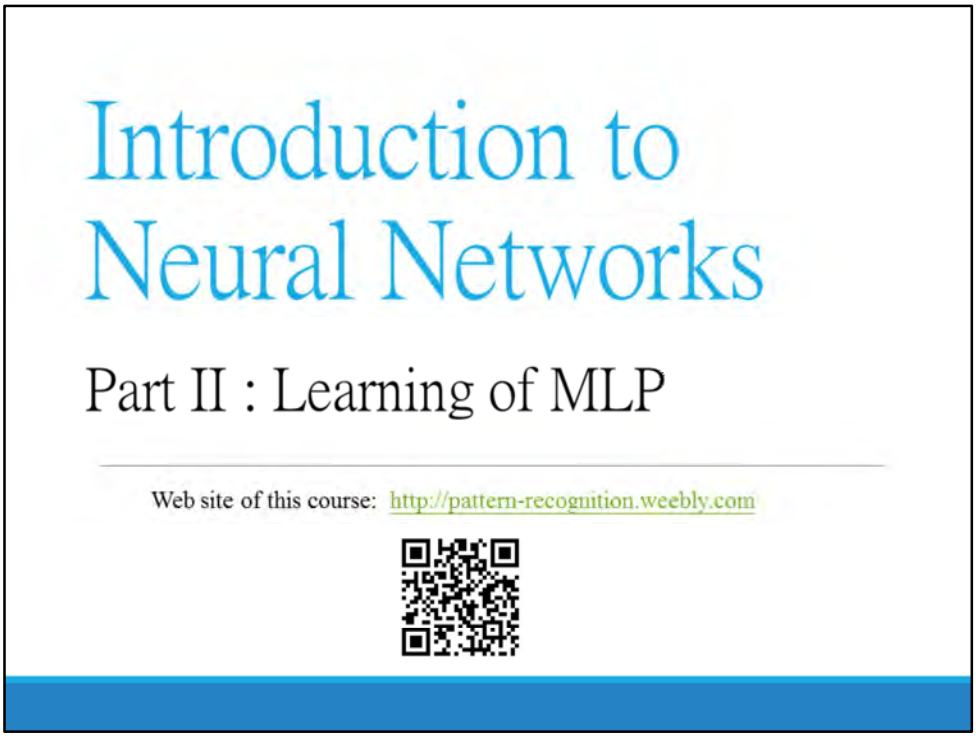
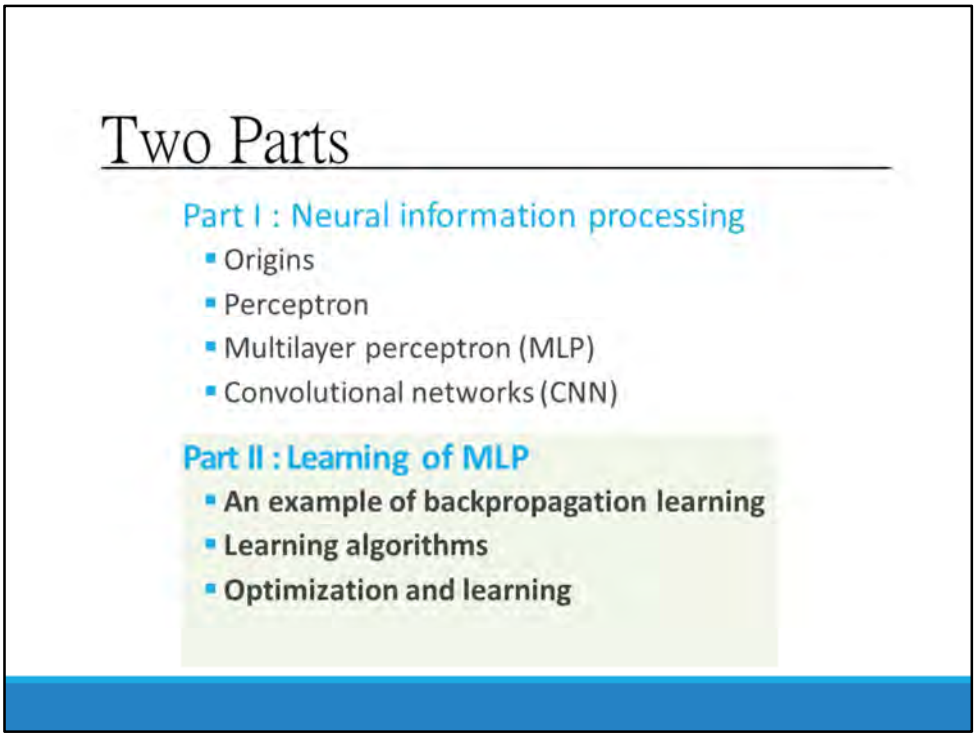
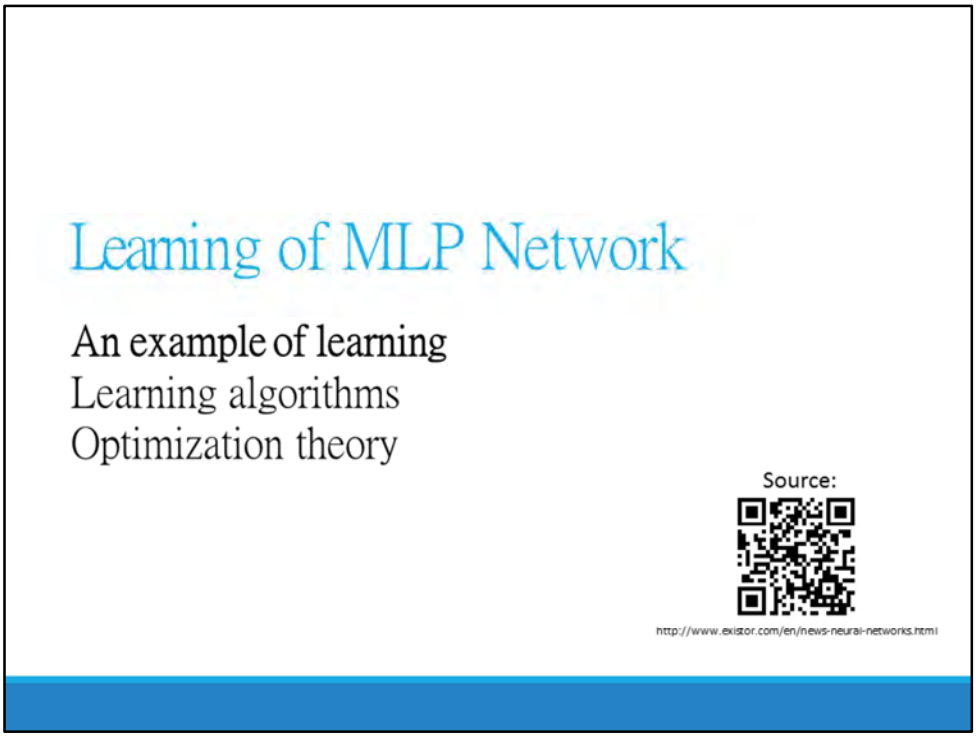


