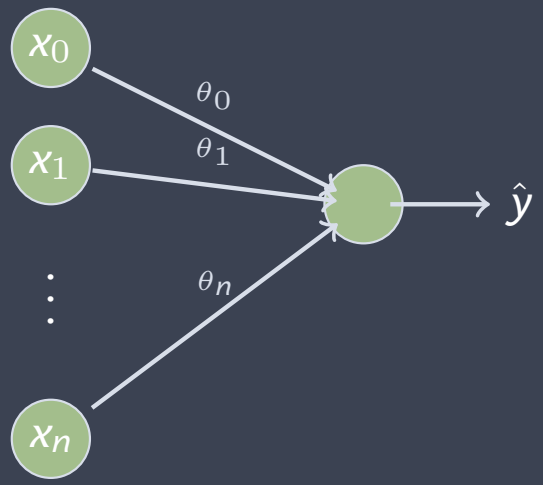
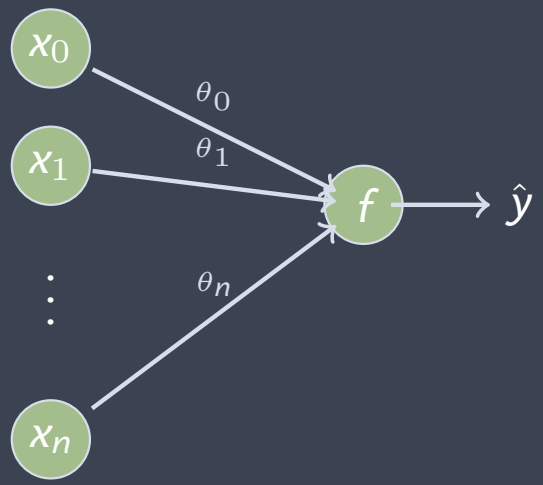
» Neural Networks

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



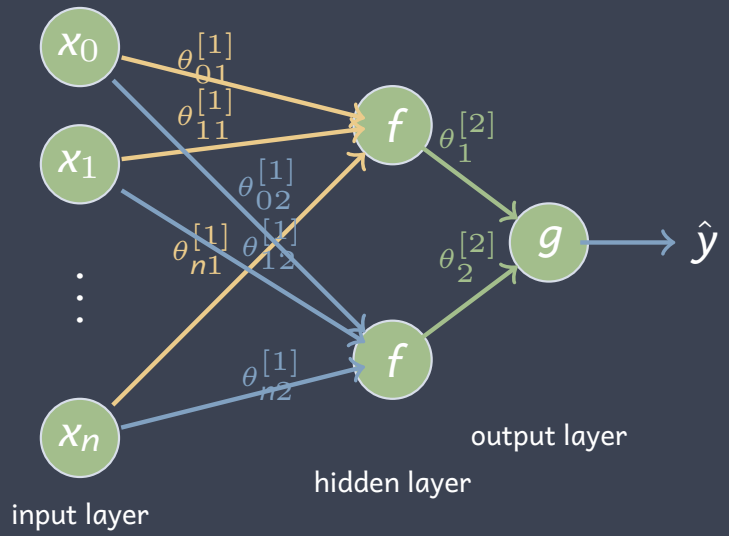
* A small generalisation: $\hat{y} = f(\theta^T x)$ where f is some function e.g. *sign*



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.



$$z_1 = f(\theta_{01}^{[1]} x_0 + \theta_{11}^{[1]} x_1 + \cdots + \theta_{n1}^{[1]} x_n)$$

$$z_2 = f(\theta_{02}^{[1]} x_0 + \theta_{12}^{[1]} x_1 + \cdots + \theta_{n2}^{[1]} x_n)$$

