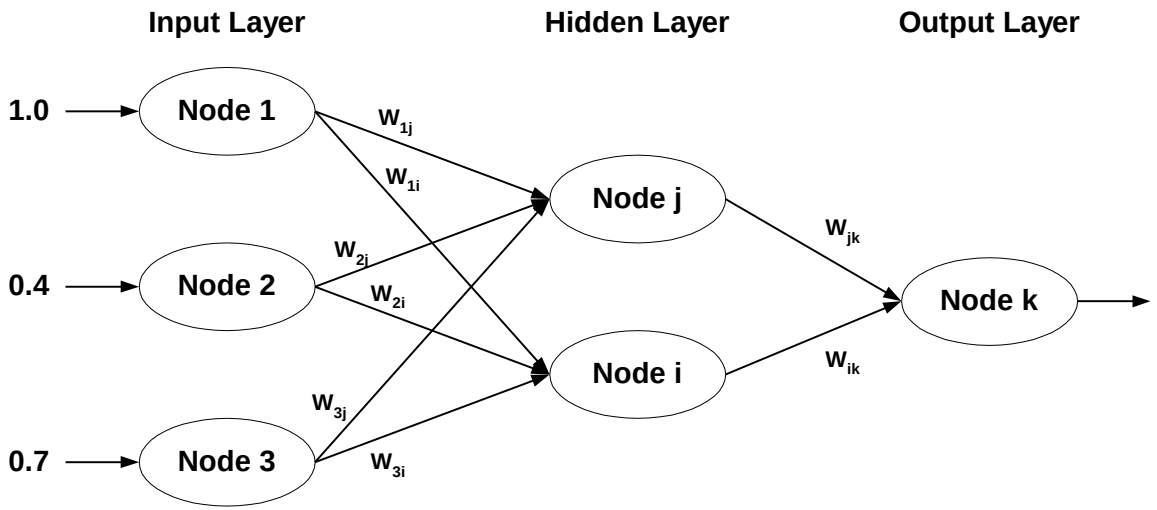
Multi-Layer Perceptrons

Multilayer Neural Networks

- The single-layer perceptron classifiers discussed previously can only deal with linearly separable sets of patterns.

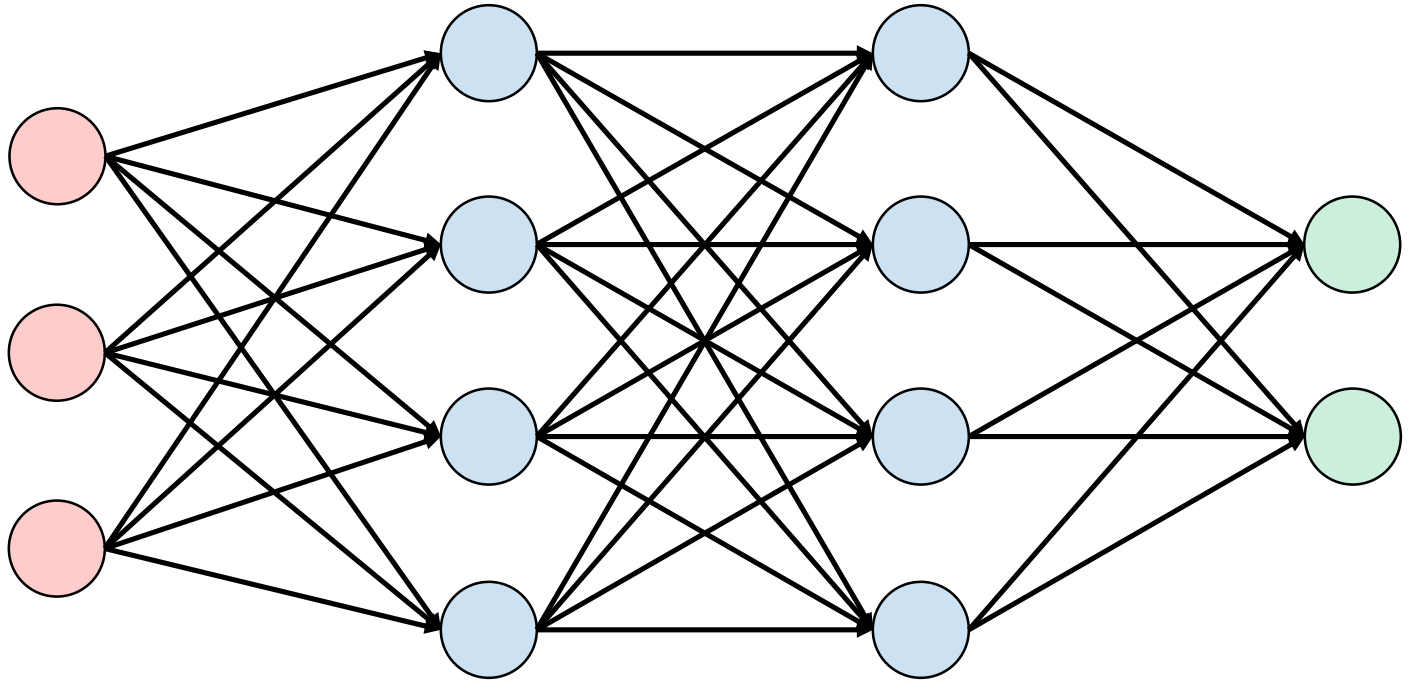
- How to solve non-linear problems using neural network?
  1) Add hidden layers in perceptrons -> multilayer networks
  2) Use non-linear activation functions(e.g. sigmoid)

- The multilayer networks are the most widespread neural network architecture



Multilayer Neural Networks

- Network with two layers of four hidden units each and one layer of two output units.



- A neural network with $D-1$ layers of hidden units and 1 output layer is termed a D-layer neural network.

Fully Connected Layers

- An important property of a layer is how its units are connected to the previous layer.
  - Obviously, this is not applicable to the input layer, that has no previous layer.
- The most simple type (and most expensive computationally) is a fully connected layer.
  - Every unit in this layer is connected to every unit in the previous layer.
- At first, we will work with fully connected layers.
  - This will be the type that you will implement first.
- Then we will talk about other types of layers.
  - Convolutional, max-pooling, LSTM, etc

Predicting outputs and learning parameters

- We can train and make predictions from neural networks using two algorithms: **forward** and **backward propagation**.

- With forward propagation we can compute the output from a set of inputs.

- With backward propagation, we can evaluate how changes in parameters changes the objective (cost) function.

- Using both forward and backgward propagation, we can learn the model parameters.

How do we learn the model parameters?

- So far we have discussed forward propagation for making predictions of outputs $O \in \mathbb{R}^K$ from an input set of features $X \in \mathbb{R}^p$.

- But how do we learn the underlying model parameters to be able to make this predictions?