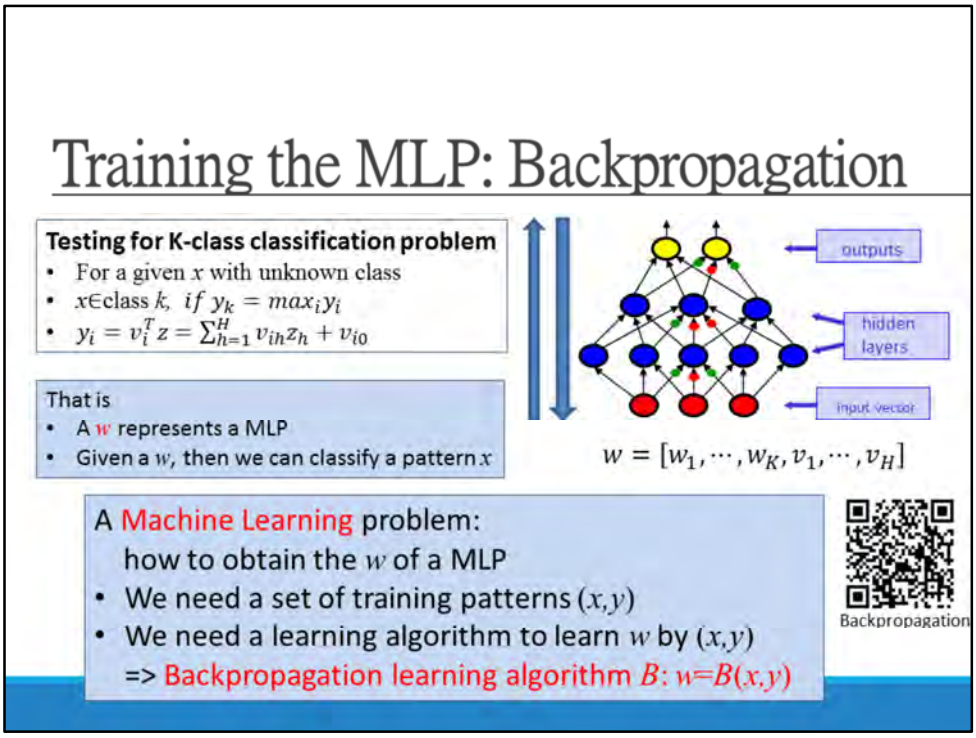
- This is the fourth lecture note of the course PATTERN RECOGNITION in English in 104-2 semester, Electrical Engineering department, Fu-Jen Catholic University.
- This course is taught by Prof. Wang, Yuan-Kai.
- In this lecture note, I will explain the backpropagation learning algorithm for MLP networks.
- Web site of this course: http://pattern-recognition.weebly.com.



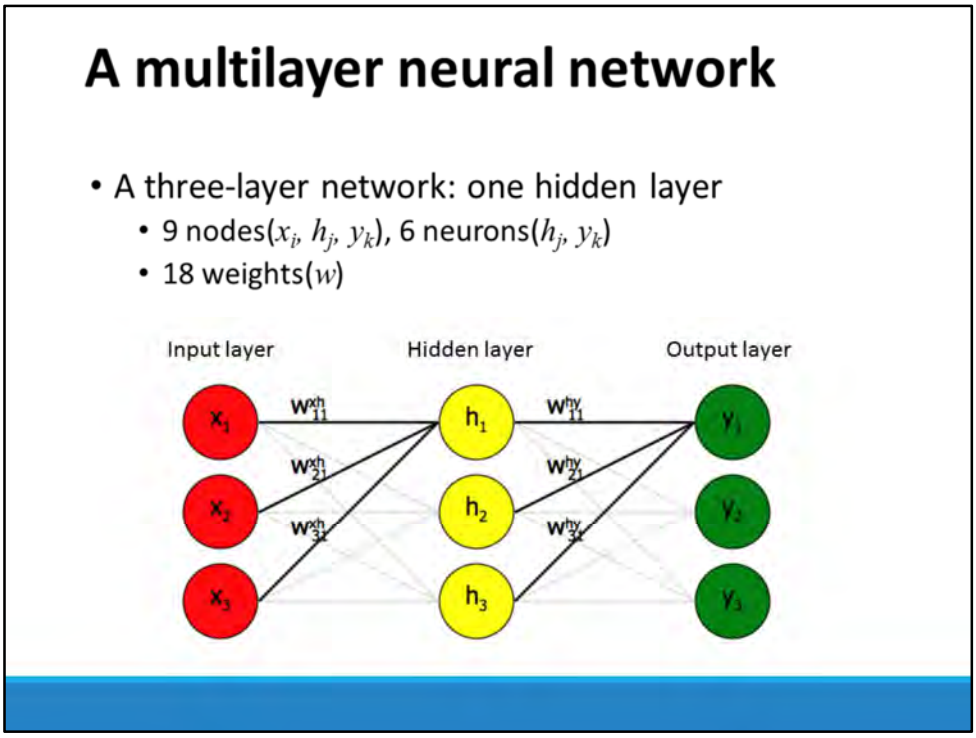
- The second topic: training of neural networks, includes 4 sections. The second lecture will also be lectured by 1 week.



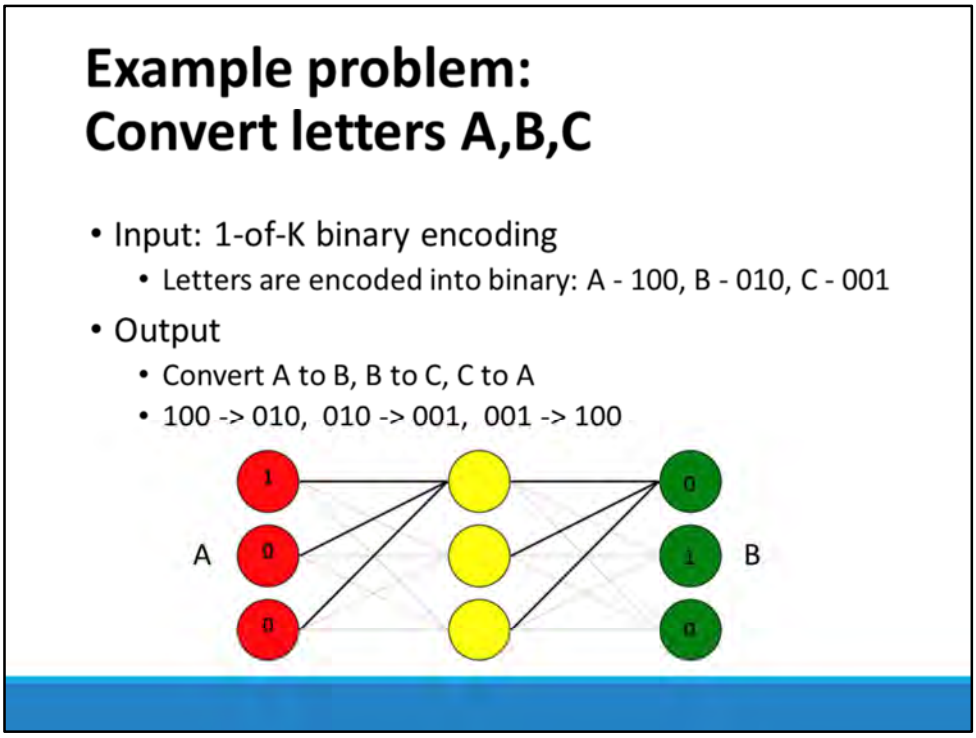
- This section gives an example of training of neural network.
- Source of the example: Machine Learning - Neural Networks Tutorial, Paul Tero of Existor Ltd, 2015/8/20.
    - http://www.existor.com/en/news-neural-networks.html.
- This example uses an online learning (stochastic gradient descent)



- **Propagation**: from input layer to output layer
  - For testing of classification
- **Back-propagation**: from output layer to input layer
  - For learning of classification



- In neural networks, the neurons are called nodes.
- The red nodes on the left are inputs, such as your eyes and ears.
- The green ones are outputs, such as your muscles or vocal cords. The yellow ones are extra ones in the middle that help to learn things.
- In neural networks, they are called hidden nodes, and they are part of a hidden layer.
- A neural network can have any number of hidden layers, but this one only has one. This is a three layer neural network.
- In neural networks all the nodes in one layer are connected to all the nodes in the next layer.
- Input nodes are generally awarded x and output nodes y. Hidden nodes are usually given h.
- In neural networks each connection has a weight associated with it, so normally the letter w is used.
    - But there are lots of these connections, 18 in this very small graph, about 7500 in the roundworm and 100,000,000,000,000 in a human adult.
    - So in this graph, the weight has a superscript indicating its position (such as xh for a weight between the x input layer and h hidden layer) and a subscript for its number (21 means between the second node on the left and first node on the right).

