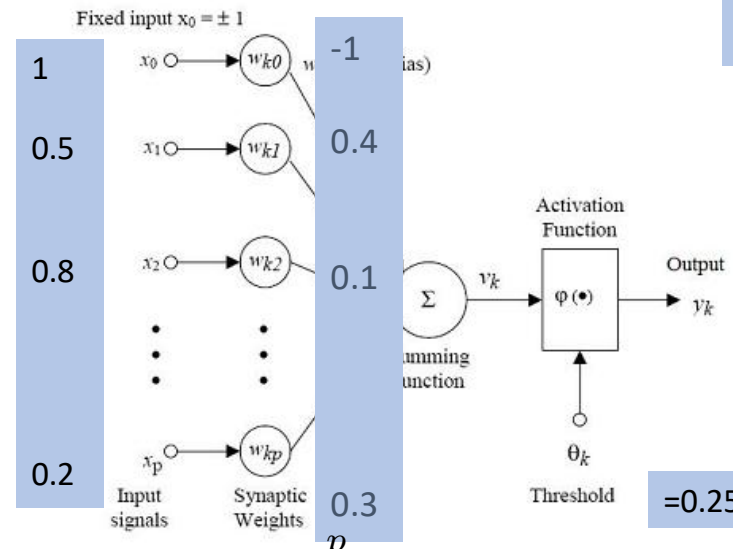
Weights

-1

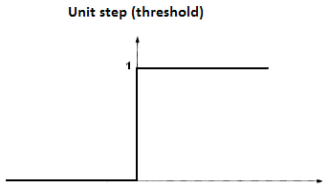
0.4

0.1

0.3



Showing output calculation for a set of inputs and weights



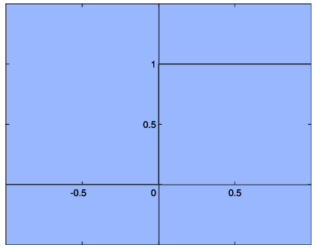
$$v_k = \sum_{j=0}^{p} w_{kj} x_j$$

=0.25

=1*(-1)+ 0.5*0.4 + 0.8*0.1 + 0.2*0.3 = -0.66
Since -0.66 is less than the threshold 0.25, the output will be 0.

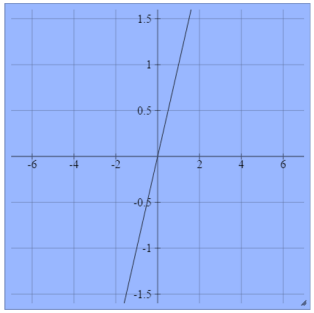
Activation Functions

- The original model used unipolar threshold nonlinearity (*sign* function) as an activation function.



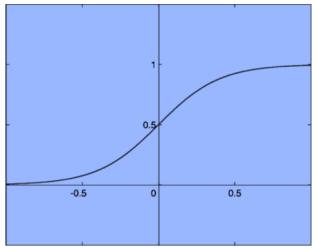
Unipolar Threshold Function

Output = 1 if *net* > 0
= 0 if *net* < 0



Linear Function

Output = net

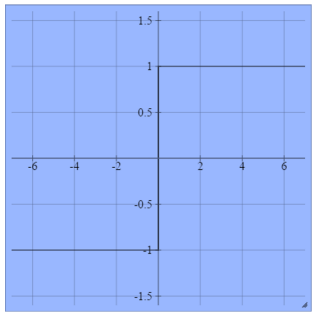


Sigmoid Function

Output = 1/(1+ exp(-net))

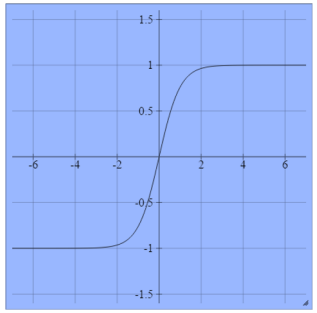
The sigmoid function was very popular earlier, but it is less often used in deep learning because it saturates and kills gradients. Also, the output range 0-1 seem to create learning difficulties for later layers. It is also known as logistic function

Activation Functions



Signum Function

Output = 1 if *net* > 0
= -1 if *net* < 0

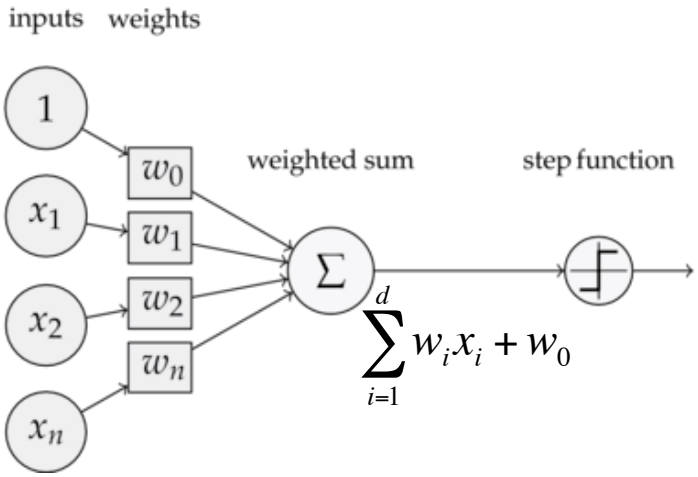


Hyperbolic Tangent Function

Output = (1- exp(-net))/(1+ exp(-net)

Like the sigmoid neuron, its activations saturate, but unlike the sigmoid neuron its output is zero-centered. Therefore, in practice the tanh non-linearity is always preferred to the sigmoid nonlinearity.

Perceptron Classifier



$$\sum_{i=1}^{d} w_i x_i + w_0 \geq 0 \text{ Then Output} = 1$$

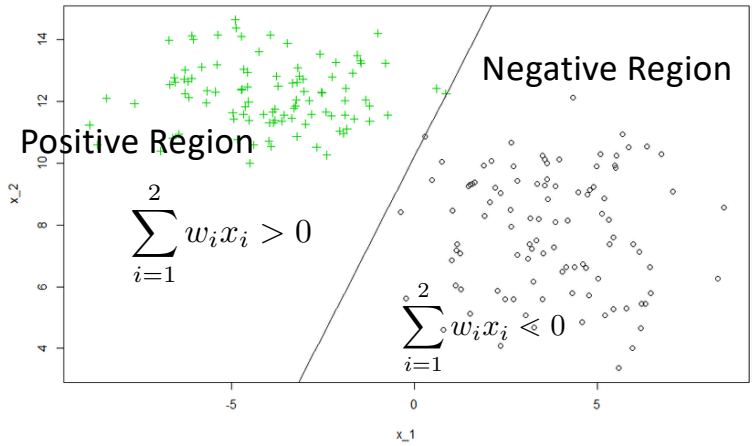
$$\sum_{i=1}^{d} w_i x_i + w_0 < 0 \text{ Then Output} = -1$$

A perceptron classifier sums the weighted input and generates an output of +1 if the weighted sum is positive; otherwise an output of -1 is generated. Designing/modeling a perceptron classifier involves finding weights given a collection of training examples from two classes.

What kind of Decision Boundary is produced by a Perceptron Classifier?

- The equation for decision boundary is:

$\sum_{i=1}^{d} w_i x_i + w_0 = 0$ This is a linear equation. With two features, the decision boundary is a line. With three or more features it is a hyperplane



Recalling the line equation form y = mx + c, you can note that the bias weight (w_o) basically determines the location of the boundary. The orientation of the boundary is given by the weights (w_1 , w_2) associated with different features.

How do we determine the weights?

- Given a collection of training examples, the weights of a perceptron classifier are determined by an algorithm, known as the *perceptron learning algorithm*.
- It is an *error-correction algorithm*. It means the weight values are updated only when the perceptron classifier makes an error on a given example.
- The algorithm is given in the next slide.