- That is, given a set of observations (x_i, y_i), $i = 1,2, \ldots, N$, where $x_i \in \mathbb{R}^p$ and $y_i \in \mathbb{R}^K$, how do we train a neural network model for making quantitative predictions (regression) or for performing classification?

- To train a neural network we need to define a cost function $J(w)$ of the set of model parameters w, and then identify the set of parameters θ that minimize a cost function $J(w)$.

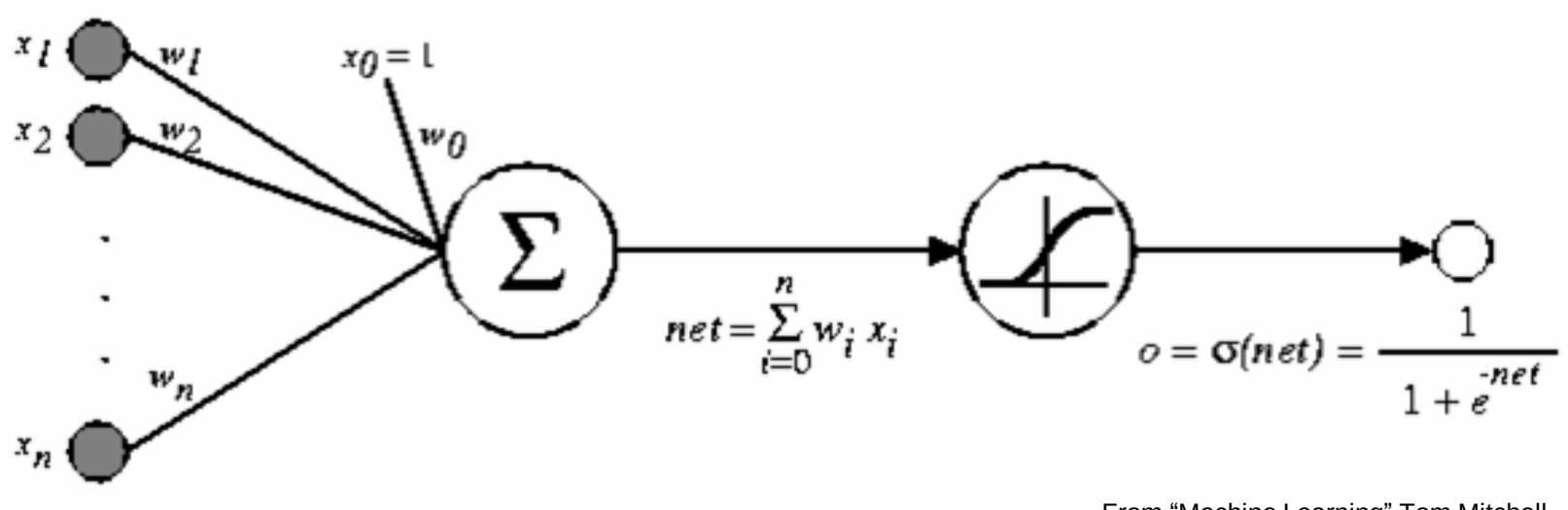
Multilayer Networks & Backpropagation Algorithm

- The training algorithm for multilayer neural network is called the backpropagation algorithm *(D. Rumelhart, G. Hinton, R. Williams, 1986)* that is capable of learning a rich variety of **nonlinear** decision surfaces.

- Backpropagation Algorithm(BP) train such multilayer networks using a gradient descent method

- A different activation function (sigmoid) is used in Backpropagation.
  - Perceptron training rule: step function
    - Not differentiable -> can't use gradient descent
  - Delta rule: linear function
    - Continuous, differentiable, still produce only linear function
  - Backpropagation: sigmoid function
    - Continuous, differentiable, non-linear function

The sigmoid activation function



From “Machine Learning” Tom Mitchell

- Use sigmoid function instead of step function or linear function
  - advantages: continuous, differentiable, non-linear

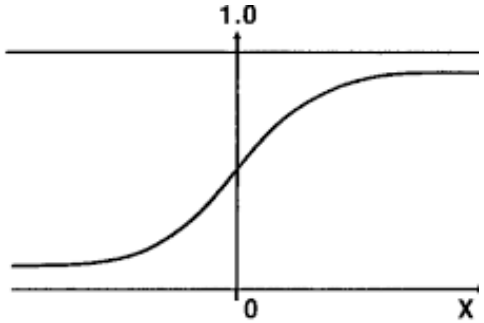
- $\sigma(x)$ is the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

The sigmoid activation function

- The sigmoid function σ(x) is also called the **logistic function**.

$$\sigma(x) = \frac{1}{1 + e^{-kx}}$$



- Output ranges between 0 and 1, increases monotonically with its input

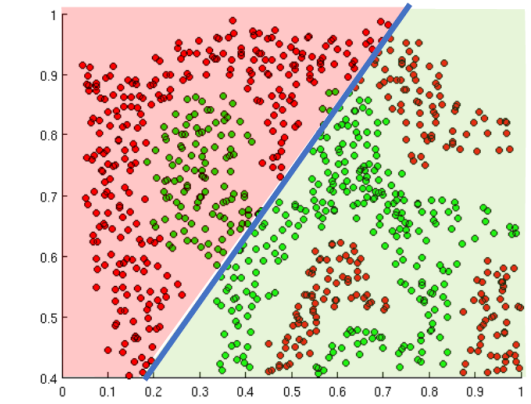
- Interesting property:

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1\text{-}\sigma(x))$$

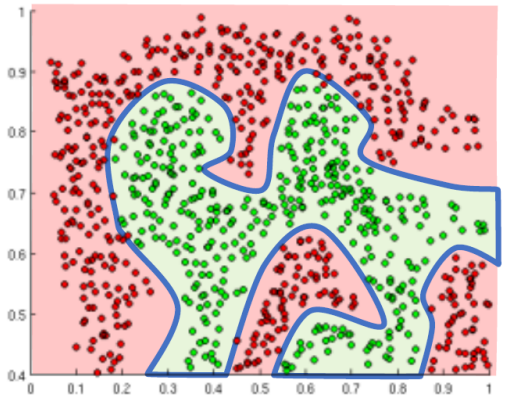
- We can derive gradient decent rules to train *Multilayer networks* of sigmoid units ⇒ Backpropagation