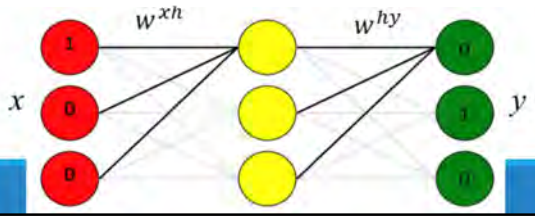


- We will train this one to tell us the next letter in the alphabet.
    - So if we present it with an "A", it should output "B", etc.
- The first problem is how to give it the letter "A".
    - Computers and neural networks only deal with numbers. So we could say that A=1, B=2, C=3.
    - But this implies an order and structure which really isn't there. It implies that B lies between A and C.
- So instead we'll use binary inputs.
    - Since our network only has three input nodes, we can only represent 3 letters.
    - The letter A will be represented as 100, B as 010 and C as 001.
    - This is called the 1-of-k encoding. Each input is a bunch of 0s and a single 1.

Training of the network

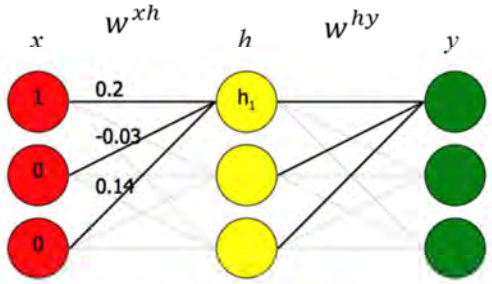
- Given a training pair (x,y)
  - x: input values, y: desired output values
- Network training will get a weight matrix $w=(w^{xh},w^{hy})$
- Basic steps to train the network
  1. Randomly initialize the weight matrix w
  2. Forward propagation: $y'=xw$
  3. Compute the error: $E=y-y'$
  4. Compute weight change value by the error: $\Delta w=f(E)$
  5. Backpropagation: $w = w - \Delta w$
  6. Go to step 2

supervised learning

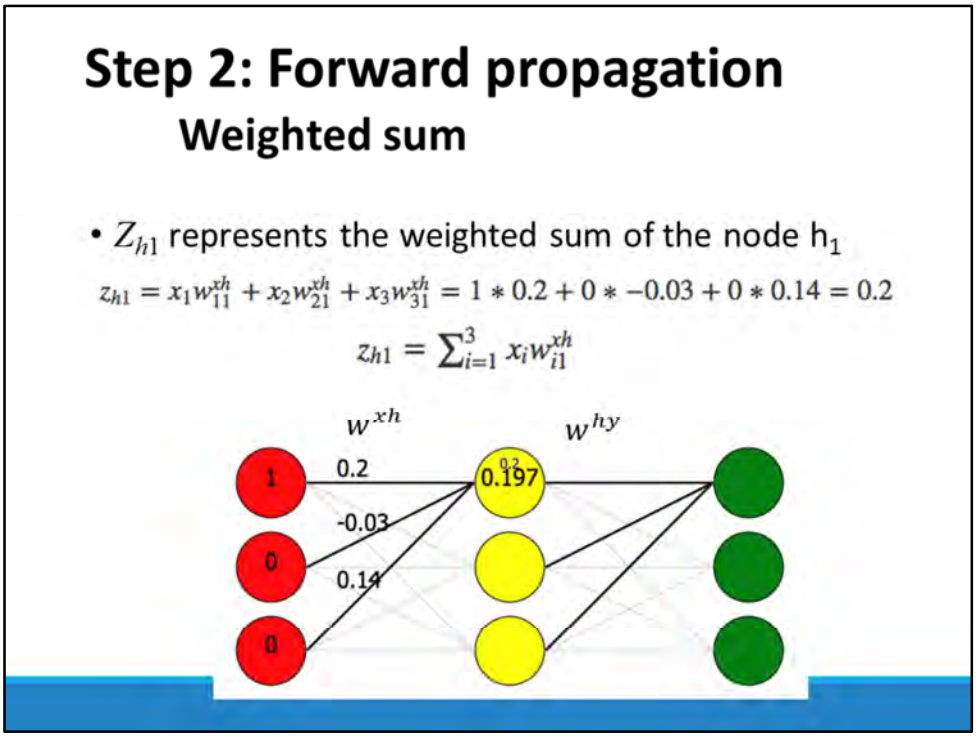


Step 1: Random starting weights

- Now we will compute the values of the first hidden node h_1 in the second layer
- The weights are usually initialised to be small random values between -1 and 1



- Now we will compute the values of the nodes in the second layer starting with the first hidden node h1.
- The weights in a neural network are usually initialised to be small random values between -1 and 1.
- This is very important because it ensures that the different connections learn different things. I've assigned some small random weights to the connections in the first layer.



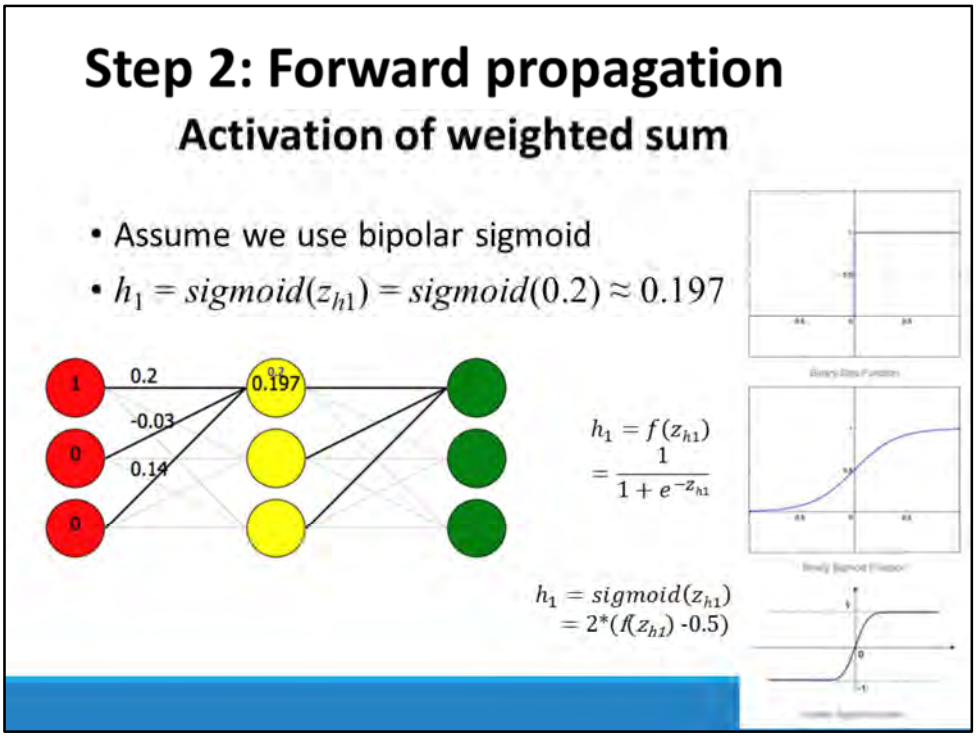
- First we have to compute the weighted sum of the inputs. This means we multiply each input times each weight and add them together.
- So we compute the weighted sum z for h1:

$$z_{h1} = x_1 w_{11}^{xh} + x_2 w_{21}^{xh} + x_3 w_{31}^{xh} = 1 * 0.2 + 0 * -0.03 + 0 * 0.14 = 0.2$$