Perceptron Learning Rule

- Uses training data from two classes. The training vectors are augmented by an additional component with value 1. This is needed to include the bias term in the calculations.
- Initialize the weight vector randomly
- Compute the perceptron output for each training example one at a time
- If the output matches with the desired output, leave the weight vector unchanged and move to the next example
- If the output is not correct, modify the weight vector either by adding to it or by subtracting from it the current training vector, and move to the next example
- Shuffle and continue repeating until perceptron produces correct output for all examples.

The perceptron output is calculated by computing the weighted sum of the input signals and comparing the result with a threshold value. If the net input is less than the threshold, the perceptron output is –1. But if the net input is greater than or equal to the threshold, the perceptron becomes activated and its output attains a value +1.

Perceptron Learning Example

We have four examples in two-dimensions. Positive Examples: $A=(1,1)^t$, $B=(2,3)^t$; Negative Examples: $C=(3,1)^t$, $D=(4,2)^t$;

With the addition of the bias term, the examples are: $A=(1,1,1)^t$, $B=(1,2,3)^t$, $C=(1,3,1)^t$, $D=(1,4,2)^t$

The training begins with an arbitrary initial weight vector. Let's choose W0= (1,1,1)

W0*A = 3 (ok), W0*B = 6 (ok), W0*C=5 (not ok, the result should be negative. So we correct weight vector)

$W1 = W0-C^t=(0,-2,0)$, Corrected/updated weight vector

W1*D=-8 (ok)

$W1*A=-2$ (not ok, apply correction), $W2=W1+A^t=(1,-1,1)$

W2*B=2 (ok), W2*C=-1(ok), W2*D=-1 (ok), W2*A=1(ok)

All examples are correctly classified. The solution is (1,-1,1)

If examples are not linearly separable, then learning will not cease.

Ok implies that the result is correct, i.e. positive for + examples and negative for - examples

We are presenting the examples without shuffling. In practice, shuffling is done to randomly change the order of training examples.

Understanding the Perceptron Learning Rule

- Let's look at the augmented vectors. The perceptron classification rule without augmentation is:

$$\sum_{i=1}^{d} w_i x_i + w_0 \geq 0 \text{ Then Output} = 1$$

$$\sum_{i=1}^{d} w_i x_i + w_0 < 0 \text{ Then Output} = -1$$

Define $\mathbf{y}^t = [\mathbf{x}^t 1] = [x_1 x_2 \ldots x_d 1]$

$\mathbf{a}^t = [\mathbf{w}^t 1] = [w_1 w_2 \ldots w_d 1]$

Then

$\mathbf{a}^t\mathbf{y} \geq 0$ Then Output = 1

$\mathbf{a}^t\mathbf{y} < 0$ Then Output = -1

Augmented pattern and weight vectors. Augmentation makes equations simpler.
Augmentation can be done at the head/tail of the vector

Defining Augmented Vector Training

- Let $\boldsymbol{y}_1, \boldsymbol{y}_2, \ldots, \boldsymbol{y}_n$ be a set of n examples in augmented feature space, which are linearly separable.
- We need to find a weight vector **a** such that
  - $\mathbf{a}^t\mathbf{y} > 0$ for examples from the positive class.
  - $\mathbf{a}^t\mathbf{y} < 0$ for examples from the negative class.
- Normalizing the input examples by multiplying them with their class label (replace all samples from class 2 by their negatives), find a weight vector **a** such that
  - $\mathbf{a}^t\mathbf{y} > 0$ for all the examples (here **y** is multiplied with class label)
- Such a weight vector is called a *separating vector* or a *solution vector*

Linearly separable means there exists a linear decision boundary to separate training data in to two classes without any error

Perceptron Criterion Function

- The perceptron learning algorithm is based on minimizing the following criterion function

- Perceptron Criterion Function:

Find an **a** that minimizes this criterion.

$$J_p(\mathbf{a}) = \sum_{\mathbf{y} \in \mathcal{Y}} (-\mathbf{a}^t \mathbf{y})$$

Set of misclassified training examples

The criterion is proportional to the sum of distances from the misclassified samples to the decision boundary

The minimization is carried out by gradient descent; an iterative method for finding minima of functions.

Perceptron Learning Algorithm

- The gradient descent-based minimization can be applied in the batch mode or in the online mode.
- In the batch mode, all misclassified training examples are first identified and then the weight vector is updated. This mode is good for small training sets.
- In the online mode, examples are considered one by one and the weight correction is applied as soon as a misclassification occurs. This is the mode we followed in the example shown earlier. This online procedure for perceptron learning can be described as below. It is often called *fixed-increment online learning rule*.

```
1 begin initialize  a, k = 0
2           do  k ← (k + 1) mod n
3               if  y_k is misclassified by a  then  a ← a + y_k
4           until all patterns properly classified
5     return a
6 end
```

Iterative update step. Notice the gradient.

An intuitive way to understand the perceptron learning algorithm is that the weight correction pushes a misclassified example towards the boundary hoping that it crosses over the boundary

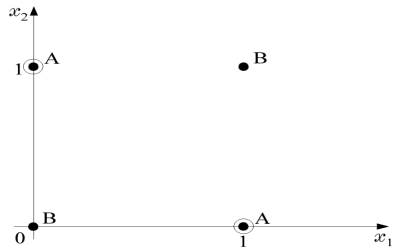
What happens when the examples are not linearly separable?

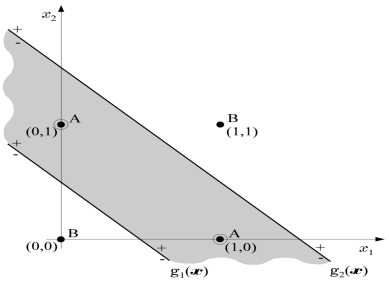
- First point: There is no way to determine if the training data/problem is a linearly separable except when working in two-dimensions.
- Two possible options to deal with when linear separability is an issue
  - Use layers of perceptrons [Shown in coming slides]
  - Modify the criterion for optimization so that we get an error minimization procedure rather than an error correction procedure.

Using Layers of Perceptrons

- Let's look at the following four examples from two classes, A and B

X_1	X_2	C
0	0	B
0	1	A
1	0	A
1	1	B





We see that points from class A cannot be separated by a line from the points from class B. Thus, it is an example of not linearly separable problem.

However, we can draw two lines to separate points from A and B. Then class B is located outside the shaded area and class A inside.

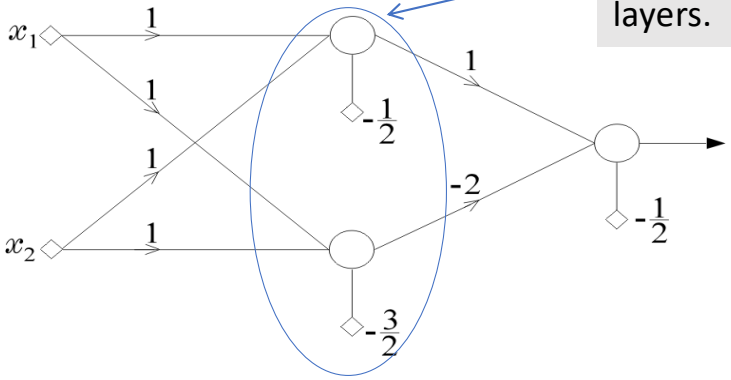
Layers of Perceptrons Example

- One can look at the arrangement of two lines from the prior slide in two steps:
  - Step 1: The first stage perceptrons perform a mapping of input producing output depending on input's position. Intermediate output
  - Step 2: Use the intermediate output to do classification

1st Stage				2nd Stage
x_1	x_2	y_1	y_2	
0	0	0(-)	0(-)	B(0)
0	1	1(+)	0(-)	A(1)
1	0	1(+)	0(-)	A(1)
1	1	1(+)	1(+)	B(0)

- Computations of the first phase perform a mapping that transforms the nonlinearly separable problem to a linearly separable one.