Importance of activation function

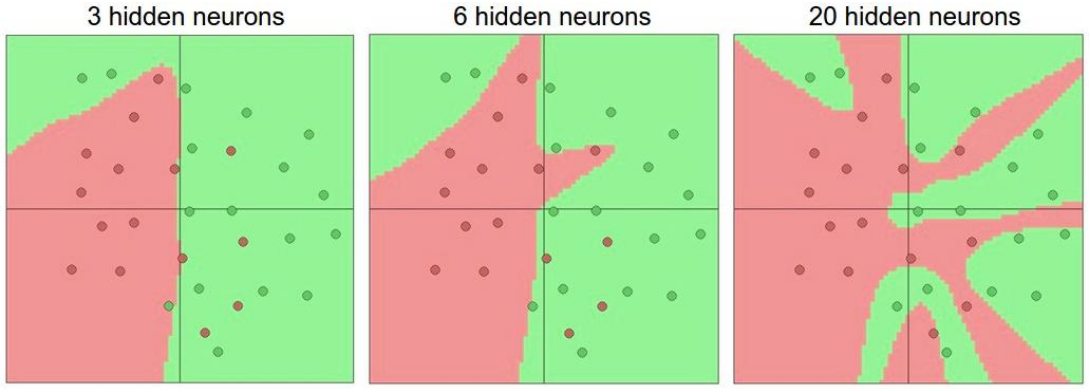
- Non-linearities needed to learn complex (non-linear) representations of data, otherwise the NN would be just a linear function
- More layers and neurons can approximate more complex functions



Linear Activation functions produce linear decisions no matter the network size



Non-linearities allow us to approximate arbitrarily complex functions



http://cs231n.stanford.edu/

Defining the cost function

- For regression type problems with K-dimensional response, we might choose the squared loss cost function (*e.g.*, least squares regression)

$$J(w) = \sum_{i=1}^{N} \sum_{k=1}^{K} (y_{ik} - o_{ik})^2$$

where y_{ik} is the value of the kth response for observation i and o_{ik} is the value of the kth predicted response for observation i.

- For classification problems with K classes, we might use the categorical cross-entropy cost function (*e.g.*, logistic regression)

$$J(w) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log o_{ik}$$

where $y_{ik} = 1$ if observation i is from class k, and $y_{ik} = 0$ otherwise, and where o_{ik} is the probability that observation i is from class k.

Batch Backpropagation Algorithm

- The Backpropagation algorithm employs a **gradient descent** to minimize the error between the network output values and the target values for these outputs.

- We will use regression problem as an example.
- With multiple output units in the network, we define error function *J(w)* to sum the errors over all of the network output units (Batch backpropagation)

$$J(w) = \frac{1}{2} \sum_{d \in D} \sum_{k \in outputs} (y_{kd} - o_{kd})^2$$

where *outputs* is the set of output units in the network, and y_{kd} and o_{kd} are the target and output values associated with the *k*th output unit and training example *d*.

- Batch backpropagation updates the weights after receiving a batch of data

Stochastic Backpropagation Algorithm

- Stochastic(on-line) backpropagation is when we omit the summation over D.

$$J(w) = \frac{1}{2} \sum_{k \in outputs} (y_k - o_k)^2$$

- The error function is defined per every sample and we update weights after reading each sample

- Faster than Batch BP, but may oscillate with high variance

- Mini-batch BP uses *n* data points (instead of one sample in SBP) at each iteration.

Training the neural network

- Backward propagation is an approach for computing gradients across the parameters in a neural-network with multiple layers in an efficient manner, in order to be able to apply gradient descent.

- One hurdle is that we do not have observed values for the hidden units, and so we cannot compare the predicted values at hidden units to observed.

- Instead, we can compute the rate at which the cost (error) function changes at these units using derivatives.

- If we have the cost function derivatives for hidden unit in a layer, then it is easy to obtain the cost function derivatives of the parameters leading to these hidden units.

Backpropagation Algorithm

- The stochastic BP algorithm is presented in next slide.
  - The algorithm applies to layered feed forward networks containing multiple layers of sigmoid units
  - Each unit at each layer is connected to all units from the preceding layer.

- x_i denotes the output of node *i*, and w_{ij} denotes its weight.

- Stochastic Backpropagation algorithm is to minimize

$$J(w) = \frac{1}{2} \sum_{k \in outputs} (t_k - o_k)^2$$

- Using Gradient Descent method

$$w_{ij} = w_{ij} + \Delta w_{ij} \quad \Delta w_{ij} = -\eta \frac{\partial J(w)}{\partial w_{ij}}$$

The (Stochastic) Backpropagation Algorithm

- Create a fully connected network.
- Initialize weights.
- Until all examples produce the correct output within ϵ (or other criteria)

For each example (x_i, t_i) in the training set do:

1. Compute the network output y_i for this example
2. Compute the error between the output and target value

$$E = \sum_{k \in outputs} \left(t_i^k - o_i^k \right)^2$$

3. Compute the gradient for all weight values, Δw_{ij}
4. Update network weights with $w_{ij} = w_{ij} + \Delta w_{ij}$

End epoch

Auto-differentiation packages such as Tensorflow, Torch, etc. help!

Backpropagation Algorithm

- In the BP algorithm, step1 propagates the input forward through the network. And the steps 2, 3 and 4 propagates the errors backward through the network.

- The main loop of BP repeatedly iterates over the training examples.
  - For each training example, 1) it computes the output & the error of the network for this example, 2) computes the gradient with respect to the error on the example, 3) then updates all weights in the network.

- This gradient descent step is iterated until the terminal condition is met

- A variety of termination conditions can be used to halt the procedure.
  - a fixed number of iterations through the loop
  - once the error on the training examples falls below some threshold
  - once the error on a separate validation set of examples meets some criteria.

Deriving the update rules

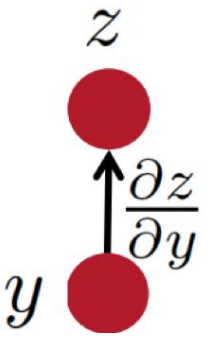
Derivation slides adapted from CIS519/419 at U Penn

Chain Rules

$$\frac{d}{dx}\left[f\left(g(x)\right)\right] = f'\left(g(x)\right)g'(x)$$

- The arrow shows functional dependence of z on y
  i.e. given y, we can calculate z.
  e.g., for example: $z(y) = 2y^2$

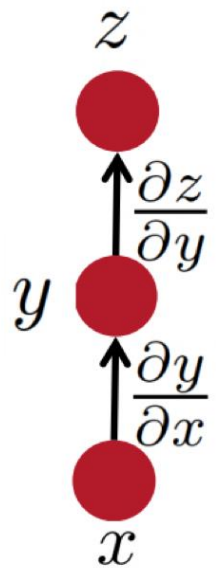
- The derivative of z, with respect to $y = \frac{\partial z}{\partial y}$



Chain Rules

- Simple chain rule
  If z is a function of y, and y is a function of x
  Then z is a function of x, as well.

- How to find $\frac{\partial z}{\partial x}$



$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$$

We will use these facts to derive the details of the Backpropagation algorithm.