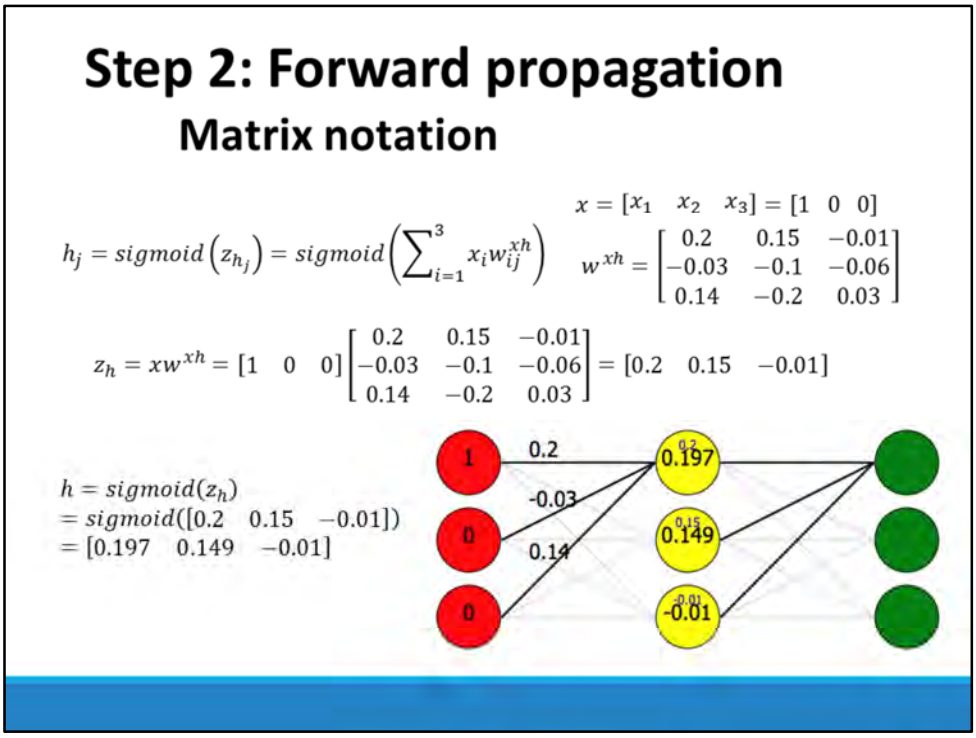
- We took the value of the first input x1 (1) and multiplied it by the weight connecting it to h1 (0.2). We did this for the second and third inputs as well and added the results together to get 0.2.
    - This can be written more succinctly using summation notation:

$$z_{h1} = \sum_{i=1}^{3} x_i w_{i1}^{xh}$$

    - The ∑ is the Greek letter Sigma and means "take a sum". It has a variable i which goes from the values 1 up to 3. So this equation takes the sum over all three inputs, just as the longer equation above.



- Activation function
  - In animal brains, a neuron either fires, passing on an electrochemical signal to other neurons, or it doesn't fire. It is thought that neurons perform a similar operation to the one above, adding up all their inputs, multiplying by a set of weights, and then "deciding" whether to fire or not, depending on whether the answer exceeded some preset threshold.
  - A neural network does something similar using an activation function. It is needed because, if the raw weighted sums computed from the first layer were passed directly to the second layer, a neural network would just be a calculator. It needs a non-linear function to separate each layer.
  - So the sum computed above is sent through an activation function such as **sigmoid** or **tanh** which converts the incoming sum into either a -1 (to mimic a neuron not firing) or +1 (for firing). Unlike a real neuron, these functions also have a limited range of values in the middle, so they can result in -0.25 or 0.01 or just 0. This helps make the neural network less rigid.
- If our network use the **bipolar sigmoid** function (output ranges from -1 to 1)
  - For numbers around 0, sigmoid is also around 0.
  - For numbers greater than 2, sigmoid gets ever closer to 1.
  - For less than -2 it approaches -1.
  - The actual value of the activation of h1 for 0.2 is : h1=sigmoid(zh1)=sigmoid (0.2) ~ 0.197.
- Our weights were initialised to very small random values, so at first all the activation values will be around 0. But after the network learns and becomes more confident, the weights will rise or fall, and many of the activation values may go to near -1 or +1.



- To compute all the activation values in the hidden nodes, we can rewrite the equation above so it applies to all the activation nodes in the h layer.
    - i refers to number of the input node, so this equation applies to x1, x2 and x3.
    - j refers to number of the hidden node, so this equation applies to h1, h2 and h3.
- To compute the activation values, we also need to know the other weights. Suppose the weights between input layer and hidden layer, w^xh, are given in the slide with a 3x3 matrix.
    - In this matrix the first row represents the weights coming from x1 going into the three hidden h nodes.
    - The second row is for x2 and so on.
    - The first column therefore represents weights going into h1 (these are the weights shown in the graph), the second column into h2 and so on.
- The three weighted sums for h can now be computed using matrix multiplication. And finally we can compute the three activation values for h=(h1, h2, h3).

Step 2: Forward propagation

Output layer

- Assume w^{hy} are the weights between hidden and output layers

$$w^{hy} = \begin{bmatrix} 0.08 & 0.11 & -0.3 \\ 0.1 & -0.15 & 0.08 \\ 0.1 & 0.1 & -0.07 \end{bmatrix}$$

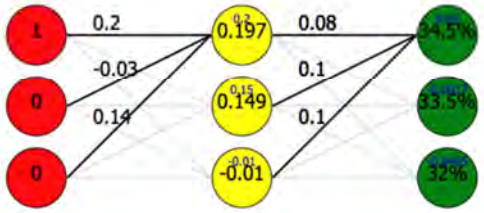
$$z_y = hw^{hy}$$

$$= [0.197 \quad 0.149 \quad -0.01] \begin{bmatrix} 0.08 & 0.11 & -0.3 \\ 0.1 & -0.15 & 0.08 \\ 0.1 & 0.1 & -0.07 \end{bmatrix} = [0.03 \quad -0.0017 \quad -0.0465]$$

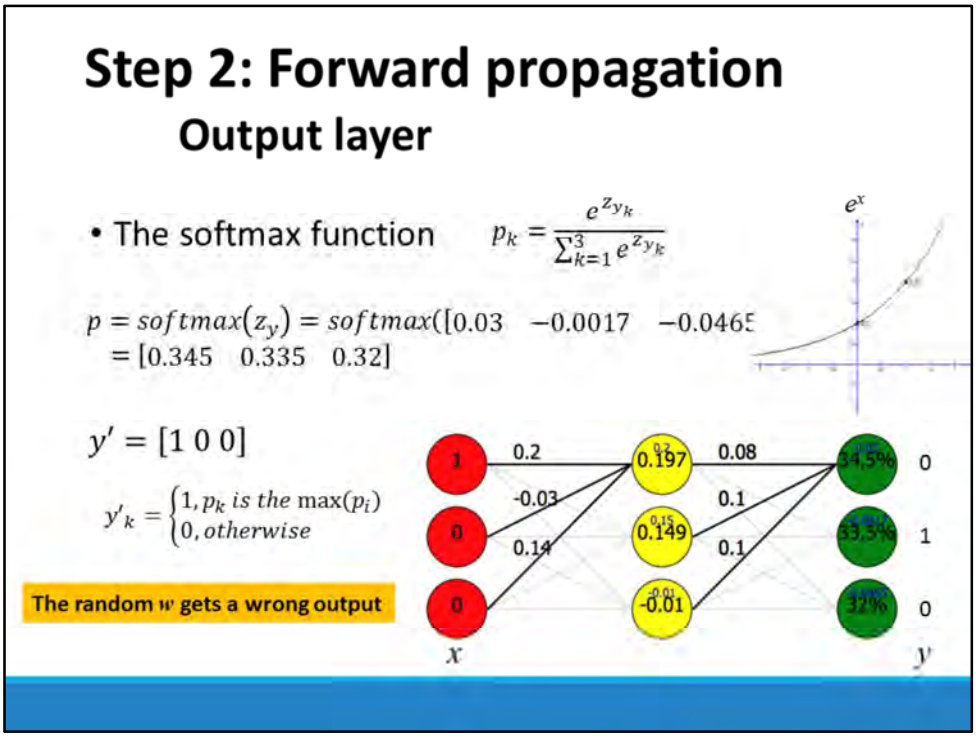
~~$y_k = sigmoid(z_{y_k})$~~

~~$= sigmoid\left(\sum_{j=1}^{3} h_i w_{jks}^{hy}\right)$~~

We usually use softmax function for output nodes, but not sigmoid. See next slide.