- The architecture

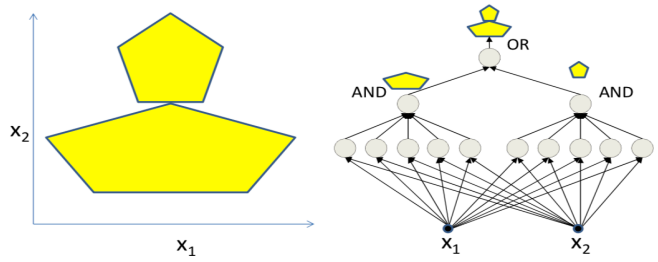


Hidden layer. Layers that are not input or output layers.

- The idea can be extended by having multiple layers

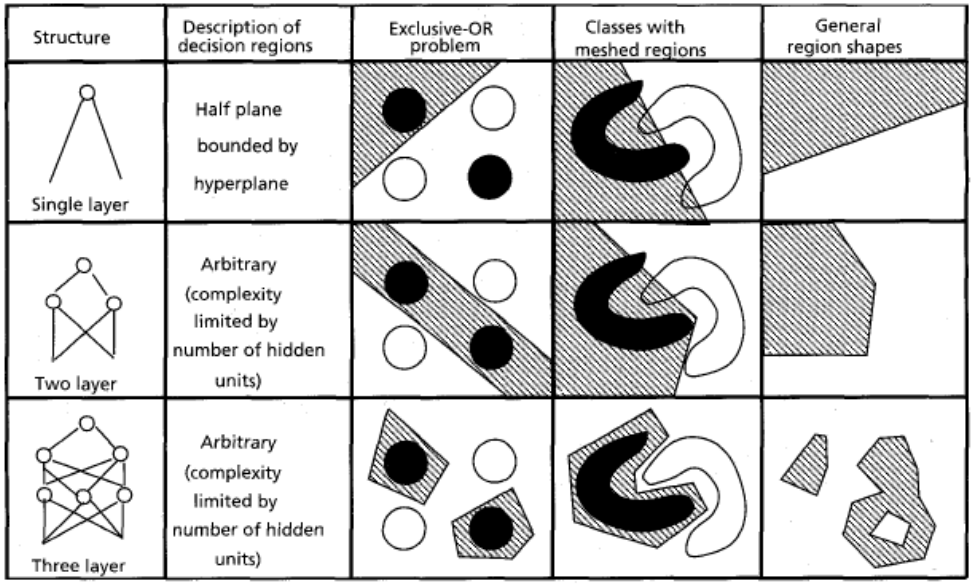
Multiple Layers of Perceptrons

- The previous example showed two layers. But the idea can be extended to multiple layers. In the picture below, two sets of five neurons each produce the two yellow regions. These are then combined to produce the final boundary



- Network to fire if the input is in the yellow area
  - "OR" two polygons
  - A third layer is required

A geometric interpretation of the role of hidden unit in a two-dimensional input space.



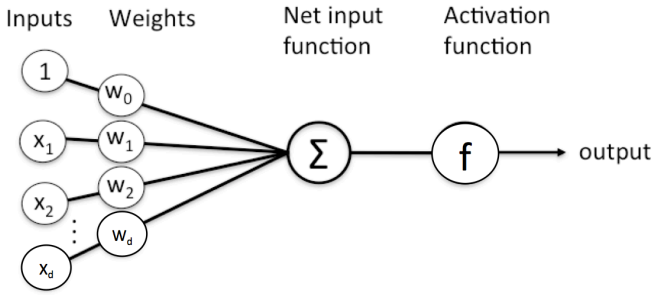
So the question is "how do we train perceptrons in multiple layers?

Answer: Make the activation function smooth and redefine the optimization criterion

There is a mathematical proof that states that any arbitrary function can be approximated with a single hidden layer network. Hence neural nets are called *universal approxima*tors.

Sigmoidal Neuron

- Let's use sigmoidal activation function in our single neuron model. Let the output be given as $\hat{y}$ as shown below.



$$\hat{y} = \frac{1}{(1 + exp(-\mathbf{w}^t\mathbf{x}))}$$

MSE Criterion

- In perceptron learning, the optimization criterion was focused on misclassified examples. In MSE (mean square error) loss function, the focus is on difference between the desired output and the output of the summation feeding into the sigmoidal activation.
- The MSE loss function is expressed as:

$$J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \mathbf{w}^t\mathbf{x}_i)^2$$

Weight Updating using MSE Loss Function

- We can take the gradient of the loss function to find the weight updating rule as shown below.

$$\frac{\partial J}{\partial \mathbf{w}} = -\sum_{i=1}^{n}(y_i - \mathbf{w}^t \mathbf{x}_i)\mathbf{x}_i$$

Again, the updating can be done in the batch mode or in the online mode. Let's focus on the online mode.

LMS (Least Mean Square) Rule

- Let's do solution updating after every training example. In that case:

$$\frac{\partial J_i}{\partial \mathbf{w}} = -(y_i - \mathbf{w}^t \mathbf{x}_i)\mathbf{x}_i$$

- Thus, updating rule becomes

$$\mathbf{w}_{new} = \mathbf{w}_{old} + \alpha(y_i - \mathbf{w}_{old}^t \mathbf{x}_i)\mathbf{x}_i$$

Error correction versus error minimization learning? This rule doesn't correct errors as is done in the perceptron learning. Rather, it tries to minimize the error.

Alpha is called the learning rate; it is typically a very small number, for example 0.1.

This updating rule is also known *Widrow-Hoff /LMS/Delta rule*

Redefining Loss Function

- Let us modify the earlier MSE loss function and compare the actual output, i.e. output of the sigmoidal function, with the target(desired) output. Thus the loss function can be expressed as

$$J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- Computing the gradient, we have

$$\frac{\partial J}{\partial \mathbf{w}} = -\sum_{i=1}^{n}(y_i - \hat{y}_i)\frac{\hat{y}_i}{\partial \mathbf{w}}$$

Refresher on the derivative of an exponential function. f' implies the first derivative of f with respect to z.

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

$$f^{'}(z) = \frac{1}{(1 + exp(-z)} \frac{exp(-z)}{1 + exp(-z)} = output * (1 - output)$$

We are going to use this in the next slide

Redefined MSE Loss Function

- Since $\hat{y} = \frac{1}{(1 + exp(-\mathbf{w}^t \mathbf{x}))}$

- We get $\frac{\partial \hat{y}_i}{\partial \mathbf{w}} = \hat{y}_i * (1 - \hat{y}_i) * \mathbf{x}_i$

- Thus for a single input example, we can write

$$\frac{\partial J}{\partial \mathbf{w}} = (y_i - \hat{y}_i) * (\hat{y}_i * (1 - \hat{y}_i)) * \mathbf{x}_i$$

Let's look at the following expression

$$\frac{\partial J}{\partial \mathbf{w}} = (y_i - \hat{y}_i) * (\hat{y}_i * (1 - \hat{y}_i)) * \mathbf{x}_i$$

Error output (1-output) input

We can write the RHS as:
Error* output*(1-output)*input. [Easy to remember]

Thus, the formula for changing the j-th component of the weight vector for the i-th training vector can be expressed as

$\boldsymbol{\delta} w_j$ = - error*output*(1-output)*x_{ij}

The negative sign is because we want to decrease the error. The highlighted part of the formula will be different if the activation function is the hyperbolic tangent function. This rule is known as the generalized delta rule