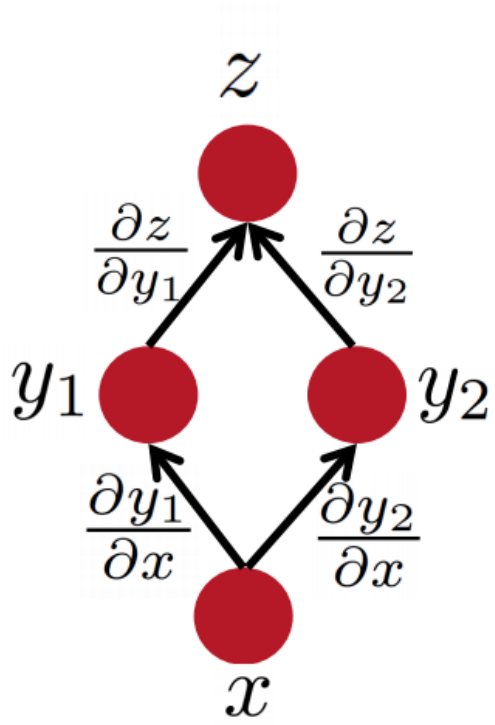
z will be the error (loss) function.
- We need to know how to differentiate z

Intermediate nodes use a logistics function (or another differentiable step function).
- We need to know how to differentiate it.

Chain Rules

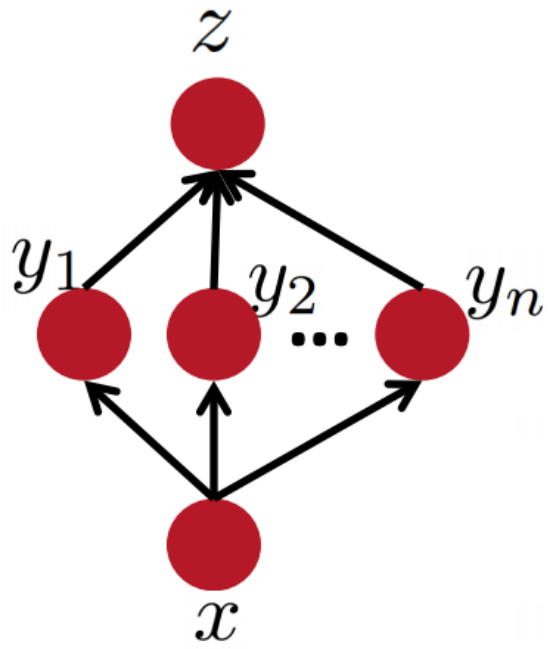
- Multiple path chain rule

$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial z}{\partial y_2}\frac{\partial y_2}{\partial x}$$



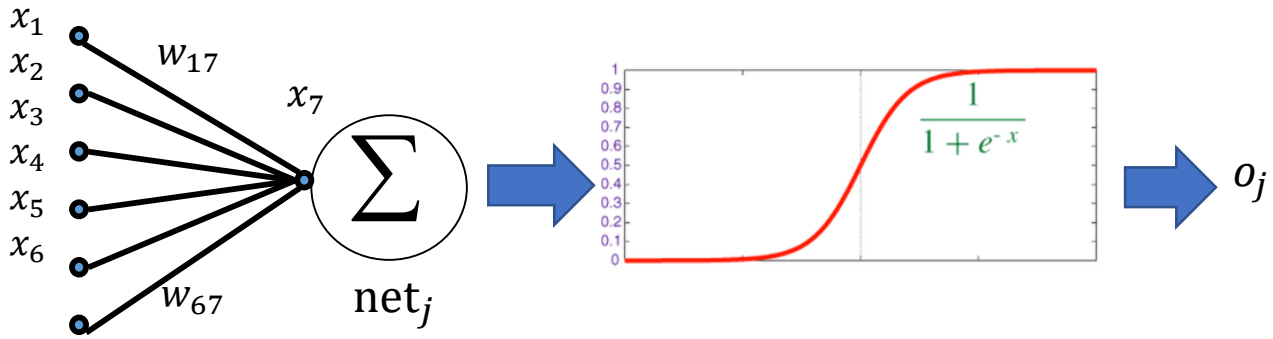
- Multiple path chain rule: general

$$\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i}\frac{\partial y_i}{\partial x}$$



Reminder: Model Neuron (Logistic)

- Neuron is modeled by a unit j connected by weighted links w_{ij} from other units i.



- Use a non-linear, differentiable output function such as the sigmoid or logistic function
- Net input to a unit is defined as: (x_i: output of node i)

$$\text{net}_j = \sum w_{ij}.x_i$$

- Output of a unit is defined as: σ: sigmoid function

$$o_j = \sigma(net_j) = \frac{1}{1+\exp(-\text{net}_j)}$$

Note:
Other gates, beyond Sigmoid, can be used (TanH, ReLu)
Other Loss functions, beyond SSR, can be used.

Derivation of Backpropagation Learning Rule: Output Unit

- Suppose the error is computed for each example (stochastic) and the error function is SSR (sum of squared error).

$$E_d(\boldsymbol{w}) = \frac{1}{2}\sum_{k \in K}(t_k - o_k)^2$$

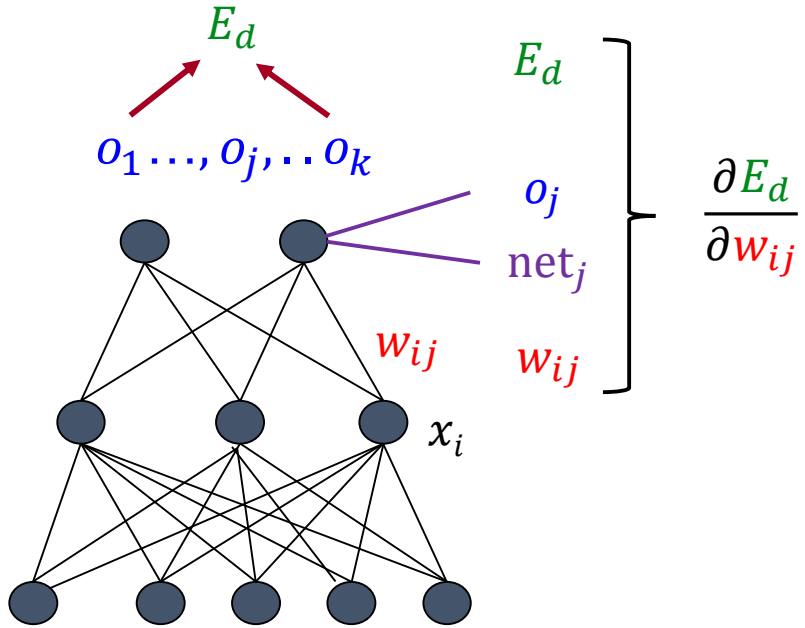
- **Weight updates of output units:**

- w_{ij} is a link weight between a hidden node x_j and an output node o_j
- w_{ij} influences the output o_j only through net_j

$$o_j = \frac{1}{1+\exp(-\text{net}_j)} \quad \text{and} \quad \text{net}_j = \sum w_{ij}.x_i$$