- We now do a similar thing for the output layer of the network.
- we first compute the weighted sums for the output layer y: z_y.
- However, unlike the hidden layer, the output layer does not have an activation function.
- There are two types of output nodes: real values and binary
  - For regression or real-value problemns, the direct output z_y are used as the neural network output.
  - For classification or binary-value problems, the direct output z_y will be passed to a "softmax" function to produce "probability values" of output nodes.
- In this example, we need to output binary values, and we need to use "softmax" to produce probabilities.

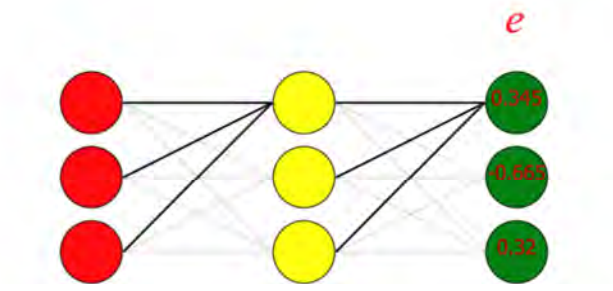


- The softmax function is important in the field of machine learning because it can map a vector to a probability of a given output in binary classification.
- Softmax function, wikipedia: https://en.wikipedia.org/wiki/Softmax_function
    - In mathematics, in particular probability theory and related fields, the softmax function, or normalized exponential, is a generalization of the logistic function that "squashes" a K-dimensional vector z of arbitrary real values to a K-dimensional vector σ(z) of real values in the range (0, 1) that add up to 1.
    - In neural network simulations, the softmax function is often implemented at the final layer of a network used for classification. Such networks are then trained under a log loss (or cross-entropy) regime, giving a non-linear variant of multinomial logistic regression.
- exp(0.03)=1.030454534, exp(-0.0017)=0.9983014442, exp(-0.0465)=0.9545645606.
- The result y=[1 0 0] is a wrong answer. The right answer should be y=[0 1 0].
    - It was supposed to answer B given the input A, but it got it wrong.
    - This is not at all surprising since all its weights were completely random and we haven't actually taught it anything.
    - The whole process above is known as the feed forward phase, because input values are fed forward into the network, until they get to the end, and output values pop out.

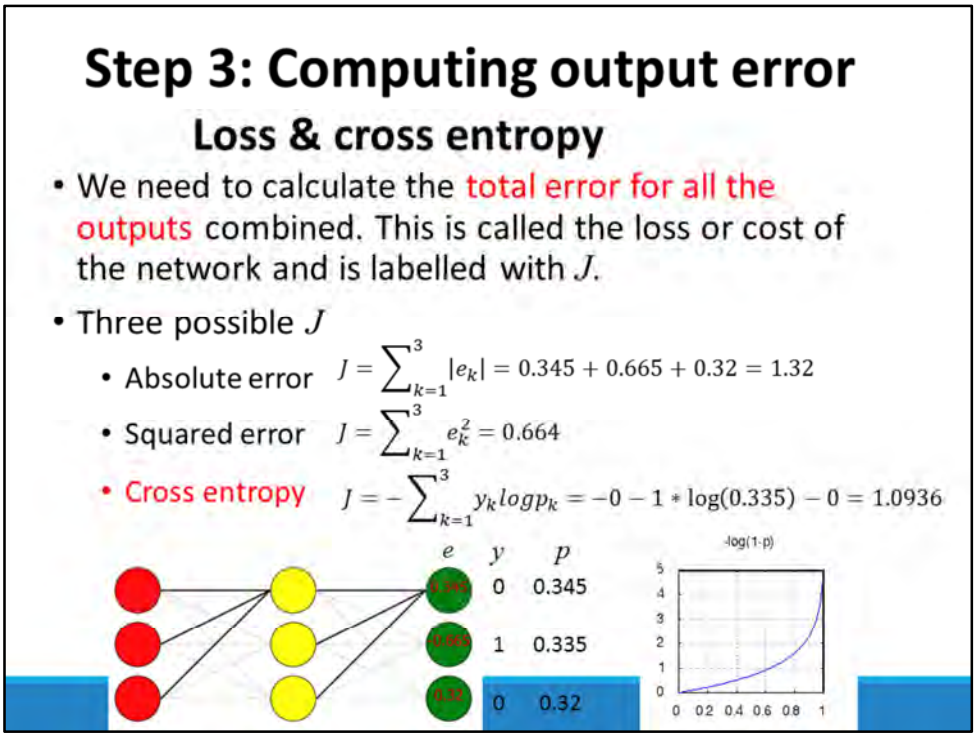
Step 3: Computing output error

$$y = [0 \quad 1 \quad 0], p = [0.345 \quad 0.335 \quad 0.32]$$

$$e = p - y = [0.345 \quad 0.335 \quad 0.32] - [0 \quad 1 \quad 0]$$
$$= [0.345 \quad -0.665 \quad 0.32]$$



- We will continue to train the network, going back through it and correcting the output error.
- But first we need to know where we went wrong. Our network is an example of supervised learning because we know the right answer.
- We know that given A, it should answer B. We can therefore compute the output error, how much it got each answer wrong by. This error is just the actual output minus the expected output.

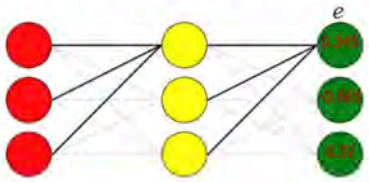


- We need to calculate the total error for all the outputs combined.
- This is called the loss or cost of the network and is often labelled with the capital letter J.
- This number is very useful as it enables us to very quickly see how well the network is doing, and how fast it is learning.
- Three possible J
    - Absolute values : they aren't very mathematically-friendly though (because they can't be differentiated).
    - Squared values: the squared error is often used.
    - Cross entropy
        - But for softmax classification networks a more efficient alternative is cross-entropy which is a type of log loss.
        - We know that our actual output p is a probability distribution.
        - We can also view y as a probability distribution. In this example, the probability of A is 0%, B is 100% and C is 0%.
        - In information theory, cross entropy is a way of measuring the difference between two probability distributions and is often used for classification networks.
        - The negative of the log function between 0 and 1 looks like the graph below. So if we have 0 error (if y=1 and we predicted p=1), then the loss will also be zero. If we got it very wrong (y=1 and we predicted p=0), the loss will be very high, approaching infinite.
- That loss number of 1.0936 is valuable. It is a real and concrete estimate of how well our network performed. We can measure it after every training example. If after hours of training, the loss is about the same, then we know we're not getting anywhere. If the loss falls dramatically at first and then settles at 0.04, then we know that we may as well stop. If it goes up, then we probably made a mistake.

Step 4: Adjusting weights

Intuition

- It feels like
  - The weights going into y_1 and y_3 should be lowered a bit, because their estimate was too high.
  - The weights going into y_2 should be raised because they were way too low and caused a large negative error.
  - The bigger the error, the more the weights should be changed.



- Now we are ready to go back and correct our network's mistakes. There are 18 adjustable weights in our network, and we have to change each of them a little bit so that the next time we input a letter A, the network is more likely to answer B. We'll start with the output weights.