$$\hat{y} = g(\theta_1^{[2]} z_1 + \theta_2^{[2]} z_2)$$

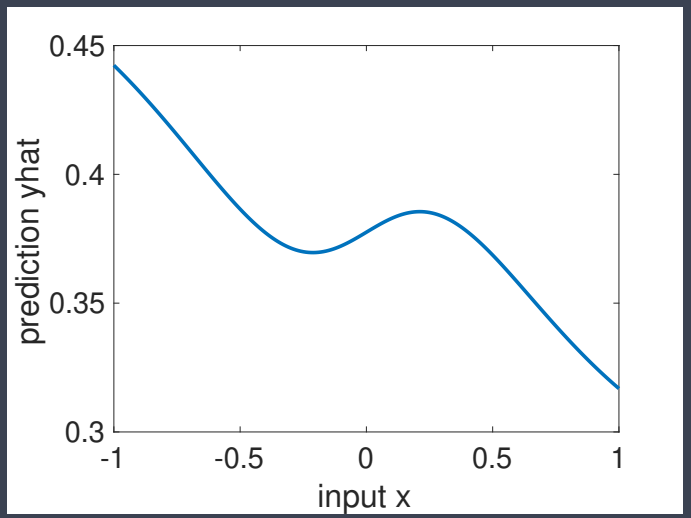
* MLP is a three layer network: (i) an *input layer*, (ii) a *hidden layer*, (iii) an *output layer*
* Not restricted to two nodes in hidden layer, can have many nodes.
* Not restricted to one output, can have many outputs
* The parameters $\theta_{01}^{[2]}$ etc are called *weights*. It quickly gets messy indexing all the weights, often they're omitted from these schematics
* The function f is called the *activation* function, g is the output function.

» Multi-Layer Perceptron (MLP)

Example

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

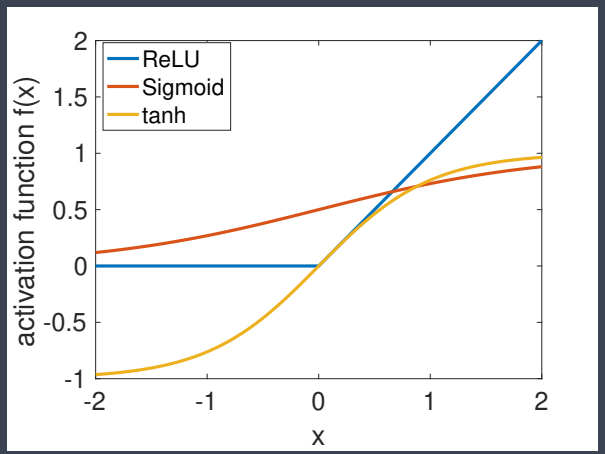
$$z_1 = f(5x), z_2 = f(2x), \hat{y} = f(z_1 - 2z_2) = f(f(5x) - 2f(2x))$$



* 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

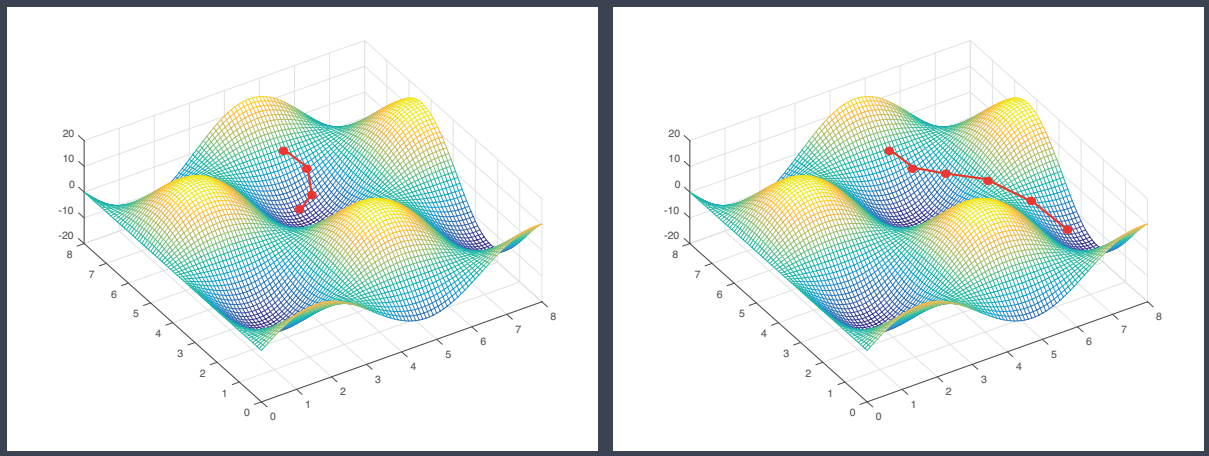
* ReLU (Rectified Linear Unit) $f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$
  * Popular in *hidden layer*. Quick to compute, observed to work pretty well.
  * But can lead to "dead" neurons where output is always zero $\rightarrow$ leaky ReLU
* Sigmoid $g(x) = \frac{e^x}{1+e^x}$
  * Sigmoid 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, not so much now.