- We need to compute

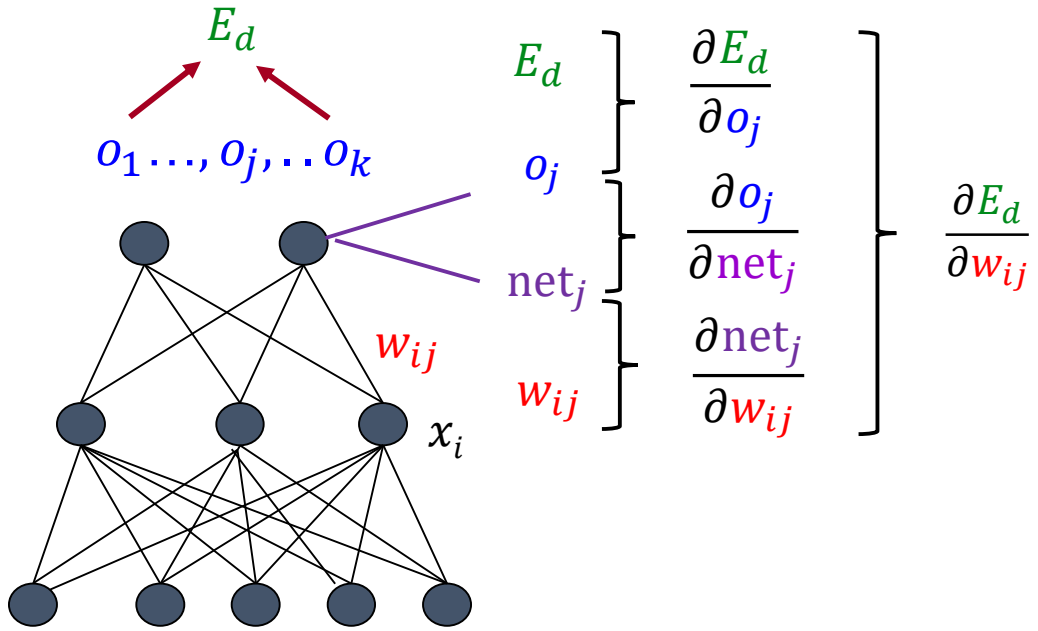
$$\frac{\partial E_d}{\partial w_{ij}}$$



Derivatives

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial o_j}\frac{\partial o_j}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}}$$

- Compute $\frac{\partial E_d}{\partial o_i}$ (error function):
  - $E_d = \frac{1}{2}\sum_{k\in K}(t_k - o_k)^2$
  - $\frac{\partial E_d}{\partial o_j} = -(t_j - o_j)$
- Compute $\frac{\partial o_j}{\partial \text{net}_j}$ (activation function):
  - $o_j = \frac{1}{1+\exp(-\text{net}_j)}$
  - $\frac{\partial o_j}{\partial \text{net}_j} = \frac{1+\exp(-\text{net}_j)}{(1+\exp(-\text{net}_j))^2} = o_j(1 - o_j)$
- Compute $\frac{\partial \text{net}_j}{\partial w_{ij}}$ (linear gate):
  - $\text{net}_j = \sum w_{ij} \cdot x_i$
  - $\frac{\partial \text{net}_j}{\partial w_{ij}} = x_i$



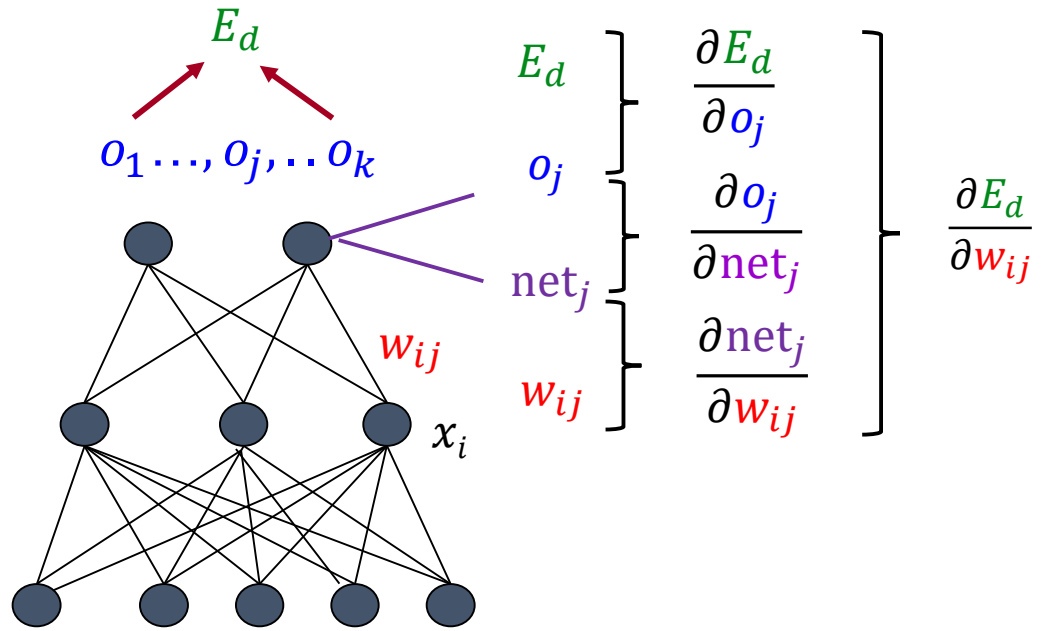
$$\sigma(x) = \frac{1}{1+e^{-x}}.$$

$$\frac{d\sigma(x)}{d(x)} = \sigma(x)\cdot(1-\sigma(x)).$$

Derivation of Backpropagation Learning Rule: Output Unit

Weight updates of output units:

- w_{ij} influences the output only through net_j
- Therefore:



$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} \frac{\partial \text{net}_j}{\partial w_{ij}}$$

$$= -(t_j - o_j)\, o_j(1 - o_j)\, x_i$$

$E_d(\boldsymbol{w})$

$$= \frac{1}{2} \sum_{k \in K} (t_k - o_k)^2$$

$$\frac{\partial E_d}{\partial o_j} = -(t_j - o_j)$$

$$o_j = \frac{1}{1 + \exp\{-\text{net}_j\}}$$

$$\frac{\partial o_j}{\partial \text{net}_j} = o_j(1 - o_j)$$

$$\text{net}_j = \sum w_{ij}.x_i$$

$$\frac{\partial \text{net}_j}{\partial w_{ij}} = x_i$$

Derivation of Backpropagation Learning Rule: Output Unit

Weight updates of output units:

- w_{ij} is updated by:

$$w_{ij} = w_{ij} + \Delta w_{ij}$$

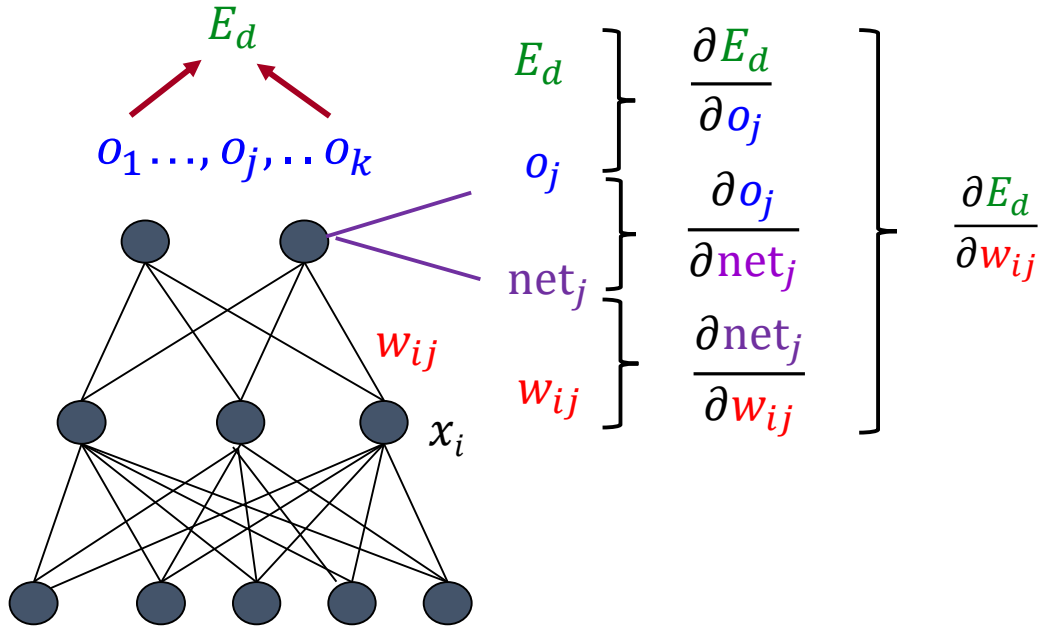
$$\Delta w_{ij} = -\eta \frac{\partial E_d}{\partial w_{ij}} = \eta (t_j - o_j) o_j (1 - o_j) x_i$$

- Let

$$\delta_j = -\frac{\partial E_d}{\partial \text{net}_j} = -\frac{\partial E_d}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} = (t_j - o_j) o_j (1 - o_j)$$

- Then

$$\Delta w_{ij} = \eta (t_j - o_j) o_j (1 - o_j) x_i = \eta \delta_j x_i$$



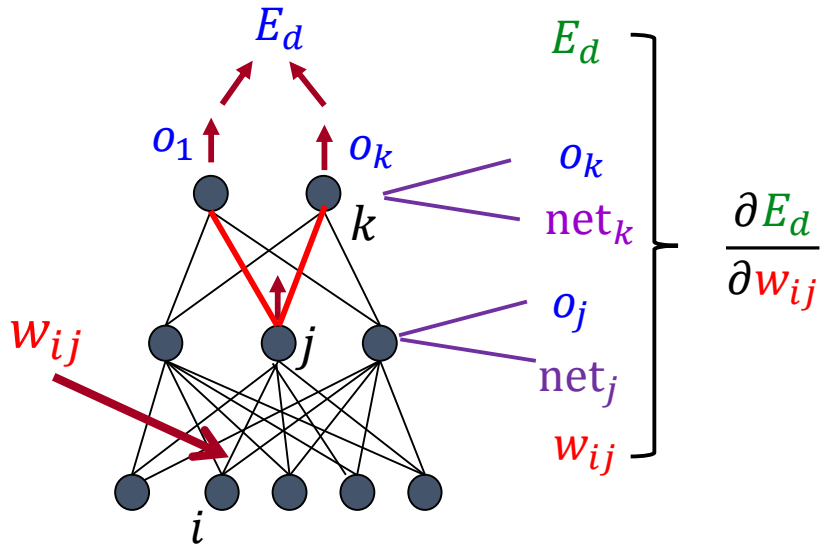
Derivation of Backpropagation Learning Rule: Hidden Unit

Weight updates of hidden units:

- w_{ij} is a link weight between hidden node i and j
- w_{ij} influences the output only through all the units whose direct input include j

- We need to compute

$$\frac{\partial E_d}{\partial w_{ij}}$$



Derivation of Backpropagation Learning Rule: Hidden Unit

Weight updates of hidden units:

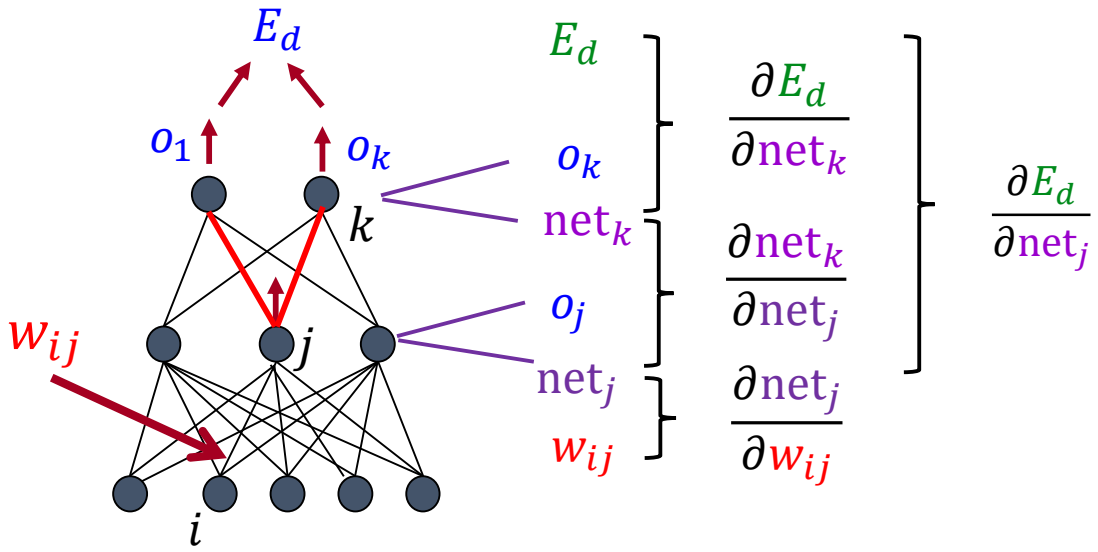
- w_{ij} Influences the output only through all the units whose direct input include j
- $parent(j)$: Set of nodes whose direct input include j

$$\frac{\partial E_d}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j}\frac{\partial \text{net}_j}{\partial w_{ij}} = \frac{\partial E_d}{\partial \text{net}_j} x_i =$$