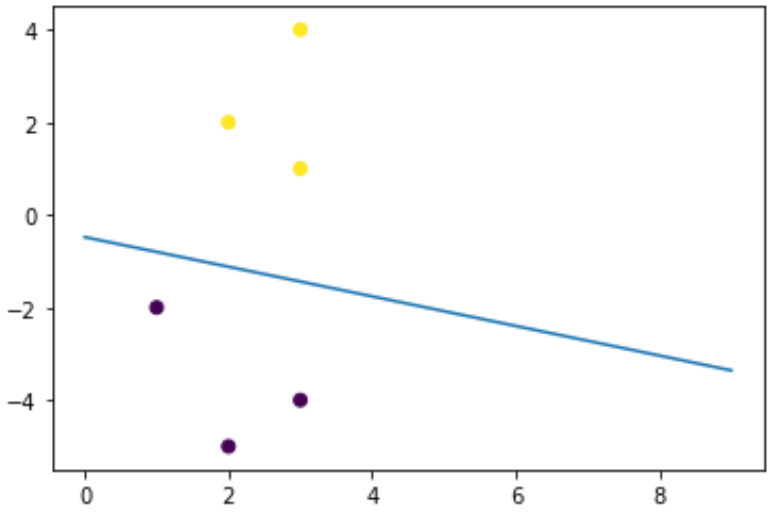
Generalized Delta Rule Example

```
import numpy as np
from numpy import linalg as LA
import math
X = np.array([[1,2,2],[1,3,1],[1,3,4],[1,3,-
4],[1,1,-2],[1,2,-5]])
Y = np.array([1,1,1,0,0,0])
alpha = 0.025
w_old = np.array([0.3,0.2,0.6])
for i in range (2):
    for j in range(6):
        net = np.matmul(w_old,X[j].T)
        out = 1/(1+np.exp(-net))
        err = (Y[j]-out)
        grad_old = err*out*(1-out)*X[j]
        w_new = w_old + alpha*grad_old
        print(w_new)
        w_old = w_new
    print(err)
```

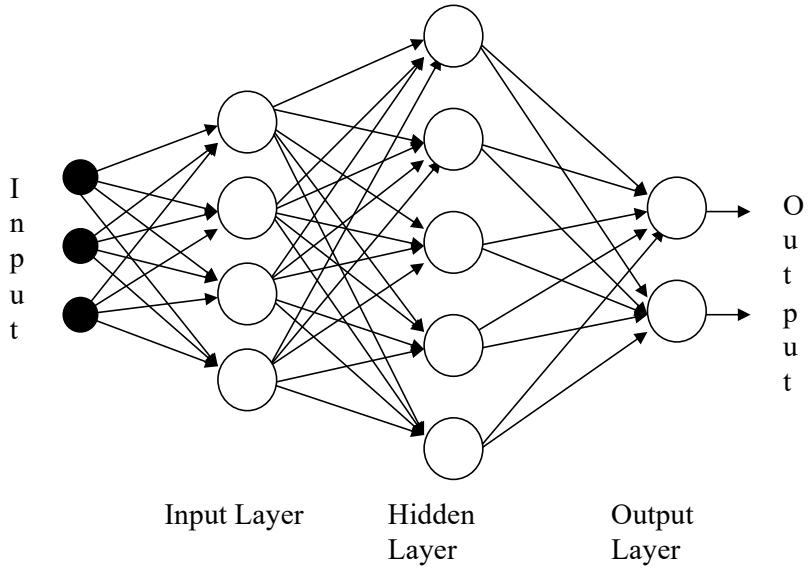
X: Training examples
Y: Target outputs
Starting value for the weight vector is arbitrary

```
[0.30036814 0.20073629 0.60073629]
[0.30104508 0.20276709 0.60141322]
[0.30107473 0.20285605 0.60153183]
[0.30039105 0.200805 0.60426657]
[0.29856541 0.19897936 0.60791784]
[0.2983903 0.19862914 0.60879339]
-0.08761857731269952
[0.29875067 0.19934989 0.60951413]
[0.29942466 0.20137184 0.61018812]
[0.29945271 0.201456 0.61030032]
[0.29880866 0.19952384 0.61287652]
[0.2970199 0.19773508 0.61645404]
[0.29685848 0.19741225 0.61726114]
-0.08395533777234554

print(np.matmul(w_old,X.T))

[ 1.92620525 1.50635636 3.35813977 -1.57994933 -0.74025155 -2.39462271]
```

```
import matplotlib.pyplot as plt
plt.plot(xp,yp)
plt.scatter(X[:,1],X[:,2], c = Y[:])
plt.show()
```



Multiple-Layer Feedforward Network Model



One can use many hidden layers.

These networks are constructed using smooth activation functions, e.g. sigmoid function.

Multiple-Layer Feedforward Network Model

- Also known as MLP (Multiple Layer Perceptrons) networks and BP (Backpropagation) networks
- Capable of generating arbitrarily complex models
- Error-minimization learning (Backpropagation learning)
- Long training time
- Performance affected by network size

How Do We Train a Multilayer Neural Network?

- In supervised learning, we are given a collection of vector pairs { $\mathbf{x}_i$, $\mathbf{y}_i$}
- Each $\mathbf{x}_i$ is a vector of real numbers of some suitable dimensionality
- Each $\mathbf{y}_i$ is a vector with components as 0 or 1 (when sigmoidal activation function is used) or -1 or 1 (when tanh is used). When the network has a single output, then each $\mathbf{y}_i$ is a scalar
- We can set up a loss function at the output and use gradient descent to adjust the weights at the output layer to minimize the loss function
- The gradient at the output is fed back to internal layers to adjust their weights. This feeding back of the gradient to internal layers is based on the chain-rule of differentiation and the resulting algorithm is known as backpropagation.

Our presentation on this topic is done without much math and derivations. Just the key points are highlighted.

Refreshing Chain Rule of Differentiation from Your Highschool Days

$$f(y) = sin(y),\ y = g(x) = 0.5x^2$$

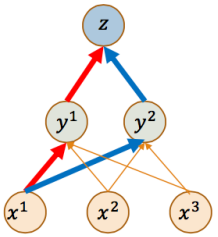
$$\frac{df}{dx} = ?$$

$$\frac{df}{dx} = \frac{df}{dy}\frac{dy}{dx} = cos(0.5x^2)\ x$$

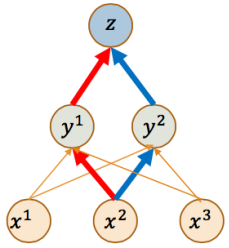
Understanding Chain Rule

- Assume a nested function, $z = f(y)$ and $y = g(x)$
- Chain Rule for scalars x, y, z
  - $\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$
- When $x \in \mathcal{R}^m, y \in \mathcal{R}^n, z \in \mathcal{R}$
  - $\frac{dz}{dx_i} = \sum_j \frac{dz}{dy_j}\frac{dy_j}{dx_i}$ → gradients from all possible paths

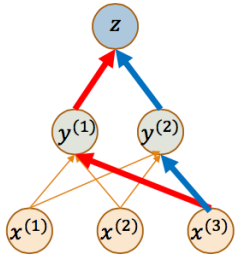
Basically, these pictures show that *z* depends upon y's which in turn depend on x's. So getting a desired value for *z* is like tuning the flow of y's and x's.



$$\frac{dz}{dx^1} = \frac{dz}{dy^1}\frac{dy^1}{dx^1} + \frac{dz}{dy^2}\frac{dy^2}{dx^1}$$



$$\frac{dz}{dx^2} = \frac{dz}{dy^1}\frac{dy^1}{dx^2} + \frac{dz}{dy^2}\frac{dy^2}{dx^2}$$



$$\frac{dz}{dx^3} = \frac{dz}{dy^1}\frac{dy^1}{dx^3} + \frac{dz}{dy^2}\frac{dy^2}{dx^3}$$

