Artificial Neural Networks
Lect. 4
Backpropagation Algorithm

Dr. Mai Elshehaly
Department of Computer Science
Suez Canal University
A taxonomy of Artificial Neural Networks (ANNs):

Artificial Neural Networks (ANNs) differ in 3 main criteria:

1. The properties of the neuron or cell (activation function, threshold)
2. The architecture of the network (number of layers, topology)
3. The learning rule (weight calculation) and the way weights are updated (synchronous, continuous)
Learning for different architectures

Single –layer Perceptron → LMS

Multilayer Feedforward Network → Backpropagation
Q: When do we need a multilayer network?
Q: When do we need a multilayer network?

When the problem is NOT linearly separable
Famous Example of Linearly Inseperable Problems: XOR

No single line can separate between the classes

© MAI ELSHEHALY
Q: What is a good training algorithm for single-layer Perceptron?
Q: What is a good training algorithm for single-layer Perceptron?

Least Mean Square (LMS)

**TABLE 3.1 Summary of the LMS Algorithm**

<table>
<thead>
<tr>
<th>Training Sample:</th>
<th>Input signal vector = x(n)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Desired response = d(n)</td>
<td></td>
</tr>
</tbody>
</table>

User-selected parameter: \( \eta \)

Initialization. Set \( \hat{w}(0) = 0 \).

Computation. For \( n = 1, 2, \ldots \), compute

\[
e(n) = d(n) - \hat{w}^T(n)x(n)
\]

\[
\hat{w}(n + 1) = \hat{w}(n) + \eta x(n)e(n)
\]
Can we use the same LMS training that we used for Single-Layer Perceptron?
Can we use the same LMS training that we used for Single-Layer Perceptron? **NO**
Can we use the same LMS training that we used for Single-Layer Perceptron? **NO**

**In order to train the network, we need to:**

1. Define a cost function: \( \epsilon \rightarrow \text{we can call it } E \)

2. Calculate the change in cost that is caused by the change in weights \( W \):
   \[
   \frac{\partial E}{\partial w_{kj}^L}
   \]

3. Follow the steepest descent to have \( \epsilon(W(n + 1)) < \epsilon(W(n)) \)
Can we use the same LMS training that we used for Single-Layer Perceptron? **NO**

We must know the relationship between the cost and the weights (all of them!)

2. Calculate the change in cost that is caused by the change in weights $W$: 
   
   $$\frac{\partial E}{\partial w^L_{kj}}$$

3. Follow the steepest descent to have $\epsilon(W(n + 1)) < \epsilon(W(n))$
Backpropagation: General Idea

We define the cost function: $E = \sum_k (d_k - a_k)^2$

The goal is to find weights that will minimize $E$

There are two regions where weights can change:

1. For neurons in the **output layer**: define $\delta^L_j$
   
   change of cost due to weight change in the output layer

2. For neurons in **hidden layers**: we define $\delta^l_j$
   
   change of cost due to weight change in layer $l$
Backpropagation: General Idea

We define the cost function: \( E = \sum_k (d_k - a_k)^2 \)

The goal is to find weights that will minimize \( E \)

There are two regions where weights can change:

1. For neurons in the **output layer**: define \( \delta_j^L \)
   
   change of cost due to weight change in the output layer

2. For neurons in **hidden layers**: we define \( \delta_j^l \)
   
   change of cost due to weight change in layer \( l \)
Change in cost as a function of change in output layer: \[ \delta^L_j = \frac{\partial E}{\partial z^L_k} \]

Where,
\[ z^L_k = \sum_j (w^L_{kj} x^L_{j-1} + b^L_k) \]

**Q:** What is the relationship between \( E \) and \( z^L_k \) ?
Change in cost as a function of change in output layer: $\delta_j^L = \frac{\partial E}{\partial z_k^L}$

$$E = \sum_k (d_k - a_k)^2$$

Where $a_k$ is a function of $z_k^L$

We use the Chain Rule:

$$\delta_j^L = \frac{\partial E}{\partial z_k^L} = \frac{\partial E}{\partial a_k^L} \frac{\partial a_k^L}{\partial z_k^L}$$
Change in cost as a function of change in output layer: \( \delta_j^L = \frac{\partial E}{\partial z_k^L} \)

- **Change in cost as the output of kth neuron changes**
- **Error at kth neuron of output layer**
- **Change in output as net input of kth neuron changes**
Change in cost as a function of change in output layer: $\delta^L_j = \frac{\partial E}{\partial z^L_k}$

$\delta^L_k = \frac{\partial E}{\partial a^L_k} \sigma'(z^L_k)$

Since $E = \frac{1}{2} \sum_k (d_k - a_k)^2$

$\frac{\partial E}{\partial a^L_k} = (d^L_k - a^L_k)$

Hence, $\delta^L_k = (d^L_k - a^L_k) \sigma'(z^L_k)$  \hspace{1cm} (eq. 1)
Backpropagation: 
General Idea

We define the cost function: \( E = \sum_k (d_k - a_k)^2 \)

The goal is to find weights that will minimize \( E \)

There are two regions where weights can change:

1. For neurons in the **output layer**: define \( \delta_j^L \)
   
   *change of cost due to weight change in the output layer*

2. For neurons in **hidden layers**: we define \( \delta_j^l \)
   
   *change of cost due to weight change in layer l*
Change in cost as a function of change in hidden layer $l$: 

$$\delta^l_j = \frac{\partial E}{\partial z^l_k}$$

Where,

$$z^l_k = \sum_j (w^l_{kj} x^{l-1}_j + b^l_k)$$

**Q:** What is the relationship between $E$ and $z^l_k$?
Change in cost as a function of change in hidden layer $l$: $\delta_j^l = \frac{\partial E}{\partial z_k^l}$