» 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 $\rightarrow$ training a neural net can be tricky



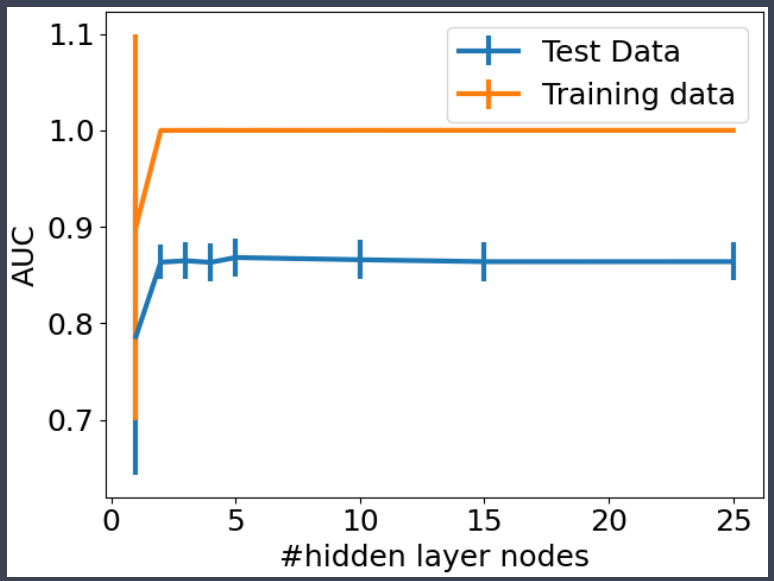
Regularisation

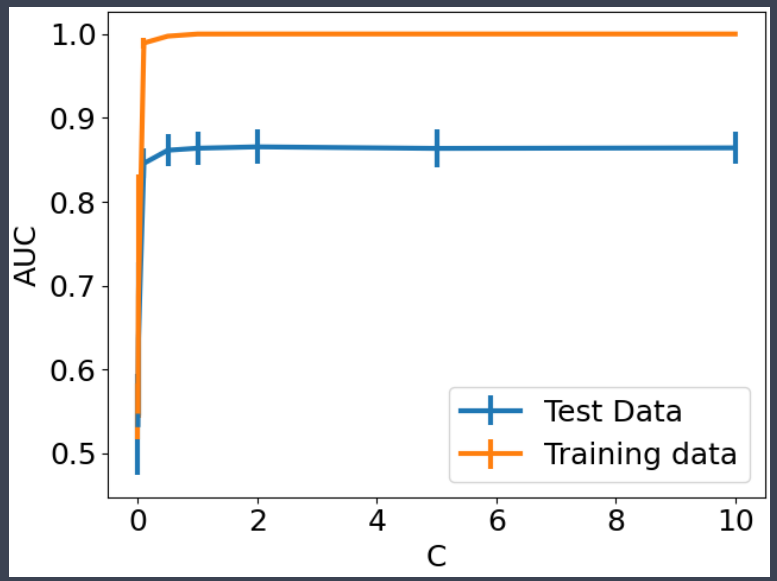
* L_2 penalty i.e. the sum of the squared weights/parameters
* Nowadays more common to use *dropout* regularisation
  * Set the outputs of randomly selected set of nodes in hidden layer to zero at each gradient descent step
  * Typically remove about 50% of nodes
  * This is similar[1] to a *weighted* L_2 penalty $\sum_{i=1}^{n} w_i\theta_i$