MLP Training

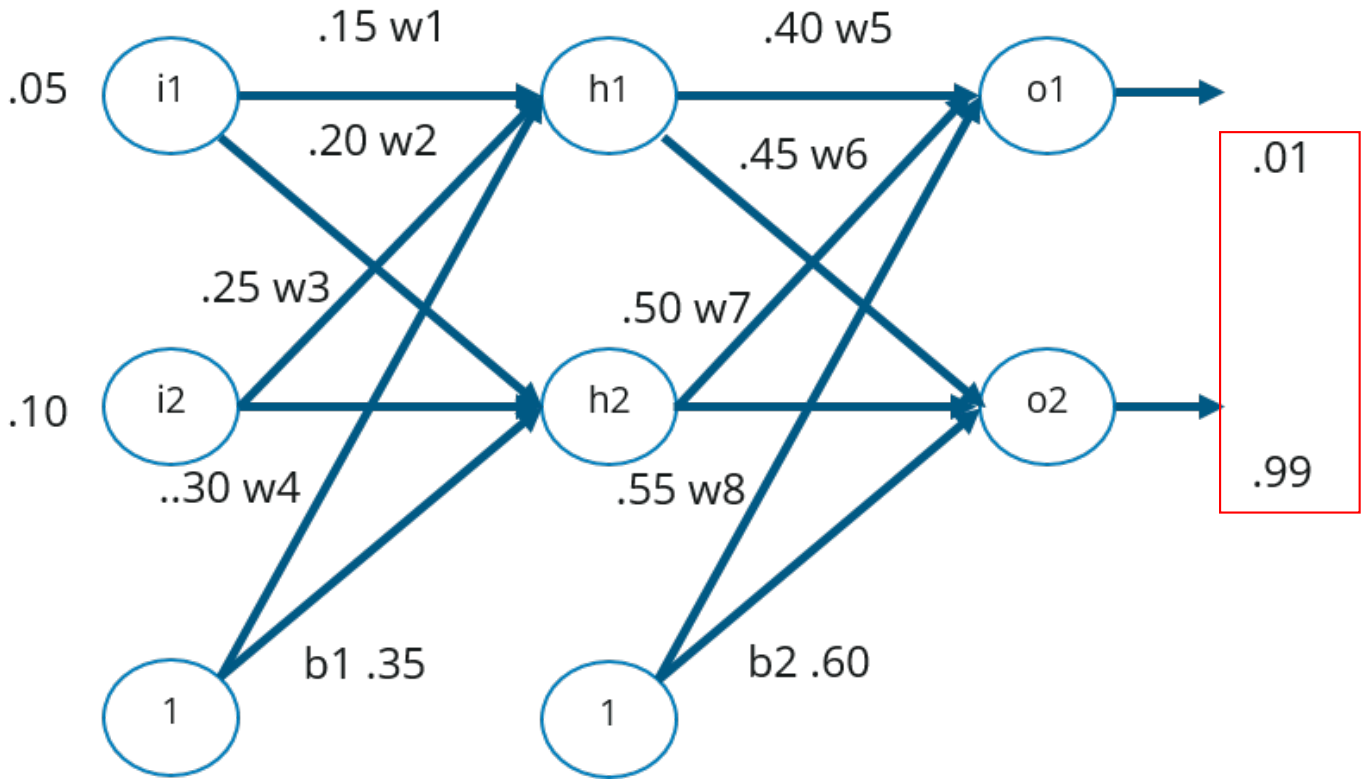
- Works in two phases
  - Feedforward phase
  - Backpropagation phase

- During test/usage phase, only feedforward phase is needed to produce the output

Feedforward Operation

- Each hidden unit computes the weighted sum of its inputs to form a net activation value, $net = \mathbf{w}^T\mathbf{x}$.
- Then, it emits an output that is a nonlinear function of its activation, $f(net)$.
- Each output unit similarly computes its net activation based on the hidden unit signals in the previous layer, and emits a value using a nonlinear function based on its activation.
- The output corresponds to a nonlinear function of the input feature vector.

Feedforward Computation Illustration



A network with 2 inputs, 2 hidden layer neurons, and 2 output neurons. The weight values are shown just before the weight symbols. The numbers at the output neurons are target output values.

Hidden Layer Output Calculations

Net Input For h1:

net h1 = w1*i1 + w2*i2 + b1*1 → net h1 = 0.15*0.05 + 0.2*0.1 + 0.35*1 = 0.3775

Output Of h1:

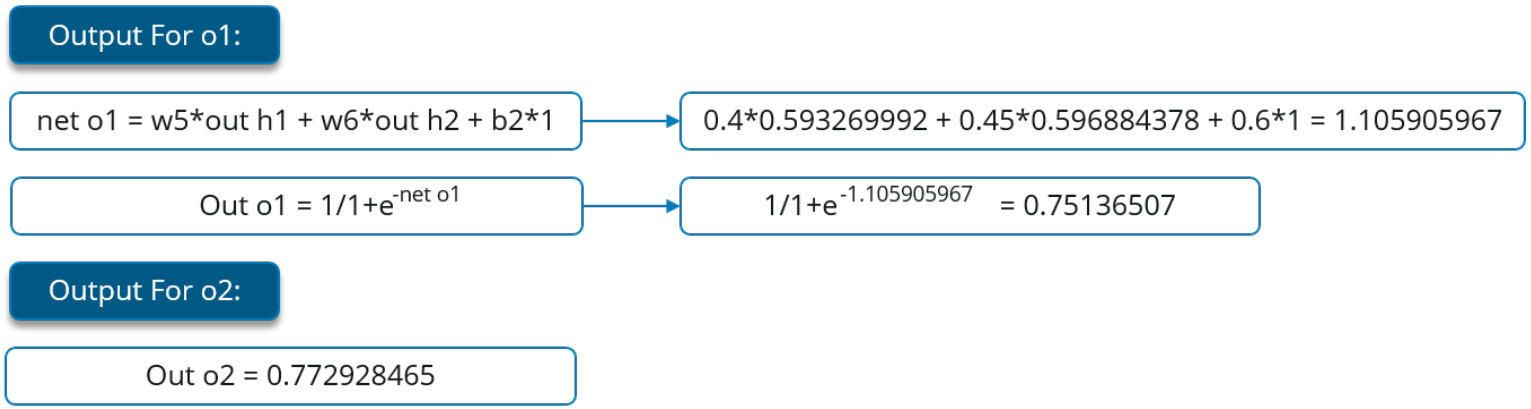
out h1 = $1/1+e^{-\text{net h1}}$ → $1/1+e^{.3775}$ = 0.593269992

Output Of h2:

Similarly we get the h2 output

out h2 = 0.596884378

Calculating the Final Output



Calculating Error

Error For o1:

$E_{o1} = \Sigma 1/2(target - output)^2$ → $\frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$

Error For o2:

$E_{o2} = 0.023560026$

This completes the forward pass

Total Error:

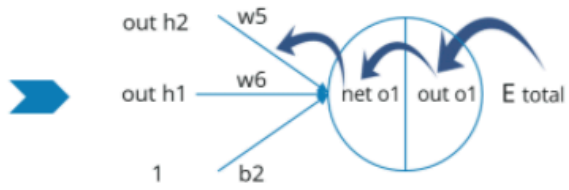
$E_{total} = E_{o1} + E_{o2}$ → $0.274811083 + 0.023560026 = 0.298371109$

Step – 2: Backward Propagation

Now, we will propagate backwards. This way we will try to reduce the error by changing the values of weights and biases.

Consider W5, we will calculate the rate of change of error w.r.t change in weight W5.

