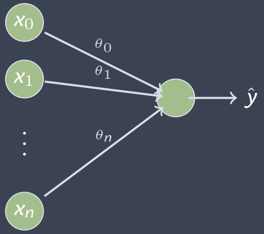
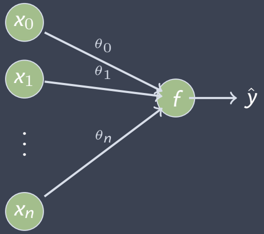
» Neural Networks

- Linear model: $\hat{y} = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots$
- Draw this schematically as:



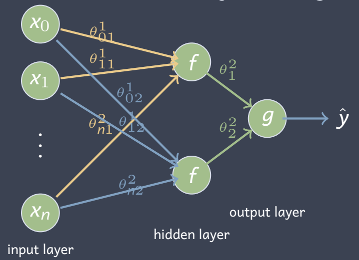
- A small generalisation: $\hat{y} = f(\theta^T x)$ where f is some function e.g. *sign*



NB:We first take the weighted sum of the inputs x_1, x_2 etc and then apply function f to result.

» Multi-Layer Perceptron (MLP)

- To get an MLP we add an extra "layer". E.g.



$$z_1 = f(\theta^1_{01}x_0 + \theta^1_{11}x_1 + \cdots + \theta^1_{n1}x_n)$$

$$z_2 = f(\theta^1_{02}x_0 + \theta^1_{12}x_1 + \cdots + \theta^1_{n2}x_n)$$

$$\hat{y} = g(\theta^2_1 z_1 + \theta^2_2 z_2)$$

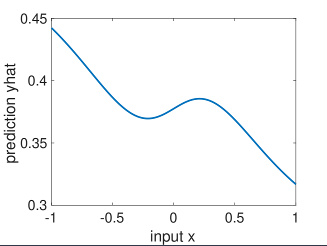
- MLP is a three layer network: (i) an *input layer*, (ii) a *hidden layer*, (iii) an *output layer*
- Not restricted to just two nodes in hidden layer, can have as many as we like.
- The parameters θ^1_{01} etc are called *weights*. It quickly gets messy indexing all the weights, often they're omitted from these schematics
- The function f is called the *activation* function, g is the output

» Multi-Layer Perceptron (MLP)

Example

* One input, two nodes in hidden layer, activation function is sigmoid $f(x) = g(x) = \frac{e^x}{1+e^x}$.

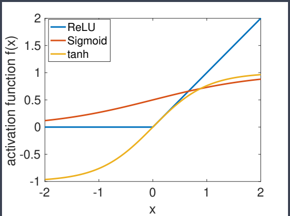
$$z_1 = f(5x), z_2 = f(2x), \hat{y} = f(z_1 - 2z_2) = f\left(f(5x) - 2f(2x)\right)$$



* By varying the number of hidden nodes and the weights the MLP can generate a wide range of functions mapping input x to output $\hat{y}$.

» Choices of Activation & Output Function

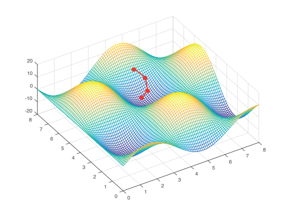
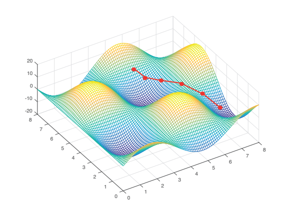
- ReLU (Rectified Linear Unit) $f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$
  - Popular in *hidden layer*. Quick to compute, observed to work pretty well.
  - But can lead to “dead” neurons where output is always zero $\rightarrow$ leaky ReLU
- Sigmoid $g(x) = \frac{e^x}{1+e^x}$
  - Sigmoid used in *output layer* when output is a probability (so between 0 and 1). For classification problems predict $+1$ when $\frac{e^x}{1+e^x} > 0.5$, -1 when $\frac{e^x}{1+e^x} < 0.5$
- tanh $g(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$
  - Used to be common for hidden layers
  - An output layer alternative for classification tasks



» Cost Function & Regularisation

Cost function:

* Typically use logistic loss function for classification problems
* And square loss $\frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)})-y^{(i)})^2$ for regression problems
* In both cases the cost function is non-convex in the neural net weights/parameters → non-convexity plus large number of weights/parameters means training a neural net is often slow/hard

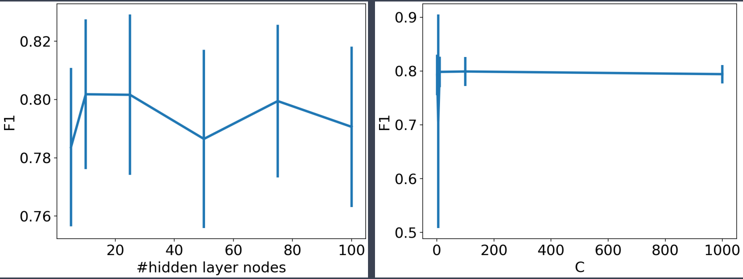



Regularisation

* Typically L_2 penalty i.e. the sum of the squared weights/parameters
* Can also use drop outs → randomly setting the outputs of a fraction of nodes in hidden layer to zero at each gradient descent step. But we don't go into this here.