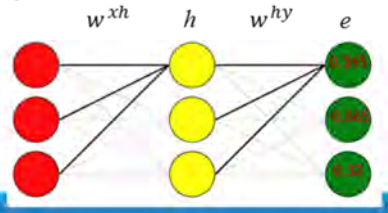
Step 4: Adjusting weights

Formula

- Mathematically the intuition is fairly easy to do.
- The error δw of the weight w
  - is proportional to the size of the thing on the other end of the connection (the activation value of the hidden node). a
  - So we can just multiply the value of the hidden node h_j times the error e_k to get δw_{jk}^{hy}

$$w_{jk}^{hy} = w_{jk}^{hy} - \delta w_{jk}^{hy}$$

$$\delta w_{jk}^{hy} \propto h_j * e_k$$



- We use the Greek letter delta δ for weight changes.

Step 4: Adjusting weights

An example

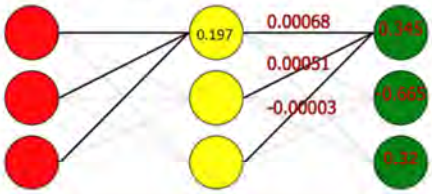
- Assume a learning rate α = 0.01

  $\delta w_{jk}^{hy} = \alpha * h_j * e_k \propto h_j * e_k$

- For example, the adjustment on the top weight connecting the first hidden node to the first output node, δw_{11}^{hy}, could just be:

  $\delta w_{11}^{hy} = \alpha * h_j * e_k$ = 0.01*0.197*0.345 = 0.00068

$$w_{11}^{hy} = w_{11}^{hy} - \delta w_{11}^{hy}$$
$$= 0.08 - 0.00068$$
$$= 0.07932$$



- The w is decreased from 0.08 to 0.07932
  - That would in turn lower the activation value of the first output node to 0.0295 and lower the probability of the answer being A by about 0.004%.
  - It is not much, but this process will be repeated thousands of times before the network is done learning.

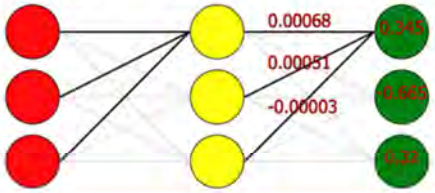
Step 4: Adjusting weights

Matrix

- We can compute all the adjustments δw^{hy} with one matrix operation. Assume a learning rate α = 0.01

$$\delta w^{hy} = \alpha h^T e = 0.01 \begin{bmatrix} 0.197 \\ 0.149 \\ -0.01 \end{bmatrix} [0.345 \quad -0.665 \quad 0.32]$$

$$= \begin{bmatrix} 0.00068 & -0.00131 & 0.00063 \\ 0.00051 & -0.00099 & 0.00047 \\ -0.00003 & 0.00007 & -0.00003 \end{bmatrix}$$



- We can compute all the adjustments with one matrix operation.
- We multiply the hidden values times the error values. Except we flip the hidden values to make them into a vertical vector (by adding the T which stands for "transpose"). Then when we multiply the 3 rows of hT times the 3 columns of eyery and we get a 9 element matrix:

Step 4: Adjusting weights

Theory

- Why the formula?

$$w_{jk}^{hy} = w_{jk}^{hy} - \delta w_{jk}^{hy}$$

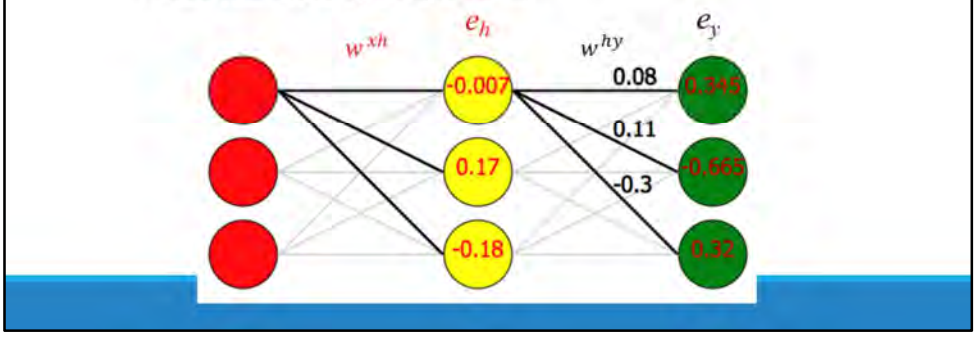
$$\delta w_{jk}^{hy} \propto h_j * e_k$$

- The theory of weights adjustment
  - Gradient descent, partial derivatives
  - The theory of optimization

Step 5: Backward propagation

Basic concept

- In Step 4 we use the error e_y to update w^{hy}
- Here we need to further update w^{xh}
  - Backpropagate the error of output layer e_y to hidden layer: the error of hidden layer e_h
  - Use the error e_h to update w^{xh}



- We must propagate the errors back to the previous layer, so that we can adjust the weights between the input and hidden layers.
- The first part of this is fairly easy. We just run the neural network in reverse to get the hidden node's errors.

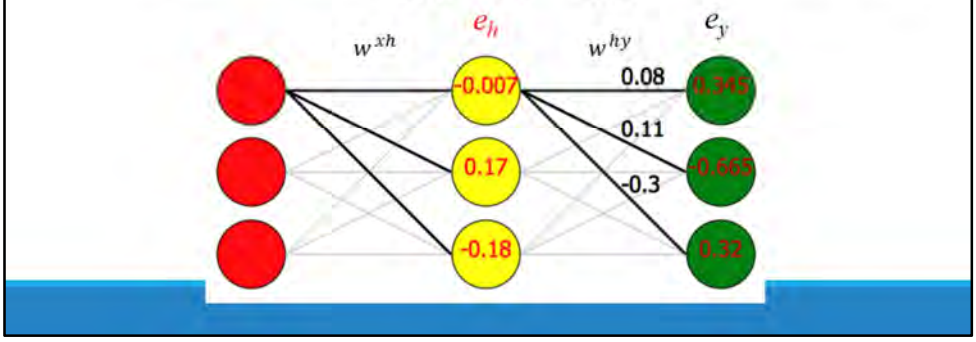
Step 5: Backward propagation

Error propagation

- Backpropagate the error of output layer e_y to hidden layer: the error of hidden layer e_h

$$e_h = e_y w^{hy}$$

$$= [0.345 \quad -0.665 \quad 0.32] \begin{bmatrix} 0.08 & 0.11 & -0.3 \\ 0.1 & -0.15 & 0.08 \\ 0.1 & 0.1 & -0.07 \end{bmatrix} = [-0.007 \quad 0.17 \quad -0.18]$$



- We must propagate the errors back to the previous layer, so that we can adjust the weights between the input and hidden layers.
- The first part of this is fairly easy. We just run the neural network in reverse to get the hidden node's errors.