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5}$$



Since we are propagating backwards, first thing we need to do is, calculate the change in total errors w.r.t the output O1 and O2.

$$E_{total} = 1/2(\text{target o1} - \text{out o1})^2 + 1/2(\text{target o2} - \text{out o2})^2$$

$$\frac{\delta Etotal}{\delta out\ o1} = -(\text{target o1} - \text{out o1}) = -(0.01 - 0.75136507) = 0.74136507$$

Now, we will propagate further backwards and calculate the change in output O1 w.r.t to its total net input.

$$\text{out o1} = 1/1+e^{-neto1}$$

$$\frac{\delta out\ o1}{\delta net\ o1} = \text{out o1 (1 - out o1)} = 0.75136507\ (1 - 0.75136507) = 0.186815602$$

Let's see now how much does the total net input of O1 changes w.r.t W5?

$$\text{net o1} = w5 * \text{out h1} + w6 * \text{out h2} + b2 * 1$$

$$\frac{\delta net\ o1}{\delta w5} = 1 * \text{out h1}\ w5^{(1-1)} + 0 + 0 = 0.593269992$$

Step – 3: Putting all the values together and calculating the updated weight value

Now, let's put all the values together:

$$\frac{\delta Etotal}{\delta w5} = \frac{\delta Etotal}{\delta out\ o1} * \frac{\delta out\ o1}{\delta net\ o1} * \frac{\delta net\ o1}{\delta w5} \rightarrow 0.082167041$$

Let's calculate the updated value of W5:

$$w5^{+} = w5 - n\frac{\delta Etotal}{\delta w5} \rightarrow w5^{+} = 0.4 - 0.5 * 0.082167041$$

Learning rate is 0.5

Updated w5 → 0.35891648

In a similar fashion, we can calculate weight changes for W6 to W8. Once done that, we apply similar calculations from the hidden layer to the input layer and calculate changes for W1 to W4.

Backpropagation Algorithm

- All weights are randomly initialized.
- We go over the forward and the backward passes using the training set adjusting the weights
- During every pass, the training data is shuffled
- Training stops when the error reaches an acceptable level.

Practical Issues in Training: Initializing Weights

- The weights are initialized randomly but it is a good practice to choose hidden unit weights in the range of [-1/sqrt(*d*) 1/sqrt(*d*)], *d*: dimensionality of the input vectors
- A good choice for the weights of the output units is the range [-1/sqrt(# of hidden units) 1/sqrt(# of hidden units)]
- The above choices for initialization ensure
  - All units learn at same pace
  - Learning typically will begin from a state where each unit is operating in the linear part of its activation function

Practical Issues in Training: Number of Hidden Units

- The number of hidden units influences learning rate and generalization accuracy
- Fewer hidden units imply relatively simpler decision boundaries. So in some sense the complexity of the problem determines the number of hidden units needed
- More hidden units mean more weights to be determined and hence more training data
- A popular rule of thumb is to choose the number of hidden units such that there are roughly 10 training examples per weight

Practical Issues in BP Training: When to Stop Training?

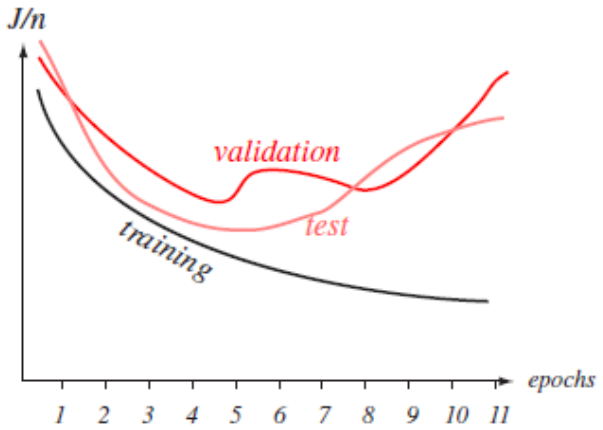


FIGURE 6.6. A learning curve shows the criterion function as a function of the amount of training, typically indicated by the number of epochs or presentations of the full training set. We plot the average error per pattern, that is, $1/n \sum_{p=1}^{n} J_p$. The validation error and the test or generalization error per pattern are virtually always higher than the training error. In some protocols, training is stopped at the first minimum of the validation set. From: Richard O. Duda, Peter E. Hart, and David G. Stork, *Pattern Classification*. Copyright © 2001 by John Wiley & Sons, Inc.

The validation data set can be used to determine when to stop training.

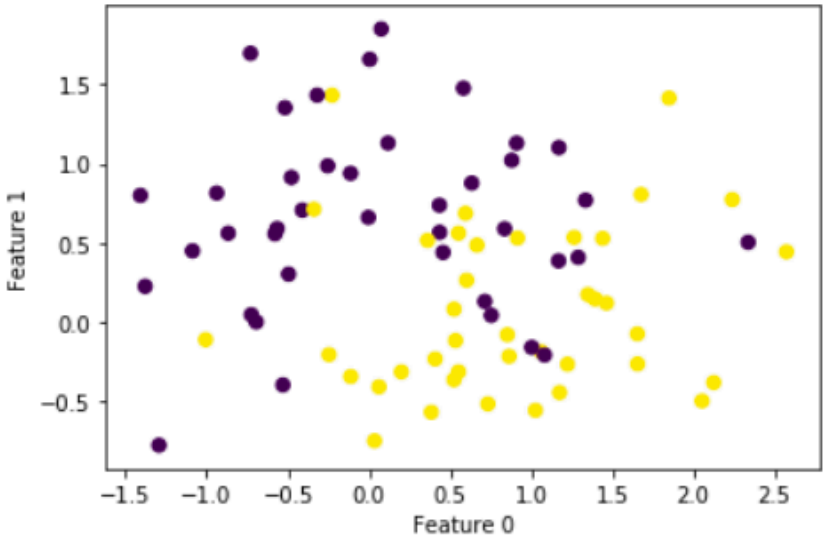
Illustration of MLP Training

- In the following slides, the results of training different configurations of a feedforward network are shown.
- The different configurations involve different activation functions and different numbers of hidden layer neurons.
- You will notice that as the number of hidden neurons is increased, the boundary tends to become more complex exhibiting overfitting behavior.

[1]:
```
import numpy as np
from sklearn.neural_network import MLPClassifier
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
```

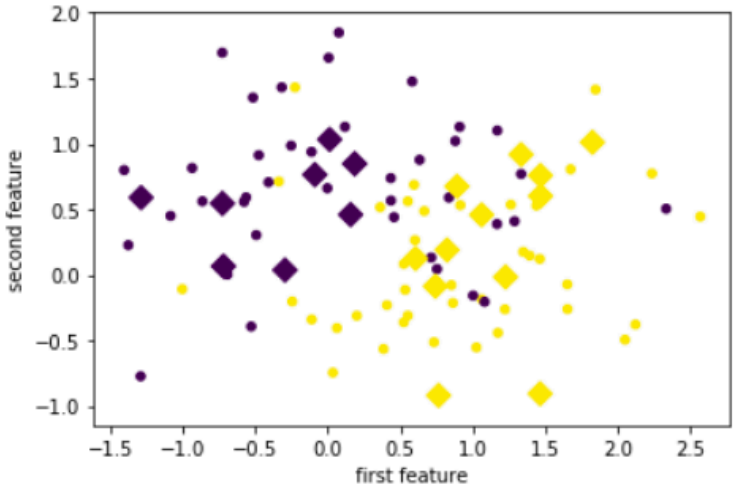
[2]:
```
X, y = make_moons(n_samples=100, noise=0.50, random_state=3)
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=40)
plt.xlabel("Feature 0")
plt.ylabel("Feature 1")
```

[2]:
```
Text(0, 0.5, 'Feature 1')
```



2-D synthetic data generation and dividing it into training and test sets. Specifying a random_state ensures reproducibility of the results