$$\text{net}_j = \sum w_{ij}.x_i$$

$$= \sum_{k \in parent(j)} \frac{\partial E_d}{\partial \text{net}_k}\frac{\partial \text{net}_k}{\partial \text{net}_j} x_i$$

$$= \sum_{k \in parent(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i$$

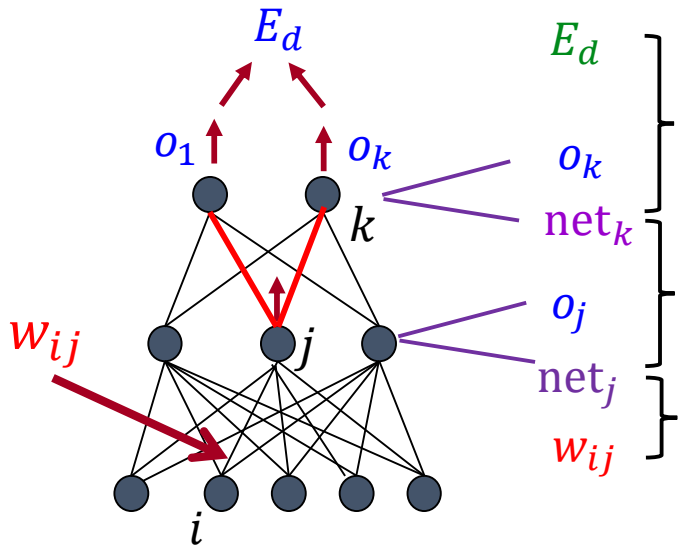


Derivation of Backpropagation Learning Rule: Hidden Unit

Weight updates of hidden units:

$$\frac{\partial E_d}{\partial w_{ij}} = \sum_{k \in parent(j)} -\delta_k \frac{\partial \text{net}_k}{\partial \text{net}_j} x_i = \sum_{k \in parent(j)} -\delta_k \frac{\partial \text{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} x_i$$

$$= \sum_{k \in parent(j)} -\delta_k \, w_{jk} \, o_j(1 - o_j) \, x_i$$



E_d

o_k

net_k

o_j

net_j

w_{ij}

$$\sum_k \frac{\partial E_d}{\partial \text{net}_k} = -\delta_k$$

$$\sum_k \frac{\partial \text{net}_k}{\partial \text{net}_j} = \frac{\partial \text{net}_k}{\partial o_j} \frac{\partial o_j}{\partial \text{net}_j} = w_{jk} o_j (1 - o_j)$$

$$\frac{\partial \text{net}_j}{\partial w_{ij}} = x_i$$

Derivation of Backpropagation Learning Rule: Hidden Unit

Weight updates of hidden units:

- w_{ij} is updated by:

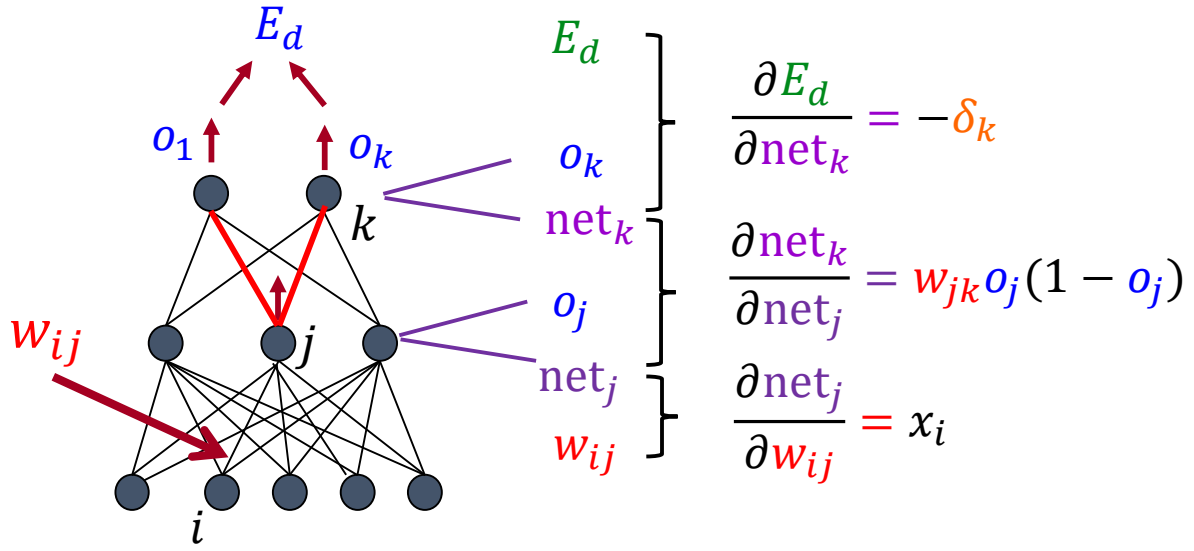
$$w_{ij} = w_{ij} + \Delta w_{ij}$$

$$\Delta w_{ij} = -\eta o_j (1 - o_j) . \left(\sum_{k \in parent(j)} -\delta_k \; w_{jk} \right) x_i$$

$$= \eta \delta_j x_i$$

where

$$\delta_j = o_j (1 - o_j) . \left(\sum_{k \in parent(j)} \delta_k \; w_{jk} \right)$$



- First determine the error for the output units.
- Then, backpropagate this error layer by layer through the network, changing weights appropriately in each layer.

Stopping Criterion

- In perceptron learning algorithm, we need to decide whether to stop or not.
- One thing we can do is:
  - Compute the cumulative squared error $J(w)$ of the perceptron at that point:
  - Compare the current value of $J(w)$ with the value of $J(w)$ computed at the previous iteration.
  - If the difference is too small (e.g., smaller than 0.00001) we stop.

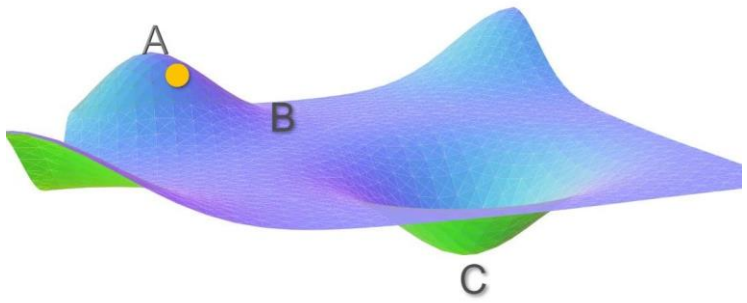
Adding Momentum

- Many variations of BP have been developed.
- Momentum: alter the weight-update rule in Step 4 in the algorithm by making the weight update on the *n*th iteration *depend partially on* the update on the (*n* -1)th iteration, as follows:

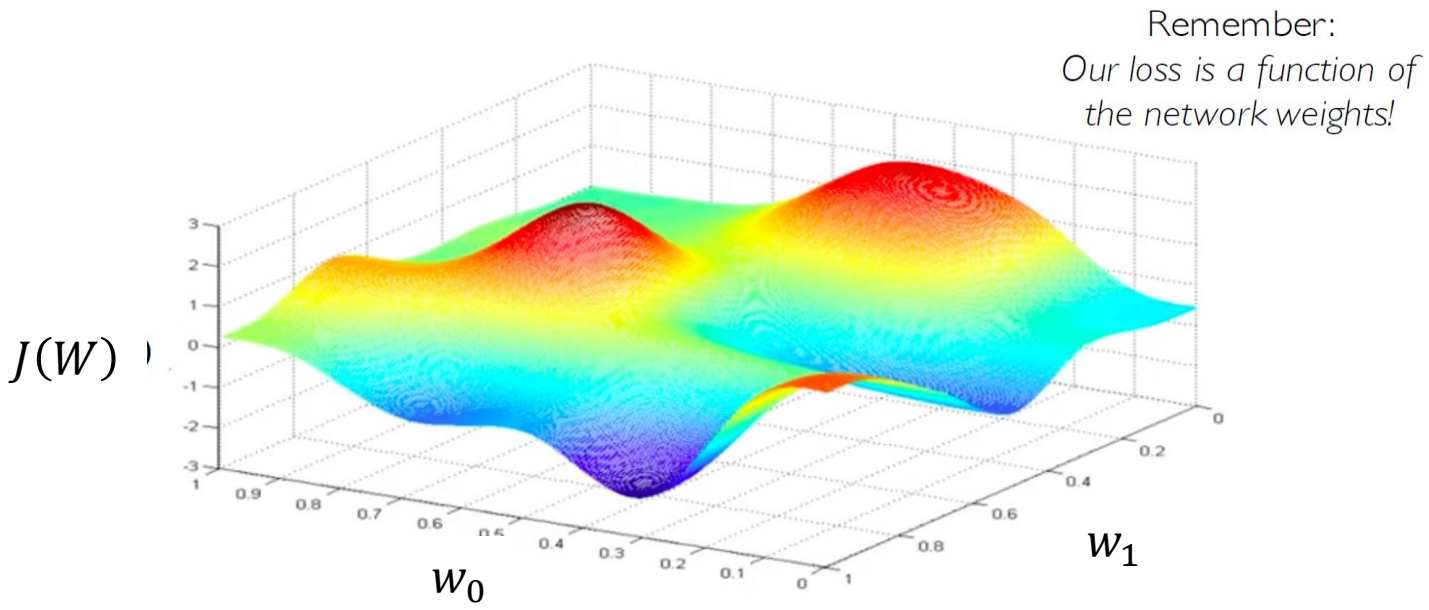
$$\Delta w_{ij}(n) = \eta \delta_j x_i + \alpha \Delta w_{ij}(n-1)$$

- Here $\Delta w_{ij}(n)$ is the weight update performed during the *n*-th iteration through the main loop of the algorithm.
  - *n*-th iteration update depend on (*n-1*)th iteration
  - α: constant between 0 and 1 is called the *momentum.*

- Role of momentum term($\Delta w_{ij}(n-1)$):
  - keeps the ball rolling through small local minima in the error surface.
  - avoids saddle points



Error Surface of Multilayer Network



- Unlike perceptron surface, we have local optimum problem

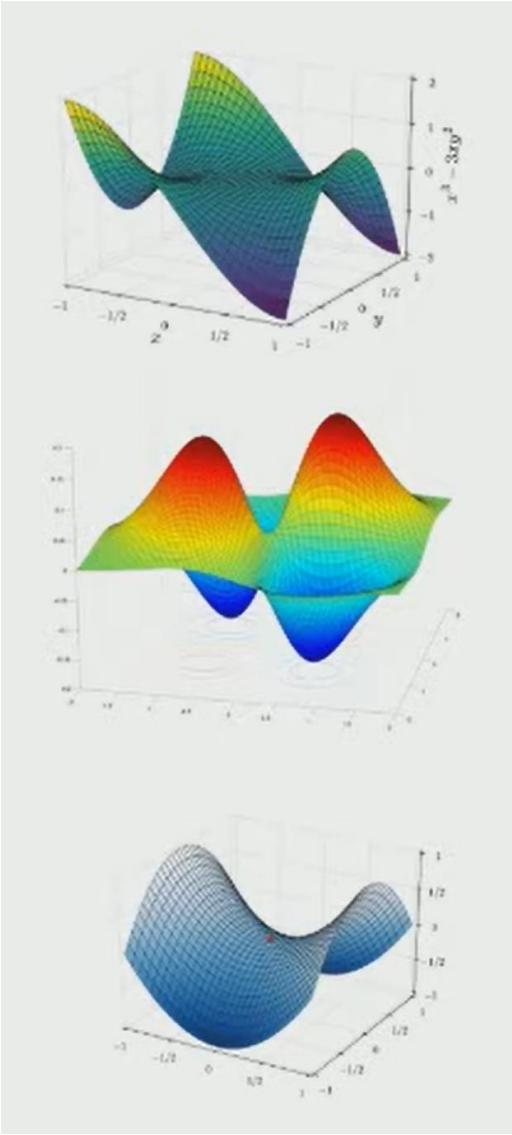
The Error Surface

Popular hypothesis:

- In large networks, saddle points are far more common than local minima
  - Frequency exponential in network size
- Most local minima are equivalent
  - And close to global minimum
- This is not true for small networks

Saddle point: A point where

- The slope is zero
- The surface increases in some directions, but decreases in others
- Gradient descent algorithms often get "stuck" in saddle points



What Neural Networks Can Compute

- An individual perceptron is a linear classifier.
  - The weights of the perceptron define a linear boundary between two classes.
- Neural networks with one hidden layer can compute any continuous function. (will be discussed in deep learning chapter)
- Neural networks with one hidden layer can compute any classification boundary. (will be discussed in deep learning chapter)
- This has been known for decades, and is one reason scientists have been optimistic about the potential of neural networks to model intelligent systems.
- Another reason is the analogy between neural networks and biological brains, which have been a standard of intelligence we are still trying to achieve.

Remarks on Backpropagation Algorithm

- Gradient descent to some local minimum
  - Perhaps not global minimum...

- Heuristics to alleviate the problem of local minima
  - Add momentum
  - Use stochastic gradient descent rather than true gradient descent.
  - Train multiple nets with different initial weights using the same data.

- There are many advanced methods in activation function, learning rates, momentum, regularization, etc

Remarks on Backpropagation Algorithm

- The perceptron may change greatly upon adding just a single new training instance
  - But it fits the training data well
  - The perceptron rule has low bias
    - Makes no errors if possible
  - But high variance
    - Swings wildly in response to small changes to input

- Backprop is minimally changed by new training instances
  - Prefers consistently over perfection
  - It is a low-variance estimator, at the potential cost of bias