» Movie Review Example

Apply MLP to movie review example. Use cross-validation to select (i) #hidden nodes, (ii) L_2 penalty weight C.



- Performance not too sensitive to #hidden nodes, so choose a small number e.g. 5
  - Ups and downs in plot likely due to failure to find global minimum of cost function (the wiggles change from run to run as initial condition for optimisation changes)
- Performance insensitive to penalty weight C, so long as $C \geq 5$ or thereabouts

» Movie Review Example

MLP settings:

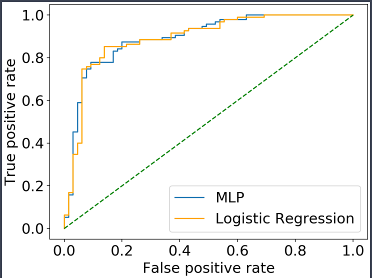
* hidden layer has 5 nodes, penalty weight $C = 5$, ReLU *activation* function

Confusion matrix:

	predicted positive	predicted negative
true positive	60	5
true negative	28	67

with $m = 160$ data points (20% test split from full data set of 800 points).

ROC Curve:



» Python Code For MLP Movie Example

```
import matplotlib.pyplot as plt
plt.rc('font', size=18);plt.rcParams['figure.constrained_layout.use'] = True

crossval=False
if crossval:
    mean_error=[]; std_error=[]
    hidden_layer_range = [5,10,25,50,75,100]
    for n in hidden_layer_range:
        print("hidden layer size %d\n"%n)
        from sklearn.neural_network import MLPClassifier
        model = MLPClassifier(hidden_layer_sizes=(n), max_iter=300)
        from sklearn.model_selection import cross_val_score
        scores = cross_val_score(model, X, y, cv=5, scoring='f1')
        mean_error.append(np.array(scores).mean())
        std_error.append(np.array(scores).std())


    plt.errorbar(hidden_layer_range,mean_error,yerr=std_error,linewidth=3)
    plt.xlabel('#hidden layer nodes'); plt.ylabel('F1')
    plt.show()

mean_error=[]; std_error=[]
C_range = [1,5,10,100,1000]
for Ci in C_range:
    print("C %d\n"%Ci)
    from sklearn.neural_network import MLPClassifier
    model = MLPClassifier(hidden_layer_sizes=(5), alpha = 1.0/Ci)
    from sklearn.model_selection import cross_val_score
    scores = cross_val_score(model, X, y, cv=5, scoring='f1')
    mean_error.append(np.array(scores).mean())
    std_error.append(np.array(scores).std())

plt.errorbar(C_range,mean_error,yerr=std_error,linewidth=3)
plt.xlabel('C'); plt.ylabel('F1')
plt.show()
```

» Python Code For MLP Movie Example (cont)

```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(hidden_layer_sizes=(5), alpha=1.0/5).fit(Xtrain, ytrain)
preds = model.predict(Xtest)
from sklearn.metrics import confusion_matrix
print(confusion_matrix(ytest, preds))
from sklearn.dummy import DummyClassifier
dummy = DummyClassifier(strategy="most_frequent").fit(Xtrain, ytrain)
ydummy = dummy.predict(Xtest)
print(confusion_matrix(ytest, ydummy))

from sklearn.metrics import roc_curve
preds = model.predict_proba(Xtest)
print(model.classes_)
fpr, tpr, _ = roc_curve(ytest,preds[:,1])
plt.plot(fpr,tpr)

from sklearn.linear_model import LogisticRegression
model = LogisticRegression(C=10000).fit(Xtrain, ytrain)
fpr, tpr, _ = roc_curve(ytest,model.decision_function(Xtest))
plt.plot(fpr,tpr,color='orange')
plt.legend(['MLP','Logistic Regression'])
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.plot([0, 1], [0, 1], color='green',linestyle='--')
plt.show()
```

» Training Neural Networks: Stochastic Gradient Descent [Optional]

Recall gradient descent to minimise cost function $J(\theta)$:

* Start with some parameter vector θ of size n
* Repeat:

  for j=0 to n $\{\delta_j := -\alpha \frac{\partial J}{\partial \theta_j}(\theta)\}$

  for j=0 to n $\{\theta_j := \theta_j + \delta_j\}$

Cost function is a sum over prediction error at each training point, e.g. $J(\theta) = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$. Rewrite as

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} l_i(\theta)$$

where e.g. $l_i(\theta) = (h_\theta(x^{(i)}) - y^{(i)})^2$. Then

$$\frac{\partial J}{\partial \theta_j}(\theta) = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial l_i}{\partial \theta_j}(\theta)$$

When m is large then calculating this sum is slow.