[1] papers.nips.cc/paper/4882-dropout-training-as-adaptive-regularization.pdf

» 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. 2
- Not much sign of overfitting, at least for this range of #hidden nodes ($C = 1$ in plot).
- Performance insensitive to penalty weight C, so long as $C \geq 0.5$ or thereabouts (#hidden nodes=2 in plot)

» Movie Review Example

MLP settings:

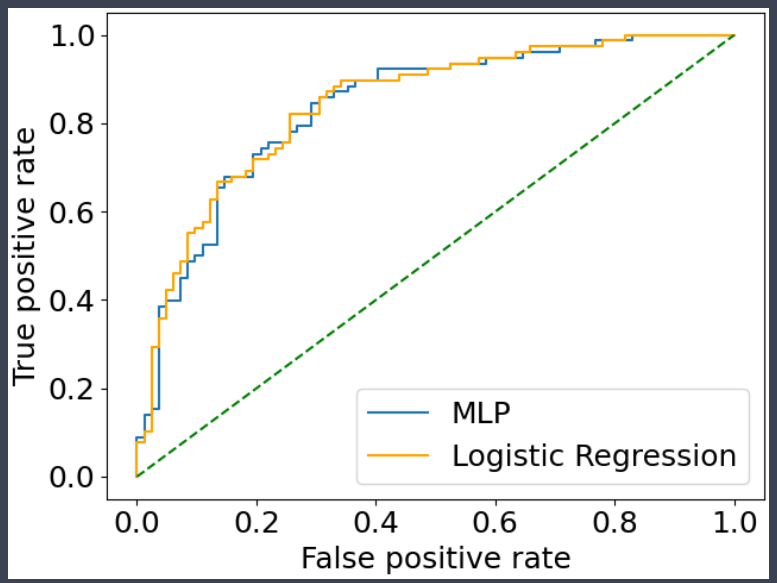
* hidden layer has 2 nodes, penalty weight $C = 1$, ReLU activation function

Confusion matrix:

true positive	62	20
true negative	19	59
	predicted positive	predicted negative

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

from sklearn.neural_network import MLPClassifier
from sklearn.model_selection import cross_validate
crossval=False
if crossval:
        mean_error=[]; std_error=[]; mean_error1=[]; std_error1=[]
        hidden_layer_range = [1,2,3,4,5,10,15,25]
        for n in hidden_layer_range:
                print("hidden layer size %d\n"%n)
                model = MLPClassifier(hidden_layer_sizes=(n), alpha=1, max_iter=500)
                scores = cross_validate(model, X, y, cv=5, return_train_score=True, scoring='roc_auc')
                mean_error.append(np.array(scores['test_score']).mean()); std_error.append(np.array(scores['test_score']).std())
                mean_error1.append(np.array(scores['train_score']).mean()); std_error1.append(np.array(scores['train_score']).
std())
        plt.errorbar(hidden_layer_range,mean_error,yerr=std_error,linewidth=3)
        plt.errorbar(hidden_layer_range,mean_error1,yerr=std_error1,linewidth=3)
        plt.xlabel('#hidden layer nodes'); plt.ylabel('AUC')
        plt.legend(['Test Data', 'Training data'])
        plt.show()

mean_error=[]; std_error=[]; mean_error1=[]; std_error1=[]
C_range = [0.001,0.01,0.1,0.5,1,2,5,10]
for C in C_range:
        print("C %d\n"%Ci)
        model = MLPClassifier(hidden_layer_sizes=(2), alpha = 1.0/C)
        scores = cross_validate(model, X, y, cv=5, return_train_score=True, scoring='roc_auc')
        mean_error.append(np.array(scores['test_score']).mean()); std_error.append(np.array(scores['test_score']).std())
        mean_error1.append(np.array(scores['train_score']).mean()); std_error1.append(np.array(scores['train_score']).std())
plt.errorbar(hidden_layer_range,mean_error,yerr=std_error,linewidth=3)
plt.errorbar(hidden_layer_range,mean_error1,yerr=std_error1,linewidth=3)
plt.xlabel('C'); plt.ylabel('AUC')
plt.legend(['Test Data', 'Training data'])
plt.show()
```

» Python Code For MLP Movie Example (cont)

```
from sklearn.neural_network import MLPClassifier
model = MLPClassifier(hidden_layer_sizes=(2), alpha=1.0).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

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

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

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

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 becomes slow to calculate.