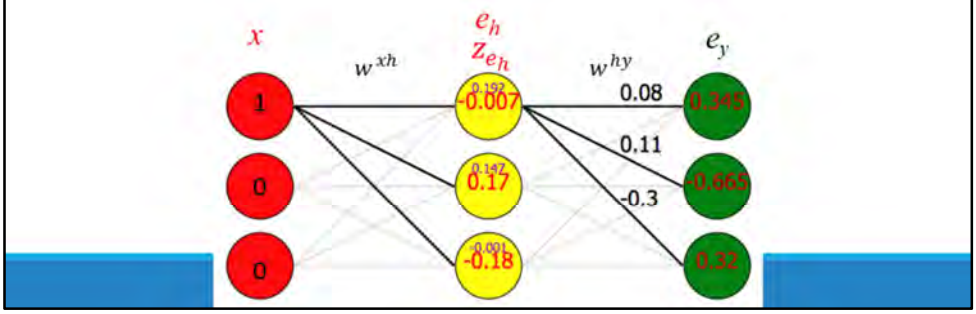
Step 5: Backward propagation

$$z_{e_h} = e_h \odot \left(1 - sigmoid^2(z_h)\right)$$

$$= [-0.007 \quad 0.17 \quad -0.18] \odot [0.961 \quad 0.978 \quad 0.999] = [0.192 \quad 0.147 \quad -0.001]$$

$$\delta w^{xh} = \alpha x^T z_{e_h} = 0.01 \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} [0.192 \quad 0.147 \quad -0.001]$$

$$= \begin{bmatrix} 0.00192 & 0.00147 & -0.00001 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

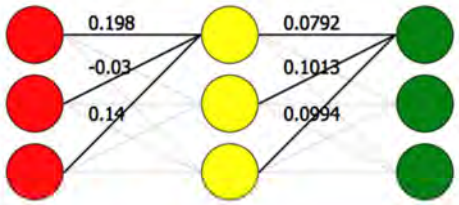


Step 5: Backward propagation

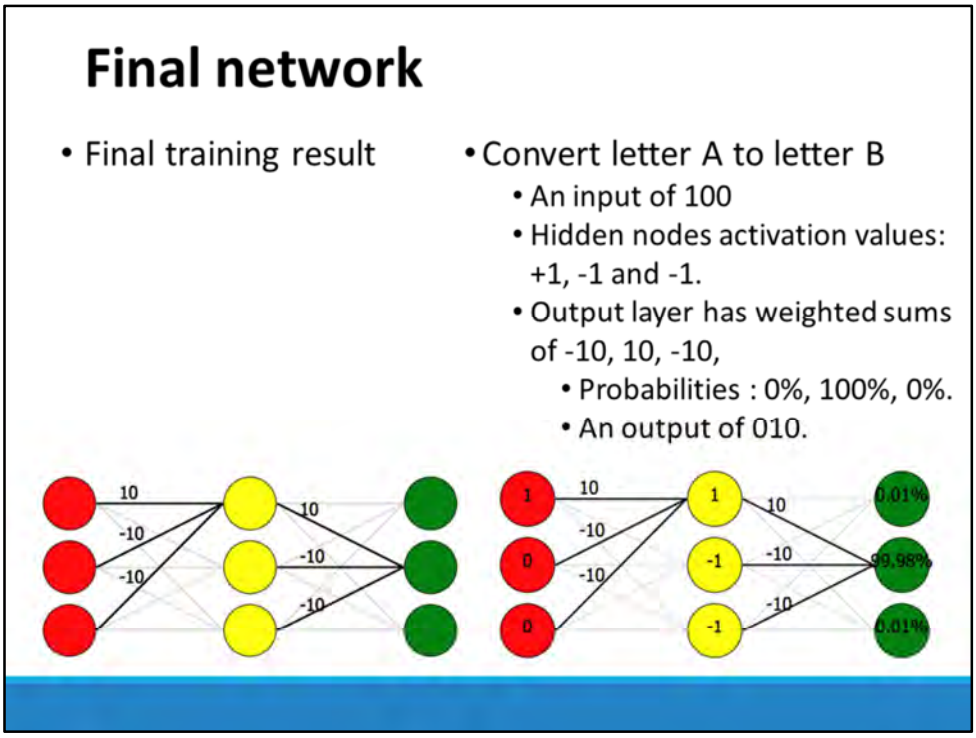
Changing weights

$$w^{xh} = \begin{bmatrix} 0.2 & 0.15 & -0.01 \\ -0.03 & -0.1 & -0.06 \\ 0.14 & -0.2 & 0.03 \end{bmatrix} \quad \delta w^{xh} = \begin{bmatrix} 0.00192 & 0.00147 & -0.00001 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

$$w^{xh} - \delta w^{xh} = \begin{bmatrix} 0.19808 & 0.14853 & -0.00999 \\ -0.03 & -0.1 & -0.06 \\ 0.14 & -0.2 & 0.03 \end{bmatrix}$$



- This complete the training of one training example (x,y).
- And then it is ready to receive the next training example (x,y).

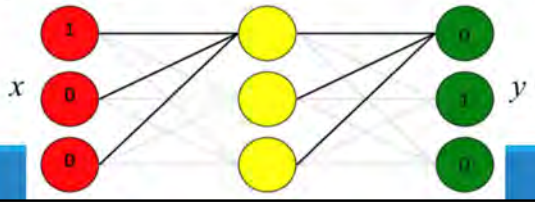


- Left graph: final training result
  - Our network only has three possible training examples: A->B, B->C and C->A.
  - Once we've given these training examples to the network thousands of times, it should end up learning the appropriate weights.
  - The final learned weights might look something like the left graph.
- Right graph: Convert letter A to letter B
  - What happens in this network is that an input of A=100 will cause the first hidden node to have a weighted sum of 10, and the other nodes to have weighted sums of -10.
  - The sigmoid will give the hidden nodes activation values of +1, -1 and -1.
  - The output layer will have weighted sums of -10, 10, -10, which will lead to probabilities of close to 0%, 100% and 0%, which is an output of 010.
- Note: in fact, this is a very simple problem and doesn't require a hidden layer.

Summary of the Single-sample Training

- Given a single training sample (x,y)
  - x: input values, y: desired output values
- Network training will get a new weight matrix w
- Basic steps to train the network
  1. Randomly initialize the weight matrix $w=(w^{xh},w^{hy})$
  2. Forward propagation: $y'=xw$
  3. Compute the error: $E=y-y'$
  4. Compute weight change value by the error: $\Delta w=f(E)$
  5. Backpropagation: $w = w - \Delta w$
  6. Go to step 2

supervised learning



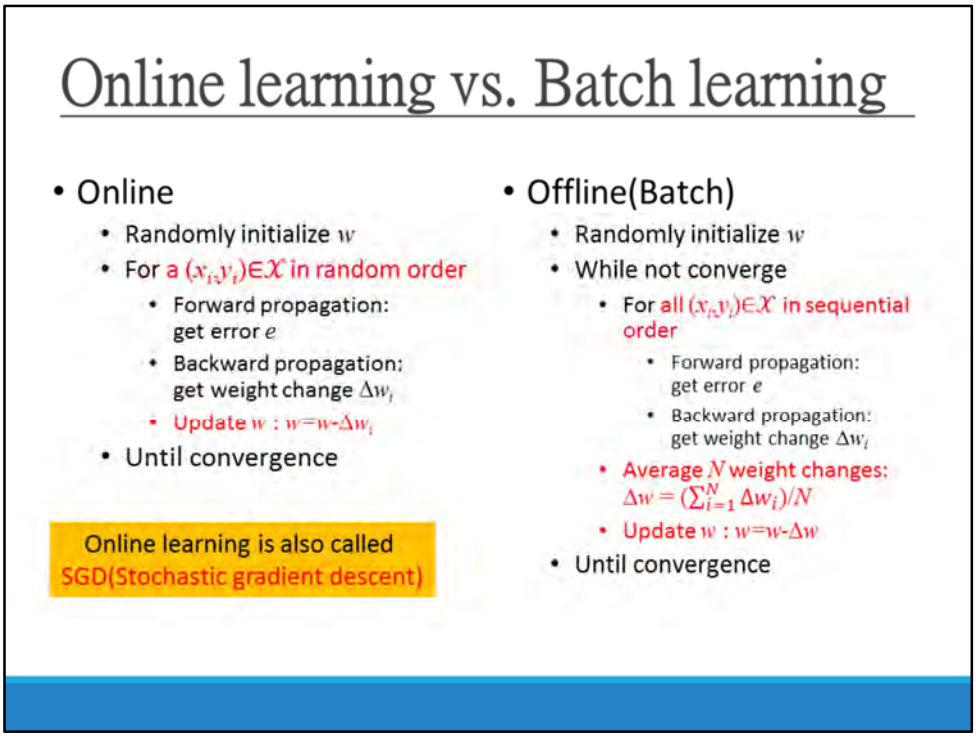
- Source of the example: Machine Learning - Neural Networks Tutorial, Paul Tero of Existor Ltd, 2015/8/20. http://www.existor.com/en/news-neural-networks.html.
- This learning method is called online learning (stochastic gradient descent).