» Training Neural Networks: Stochastic Gradient Descent [Optional]

Stochastic gradient descent (SGD) to minimise cost function $J(\theta)$:

* Start with some parameter vector θ of size n
* Repeat:
  Pick training data point i,
  e.g. randomly or by cycling through all data points.
  for j=0 to n $\{\delta_j := -\alpha \frac{\partial l_i}{\partial \theta_j}(\theta)\}$
  for j=0 to n $\{\theta_j := \theta_j + \delta_j\}$

At each update we use just one point from the training data, so avoid sum over all points …

* Each update is fast to compute
* But need more iterations to minimise $J(\theta)$.

Now add mini-batches and parallelise …

» Training Neural Networks: Stochastic Gradient Descent [Optional]

Stochastic gradient descent with mini-batches of size q:

* Start with some parameter vector θ of size n
* Repeat:
*   for $i = 1$ to q:

        Pick training data point i,

            e.g. randomly or by cycling through all data points.

        for j=0 to n $\{\delta_j := -\alpha \frac{\partial l_i}{\partial \theta_j}(\theta)\}$

        for j=0 to n $\{\theta_j := \theta_j + \delta_j\}$

If have q processors then each can run for-loop in parallel and takes same time as one SGD update. Now:

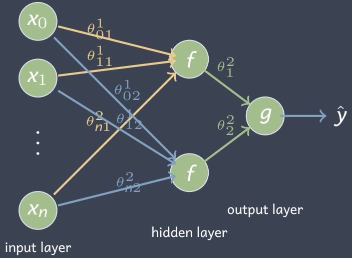
* Each update is fast to compute
* Reduce number of iterations by factor of q compared to vanilla SGD.

Because of communication and synchronization costs between processors often make mini-batch size larger than number of processors and at each round calc a few updates separately on each processor, not just one.

» Training Neural Networks: Stochastic Gradient Descent [Optional]

Calculating gradient $\frac{\partial l_i}{\partial \theta_j}$ for neural nets

- Calculate output $\hat{y}$ of neural network → *forward propagation* (the sorts of neural nets we're considering are sometimes called *feedforward networks*

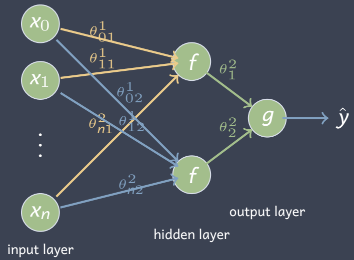


Apply training data input $x^{(i)}$ to hidden layer and calculate outputs of hidden layer, then apply outputs from hidden layer to output layer and calculate output $\hat{y}$.

» Training Neural Networks: Stochastic Gradient Descent [Optional]

Calculating gradient $\frac{\partial l_i}{\partial \theta_j}$ for neural nets

* To calculate derivatives $\frac{\partial l_i}{\partial \theta_j}$ for all weights/parameters j efficiently use *backpropagation*.
    * Calculate difference between neural network output $\hat{y}$ and training data output $y^{(i)}$. Adjust weights θ^2_1, θ^2_2 connecting hidden layer and output layer to reduce this error.
    * Now calculate how hidden layer outputs should be adjusted to reduce error. Adjust weights θ^1_{01} etc connecting input layer to hidden layer accordingly.



* Backpropagation = process for calculating $\frac{\partial l_i}{\partial \theta_j}$ for all weights θ_j. But often backpropagation is also used as shorthand for the whole process of stochastic gradient descent.

» Summary

* A neural net is just another model i.e. a function mapping from input to prediction. Biological analogies are generally spurious and just confusing.
* Hard to interpret what the weights mean → its a *black box* model
* Can be tricky/slow to train → cost function is non-convex in weights/parameters, plus often many weights/parameters that need to be learned
* Popular in 1990s, then less so. Resurgence of interest from around 2010 due to use in image processing → mainly relates to their use for feature engineering and especially the use of *convolutional layers*.