» Training Neural Networks: Stochastic Gradient Descent

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

- Pseudo-code:
  - 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.
    $\theta_j := \theta_j - \alpha \frac{\partial l_i}{\partial \theta_j}(\theta_j)$ for j=0 to n
- At each update we use just one point from the training data, so avoid sum over all points i.e. SGD update is:

$$\theta_j \leftarrow \theta_j - \alpha \frac{\partial l_i}{\partial \theta_j}(\theta_j)$$

instead of gradient descent update:

$$\theta_j \leftarrow \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} \frac{\partial l_i}{\partial \theta_j}(\theta_j)$$

- Each SGD 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

Stochastic gradient descent with mini-batches of size q:

* SGD update with mini-batch size q:

$$\theta_j \leftarrow \theta_j - \alpha \frac{1}{q} \sum_{i \in I_q} \frac{\partial l_i}{\partial \theta_j}(\theta_j)$$

  where I_q is a set of q training data points.
* Choose I_q by shuffling training data and then cycling through it or by selecting q points randomly from training data
* Pseudo-code:
  * Training data with m points, mini-batch size q
  * Repeat:
  * shuffle training data
  * for $k = 1$ to m step q:
  * $\delta_j = 0$ for j=0 to n
  * for $i = 1$ to q: # k'th mini-batch

$$\delta_j := \delta_j + \frac{\partial l_{k+i}}{\partial \theta_j}(\theta_j) \text{ for } j\text{=0 to } n$$

$$\theta_j := \theta_j - \frac{\alpha}{q}\delta_j \text{ for } j\text{=0 to } n$$

» Training Neural Networks: Stochastic Gradient Descent

If have k processors and mini-batch size q:

- Divide q into k batches of size q/k.
- Parallelise the for $i = 1$ to q loop i.e. split into k for $i = 1$ to q/k loops and run each on one processor $\rightarrow$ q loop now runs k times faster.

How to choose mini-batch size q?

- Typical values of q are 32-256
- Computation time tends to increase when batch size q gets too small (can't exploit parallelism as well, communication and synchronization costs between processors are amortised by using larger q/k)
- Small batches provide a sort of regularisation. Using large batches is often observed to lead to over-fitting (poor predictions for new data).
  - This aspect remains poorly understood, best we have are heuristics[2]
- Note: choosing batch size $q = m$ the training data size then mini-batch SGD = gradient descent

[2] See "On Large-Batch Training For Deep Learning", ICLR 2017 https://openreview.net/pdf?id=H1oyRlYgg

» Some Terminology

When using SGD in sklearn and other packages you might see the following terms:

* *Epoch*.
    * SGD update with mini-batch size q:

$$\theta_j \leftarrow \theta_j - \alpha \frac{1}{q} \sum_{i \in I_q} \frac{\partial l_i}{\partial \theta_j}(\theta)$$

      where I_q is a set of q training data points.
    * If shuffle training data and then cycle through it, then one cycle = an epoch i.e. one iteration of repeat loop in our pseudo-code
    * After first epoch each training data point has been used once, in second epoch twice, and so on
    * Often train for a fairly small number of epochs, e.g. 1-25
* *Momentum*. With SGD the gradient updates are "noisy". So can average out this "noise" to try to find a good downhill direction.
* *Adam*. An approach for automatically choosing the step-size α plus using momentum. Currently the default in most packages, its ok to leave it that way for assignments in this module.

For those of you taking optimization module in semester 2 we'll go into these in a lot more detail.