[4]:
```
mlp = MLPClassifier(hidden_layer_sizes =5,activation = 'logistic',learning_rate_init=0.05,
                    batch_size=10,solver='lbfgs', random_state=0, max_iter=500).fit(X_train, y_train)
result = mlp.predict(X_test)
marker = np.append(y_train,result, axis=0)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20)
plt.scatter(X_test[:, 0], X_test[:, 1], c=result,marker = "D", s=80)
plt.xlabel("first feature")
plt.ylabel("second feature")
plt.show()
```



Logistic activation is same as the sigmoidal activation. Solver refers to a particular gradient search technique

[5]:
```
print("Training_accuracy: {:.2f}".format(mlp.score(X_train, y_train)))
y_pred = mlp.predict(X_test)
print("Test_accuracy: {:.2f}".format(mlp.score(X_test, y_test)))
y_pred = mlp.predict(X_test)
```

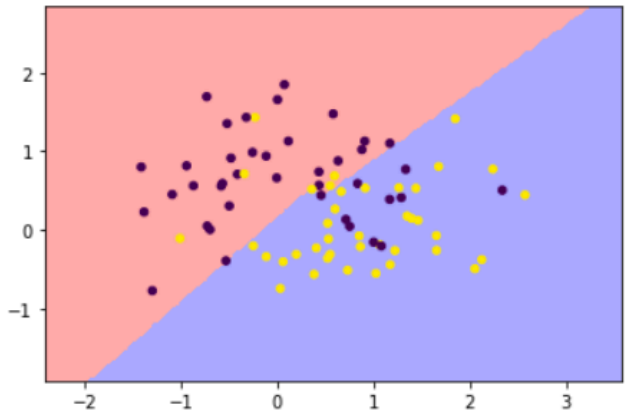
```
Training_accuracy: 0.81
Test_accuracy: 0.60
```

[5]:
```
print("Training_accuracy: {:.2f}".format(mlp.score(X_train, y_train)))
y_pred = mlp.predict(X_test)
print("Test_accuracy: {:.2f}".format(mlp.score(X_test, y_test)))
y_pred = mlp.predict(X_test)
```

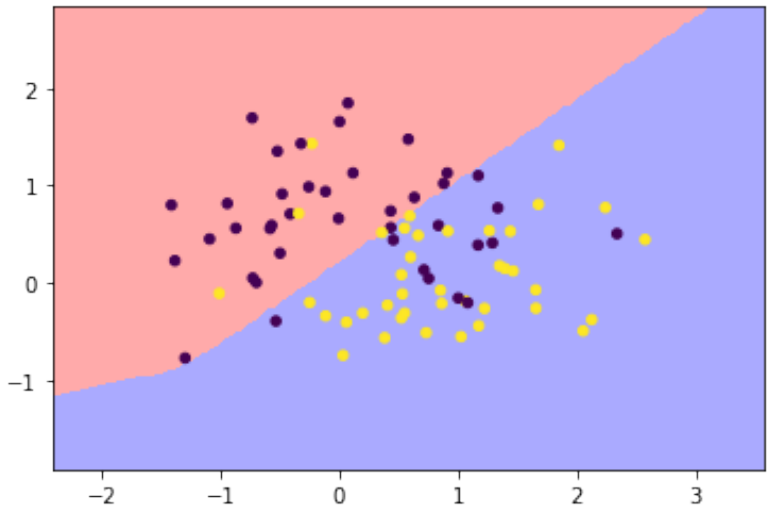
```
Training_accuracy: 0.81
Test_accuracy: 0.60
```

[6]:
```
# Plot the decision boundary
from matplotlib.colors import ListedColormap
cmap_light = ListedColormap(['#FFAAAA', '#AAFFAA', '#AAAAFF'])
# Plot the decision boundary. For that, we will assign a color to each
    # point in the mesh [x_min, x_max]x[y_min, y_max].
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.02),
np.arange(y_min, y_max, 0.02))
Z = mlp.predict(np.c_[xx.ravel(), yy.ravel()])
```

[7]:
```
Z = Z.reshape(xx.shape)
plt.pcolormesh(xx, yy, Z, cmap=cmap_light)
plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, s=20)
plt.show()
```



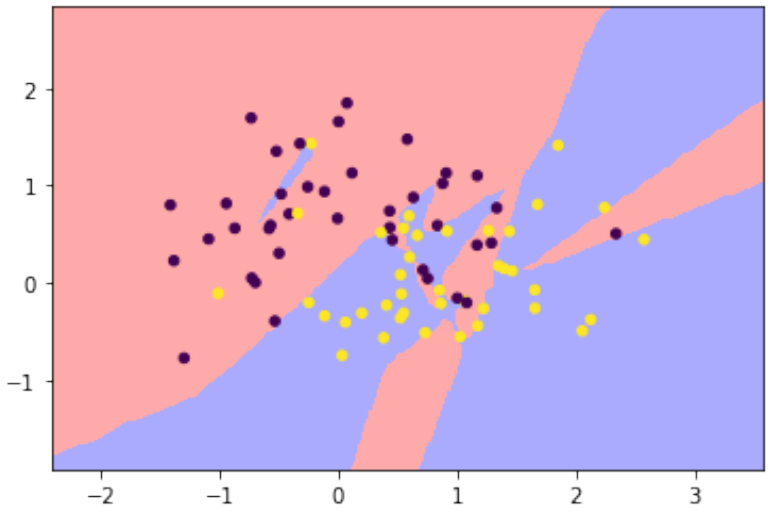
MLP Training Example



Hidden layer size = 2
Learning rate = 0.05
Activation Function = Logistic
Training Accuracy = 80%
Test accuracy = 60%

Not much different from the result with 5 hidden neurons

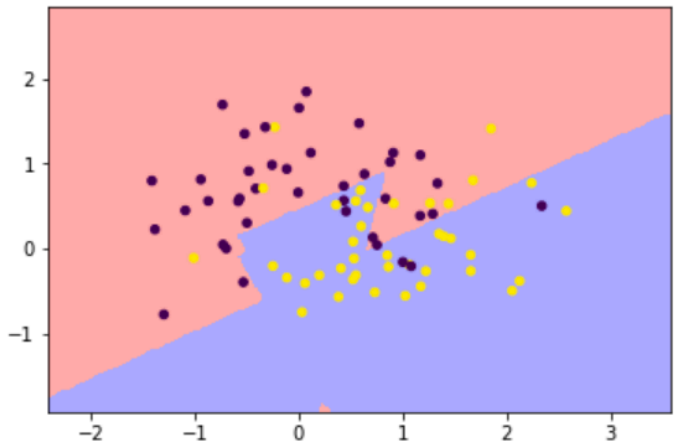
MLP Training Example



Hidden layer size = 10
Learning rate = 0.05
Activation Function = Logistic
Training Accuracy = 95%
Test accuracy = 45%

Note the training accuracy is very high but test accuracy has fallen. The decision boundary is way too complex showing overfitting.