Learning of MLP Network

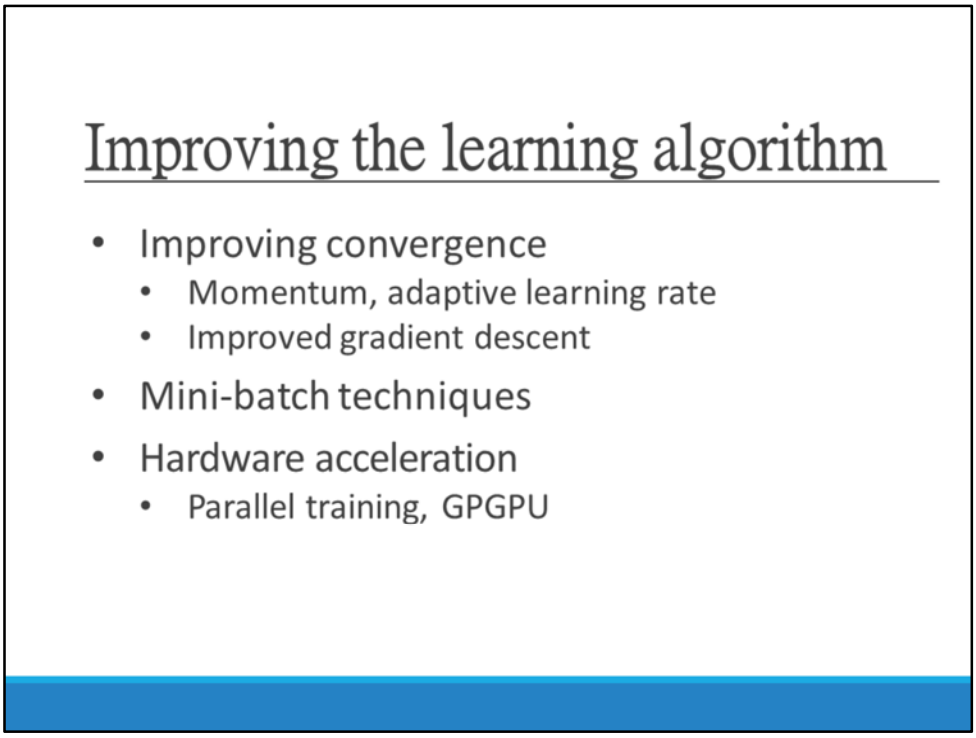
An example of backpropagation learning
Learning algorithms
Optimization and learning

The learning algorithm

- We just know how to train the MLP for "only one" learning sample: (x,y)
- How to train the MLP for a lot of learning samples, $\mathcal{X}=\{(x_1,y_1), (x_2,y_2), \dots , (x_N,y_N)\}$?
  - Online learning
  - Offline(Batch) learning



- Online learning: update MLP weights W for every learning samples
  - Also called SGD method
- Offline learning: update MLP weights W for all learning samples
  - Also called batch method
- Both methods have different characteristics. See neural network textbooks for details.
- When the learning loop will converge?
  - We need to calculate the total error of learning (or verification) samples.
  - Here we omit the details of convergence condition



- Goal of convergence improvement
  - Acceleration of convergence, lower error rate of learning
  - These two goals are contradictory
- There are many improved gradient descent algorithms: line search, bracket search, ...

Parallel training of neural nets

An active topic of research.

- No clear winner yet.

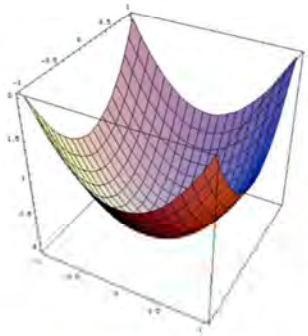
Baseline: lock-free stochastic gradient

- Assume shared memory
- Each processor access the weights through the shared memory
- Each processor runs SGD on different examples
- Read and writes to the weight memory are unsynchronized.
- Synchronization issues are just another kind noise...

Learning of MLP Network

An example of backpropagation learning
Learning algorithms
Optimization and learning

Convex



Definition

$\forall x, y, \forall 0 \leq \lambda \leq 1,$
$f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda) f(y)$

Property

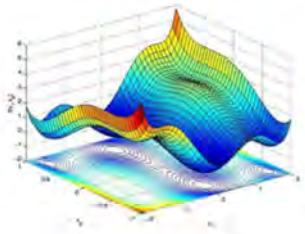
Any local minimum is a global minimum.

Conclusion

Optimization algorithms are easy to use.
They always return the same solution.

Example: Linear model with convex loss function.
- Curve fitting with mean squared error.
- Linear classification with log-loss or hinge loss.

Non-convex

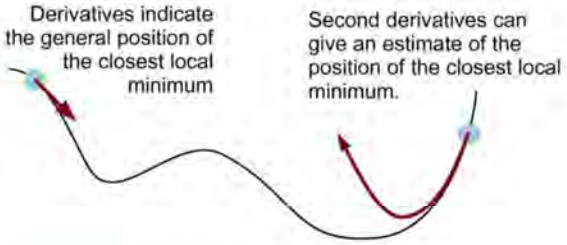


Landscape
- local minima, saddle points.
- plateaux, ravines, etc.

Optimization algorithms
- Usually find local minima.
- Good and bad local minima.
- Result depend on subtle details.

Examples
- Multilayer networks.
- Clustering algorithms.
- Learning features.
- Mixture models.
- Hidden Markov Models.
- Selecting features (some).

Derivatives



No such local cues without derivatives
- Derivatives may not exist.
- Derivatives may be too costly to compute.

Optimization vs. learning

Empirical cost

- Usually $f(w) = \frac{1}{n} \sum_{i=1}^{n} L(x_i, y_i, w)$
- The number n of training examples can be large (billions?)

Redundant examples

- Examples are redundant (otherwise there is nothing to learn.)
- Doubling the number of examples brings a little more information.
- Do we need it during the first optimization iterations?

Examples on-the-fly

- All examples may not be available simultaneously.
- Sometimes they come on the fly (e.g. web click stream.)
- In quantities that are too large to store or retrieve (e.g. click stream.)

Offline vs. online

Minimize $C(w) = \frac{\lambda}{2}\|w\|^2 + \frac{1}{n}\sum_{i=1}^{n} L(x_i, y_i, w)$.

Offline: process all examples together

– Example: minimization by gradient descent

Repeat: $w \leftarrow w - \gamma\left(\lambda w + \frac{1}{n}\sum_{i=1}^{n}\frac{\partial L}{\partial w}(x_i, y_i, w)\right)$

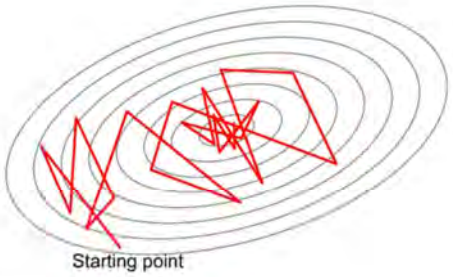
Offline: process examples one by one

– Example: minimization by stochastic gradient descent

Repeat: (a) Pick random example x_t, y_t

(b) $w \leftarrow w - \gamma_t\left(\lambda w + \frac{\partial L}{\partial w}(x_t, y_t, w)\right)$

Stochastic Gradient Descent



- Very noisy estimates of the gradient.
- Gain γ_t controls the size of the cloud.
- Decreasing gains $\gamma_t = \gamma_0(1 + \lambda\gamma_0 t)^{-1}$.
- Why is it attractive?