» Some Terminology

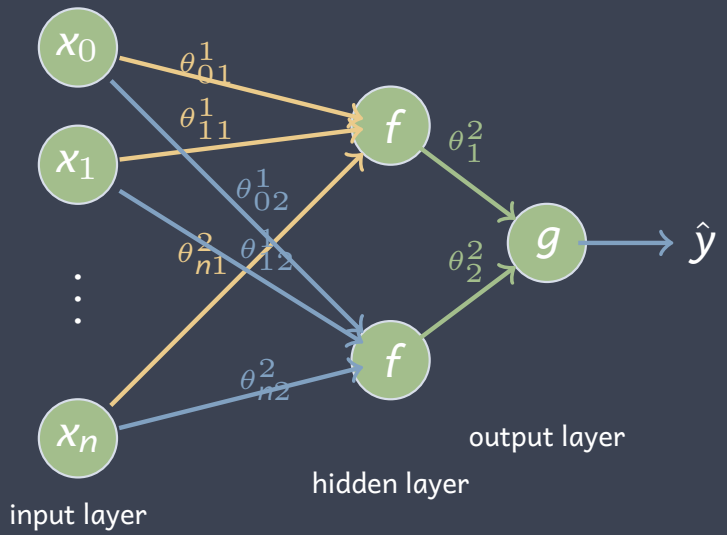
Early stopping:

* An old idea is to try to achieve regularisation by stopping SGD early i.e. before cost function as converged to its minimum → early stopping
    * Repeat:
    * Keep a hold-out test set from training data e.g. 10% of data
    * Run SGD for a while, e.g. 1 epoch, on remaining training data
    * Evaluate cost function on (i) held-out test data and (ii) on training data used for SGD
    * Stop when cost function of test data stops decreasing and/or when these two values start to diverge
* Often used with SGD in combination with a penalty or dropouts for regularisation

» Training Neural Networks: Stochastic Gradient Descent

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

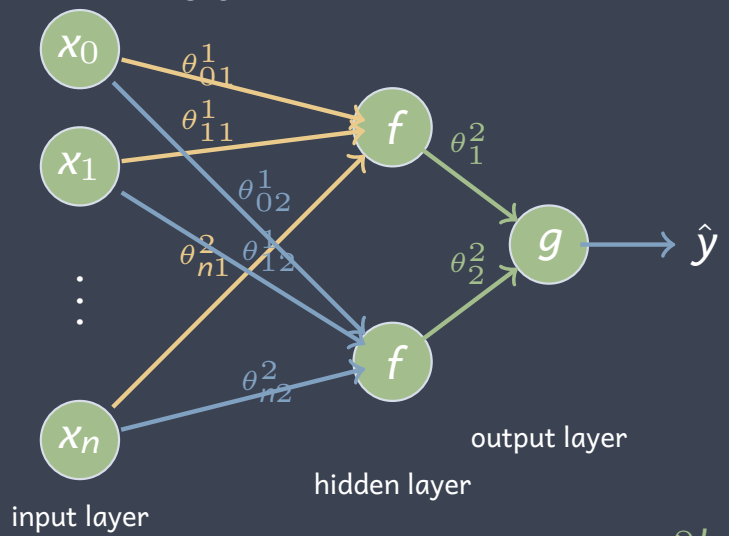
* Calculate output $\hat{y}$ of neural network $\rightarrow$ *forward propagation* (the sorts of neural nets we're considering are sometimes called *feedforward networks*



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

* 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_1^2, 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_{01}^1 etc connecting input layer to hidden layer accordingly.



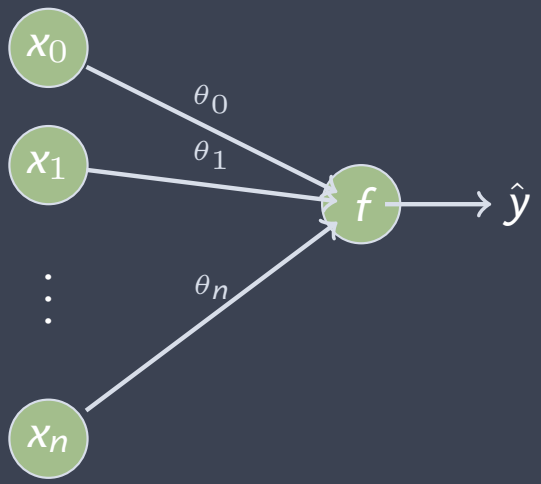
* Backpropagation = process for calculating $\frac{\partial l_i}{\partial \theta_j}$ for all weights θ_j. But often backpropagation is also used as shorthand for the whole process of stochastic gradient descent.
* Details of calc don't matter for us, all we need to know is that output is the $\frac{\partial l_i}{\partial \theta_j}$'s