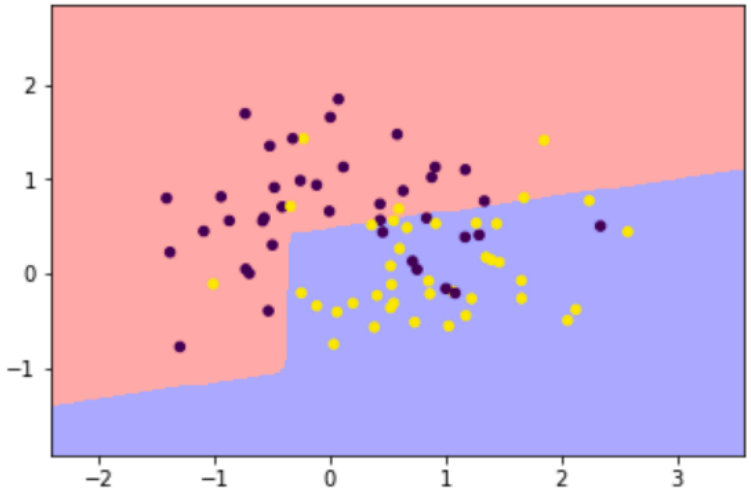
MLP Training Example



Hidden layer size = 5
Learning rate = 0.05
Activation Function = Tanh
Training Accuracy = 84%
Test accuracy = 75%

Better performance compared to the sigmoidal activation

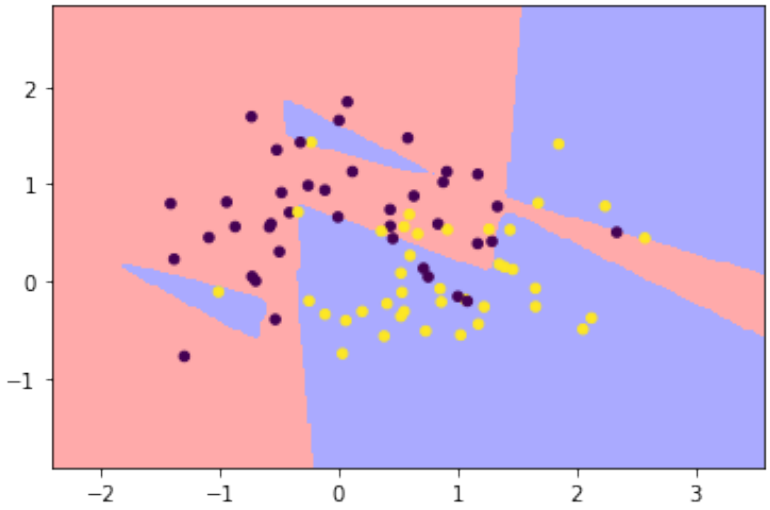
MLP Training Example



Hidden layer size = 2
Learning rate = 0.05
Activation Function = Tanh
Training Accuracy = 81%
Test accuracy = 70%

Good result, no hint of overfitting

MLP Training Example



Hidden layer size = 10
Learning rate = 0.05
Activation Function = Tanh
Training Accuracy = 85%
Test accuracy = 80%

Best accuracies so far but slight overfitting as evident from the boundary

Effect of Normalization

```
features, target = load_wine(return_X_y=True)
X_train, X_test, y_train, y_test = train_test_split(features, target,
                                                    test_size=0.30,
                                                    random_state=15)
```

```
mlp = MLPClassifier(hidden_layer_sizes =75,activation = 'tanh',learning_rate_init=0.01,batch_size=10,solver='lbfgs', random_state=0).fit(X_train, y_train)
result = mlp.predict(X_test)
```

```
print("Training_accuracy: {:.2f}".format(mlp.score(X_train, y_train)))
y_pred = mlp.predict(X_test)
print("Test_accuracy: {:.2f}".format(mlp.score(X_test, y_test)))
y_pred = mlp.predict(X_test)
```

```
Training_accuracy: 0.92
Test_accuracy: 0.81
```

When features have vastly different scales, normalization becomes important.

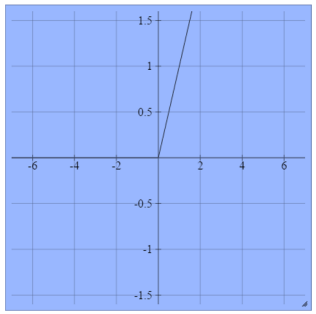
```
scalar = StandardScaler()
scalar.fit(X_train)
X_train_scaled=scalar.transform(X_train)
X_test_scaled = scalar.transform(X_test)
```

```
mlp = MLPClassifier(hidden_layer_sizes =75,activation = 'tanh',learning_rate_init=0.01,batch_size=10,solver='lbfgs', random_state=0).fit(X_train_scaled, y_train)
result = mlp.predict(X_test_scaled)
```

```
print("Training_accuracy: {:.2f}".format(mlp.score(X_train_scaled, y_train)))
y_pred = mlp.predict(X_test_scaled)
print("Test_accuracy: {:.2f}".format(mlp.score(X_test_scaled, y_test)))
y_pred = mlp.predict(X_test_scaled)
```

```
Training_accuracy: 1.00
Test_accuracy: 0.98
```

Currently Favored Activation Function

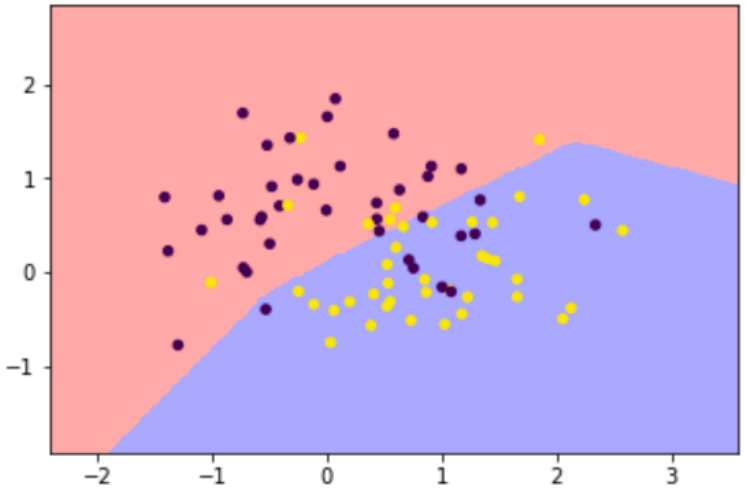


Rectified Linear Unit (ReLU) Function

Output = net if *net* > 0
= 0 if *net* < 0
Or Output = max(0, net)

This function has become popular for neurons. Empirical results have shown that training convergence is 5-6 times faster using this function compared to tanh function. Computationally it is simpler. However, ReLU usage can cause part of the network to result in dead units because linear part of the function can drive some units to operate in the left half part of ReLU function. Thus, it requires greater care in choosing the learning rate.

MLP Training Example with Relu



Hidden layer size = 5
Learning rate = 0.05
Activation Function = Relu
Training Accuracy = 79%
Test accuracy = 70%

Cross Entropy Criterion

Used for multi-class problems. Let C stand for a random variable representing class label of a training example. Let $p(c_i)$ stand for the probability of label being c_i. Then the entropy of the training set is

$$H(C) = -\sum_i p(c_i) \log p(c_i)$$

Let's say our training data has four labels: cat, dog, horse, sheep. Let the mix of labels in our training data be cat 40%, dog 10%, horse 25%, sheep 25%. Then

Entropy of our training set $H = -(0.4 \log 0.4 + 0.1\log 0.1 + 0.25\log 0.25 + 0.25\log 0.25 = 1.86$

Entropy is a measure of uncertainty. What will happen if all labels have equal representation in the mix?

Cross Entropy Criterion

Now consider two neural networks, Net1 and Net2.
Net1 produces the following classification result on our training data:
Cat 25%, dog 25%, horse 25%, and sheep 25%
Net2 yields the following result:
Cat 40%, dog 10%, horse 10%, sheep 40%

How do we say whether Net1 is better or Net 2 is better?
By cross entropy

$$H(C, \hat{C}) = -\sum_i p(c_i) \log p(\hat{c}_i)$$

$\hat{C}$ stands for label distribution in the classifier output

H(C, Net1) = 2.0 *H(C, Net2)* = 2.02

Net1 is slightly better because cross entropy in this case, 2.0, is closer to the entropy of the training set, 1.86

Cross entropy is a measure to determine how similar two probability distributions are. It is not a symmetric function. The loss function surface with this criterion is considered better for gradient search.

Summary of Neural Classifiers

- Able to model complex relationships
- Choice of parameters such as the number of hidden layer neurons, learning rate, normalization etc. play an important role
- The current state of neural classifiers incorporates 100 or more layers in what are known as deep learning neural networks.