» 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 hype.
- Hard to interpret what the weights mean $\rightarrow$ its a *black box* model
- Can be tricky/slow to train $\rightarrow$ 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 $\rightarrow$ mainly relates to their use for feature engineering and especially the use of *convolutional layers* and *transformer blocks*.

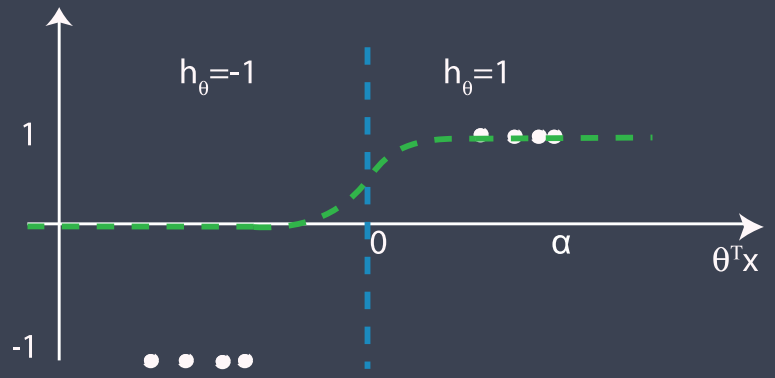
» SoftMax Layer: Expressing Logistic Regression In NN Terminology

* One layer of a neural net is $\hat{y} = f(\theta^T x)$:



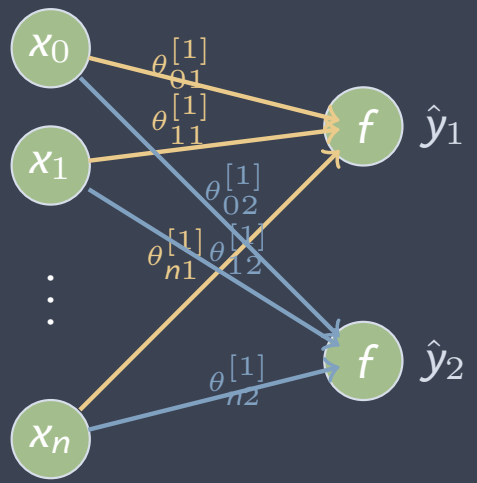
* Select $f(\cdot)$ to be sign function and we're back to logistic regression/SVM model
* Alternatively, recall that in logistic regression with two classes we map from $\theta^T x$ to a confidence value using:

$$Prob(y = 1) = \frac{e^{\theta^T x}}{e^{\theta^T x} + 1}, \; Prob(y = 0) = 1 - Prob(y = 1) = \frac{1}{e^{\theta^T x} + 1}$$



» SoftMax Layer: Expressing Logistic Regression In NN Terminology

* Define two outputs $\hat{y}_1 = f(\theta^T x) = \frac{e^{\theta^T x}}{e^{\theta^T x}+1}$, $\hat{y}_2 = \frac{1}{e^{\theta^T x}+1}$. If $\hat{y}_1 > \hat{y}_2$ predict class 1, else predict class 2.
* What about if have $K > 2$ classes?
  * Train a separate linear model for each $k = 1, \ldots, K$, so have $z_k = \theta_k^T x$ where θ_k is vector of parameters for class k.
  * Probabilities have to sum to 1, so then normalise: $\hat{y}_k = Prob(y = k) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$
  * Predict class based on which output $\hat{y}_k$ is largest
  * This is called *softmax* function. Can draw schematically as:



  * Called a *softmax* layer → its identical to a multi-class logistic regression model, despite NN terminology