We use the Chain Rule:

$$\delta_j^l = \frac{\partial E}{\partial z_k^l} = \frac{\partial E}{\partial z_{k}^{l+1}} \frac{\partial z_{k}^{l+1}}{\partial z_{k}^l}$$
Change in cost as a function of change in hidden layer: $\delta^l_j = \frac{\partial E}{\partial z^l_k}$

Change in cost as a function of change in layer $l+1$

Error at $k$th neuron of hidden layer $l$

Change in $l+1$th layer as change happens to layer $l$
Change in cost as a function of change in hidden layer: \( \delta^l_j = \frac{\partial E}{\partial z^l_k} \)
Change in cost as a function of change in hidden layer:

$$\delta^l_j = \frac{\partial E}{\partial z_k^l}$$

$$\delta_k^l = \frac{\partial E}{\partial z_k^{l+1}} \frac{\partial z_k^{l+1}}{\partial z_k^l}$$

$$= \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_k^l}$$

$$z_k^{l+1} = \sum_j (w_{kj}^{l+1} x_j^l + b_k^{l+1})$$

$$= \sum_j (w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1})$$

$$\frac{\partial z_k^{l+1}}{\partial z_k^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$
Change in cost as a function of change in hidden layer: 

$$\delta_j^l = \frac{\partial E}{\partial z_k^l}$$

$$\delta_k^l = \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_k^l}$$

$$\frac{\partial z_k^{l+1}}{\partial z_k^l} = \delta_k^{l+1} \frac{\partial z_k^{l+1}}{\partial z_k^l}$$

$$z_k^{l+1} = \sum_j (w_{kj}^{l+1} x_j^l + b_k^{l+1}) = \sum_j (w_{kj}^{l+1} \sigma(z_j^l) + b_k^{l+1})$$

$$\frac{\partial z_k^{l+1}}{\partial z_k^l} = w_{kj}^{l+1} \sigma'(z_j^l)$$

Substitute in Eq. 2
Change in cost as a function of change in hidden layer: \( \delta^l_j = \frac{\partial E}{\partial z^l_k} \)

\[
\delta^l_k = \delta^l_{k+1} \frac{\partial z^{l+1}_k}{\partial z^l_k} \\
= \delta^l_{k+1} w^{l+1}_{kj} \sigma'(z^l_j)
\]

Substitute in Eq. 2

\[
\delta^l_k = \frac{\partial E}{\partial z^{l+1}_k} \frac{\partial z^{l+1}_k}{\partial z^l_k} \\
= \delta^l_{k+1} \frac{\partial z^{l+1}_k}{\partial z^l_k} \\
= \delta^l_{k} \frac{\partial z^{l+1}_k}{\partial z^l_k} \\
\text{where} \\
\frac{\partial z^{l+1}_k}{\partial z^l_k} = w^{l+1}_{kj} \sigma'(z^l_j)
\]
Backpropagation:

General Idea

We define the cost function: \( E = \sum_k (d_k - a_k)^2 \)

The goal is to find weights that will minimize \( E \)

There are two regions where weights can change:

1. For neurons in the **output layer**: define \( \delta_j^L = (d_k^L - a_k^L)\sigma'(z_k^L) \)

   *change of cost due to weight change in the output layer*

2. For neurons in **hidden layers**: we define \( \delta_j^l = \delta_k^{l+1} w_{kj}^{l+1} \sigma'(z_j^l) \)

   *change of cost due to weight change in layer \( l \)*
Updating weights and bias:

Using $\delta^l_j$ we can define the change in cost as a function of weights on a given layer as:

$$\frac{\partial E}{\partial w^l_{k,j}} = \delta^l_j a^{l-1}_k$$

We can also define the change in cost as a function of biases on a given layer as:

$$\frac{\partial E}{\partial b^l_j} = \delta^l_j$$
Backpropagation Algorithm

Start with randomly chosen weights $[w_{jk}^l]$

While error is unsatisfactory:

  for each input pattern $x$:

    feedforward: for each $l = 1, 2, \ldots, L$ compute $z_k^l$ and $a_k^l$

    Compute the error at output layer: $\delta_k^L = (d_k^L - a_k^L)\sigma'(z_k^L)$

    Backpropagate the error: for $l = L-1, L-2, \ldots 2$

      compute $\delta_k^l = \delta_k^{l+1}w_{kj}^{l+1}\sigma'(z_j^l)$

    Calculate the gradients: $\frac{\partial E}{\partial w_{kj}^l} = \delta_j^l a_k^{l-1}$ and $\frac{\partial E}{\partial b_j^l} = \delta_j^l$

  end for

end while
Defining the Network class:

class Network(object):
    def __init__(self, sizes):
        self.num_layers = len(sizes)
        self.sizes = sizes
        self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
        self.weights = [np.random.randn(y, x)
                        for x, y in zip(sizes[:-1], sizes[1:])]
$\text{net} = \text{Network}([3, 2, 1])$

Biases:

$\begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}$ and $[b_1^2]$

Weights:
def __init__(self, sizes):
    """The list `sizes` contains the number of neurons in the respective layers of the network. For example, if the list was [2, 3, 1] then it would be a three-layer network, with the first layer containing 2 neurons, the second layer 3 neurons, and the third layer 1 neuron. The biases and weights for the network are initialized randomly, using a Gaussian distribution with mean 0, and variance 1. Note that the first layer is assumed to be an input layer, and by convention we won't set any biases for those neurons, since biases are only ever used in computing the outputs from later layers."""
    self.num_layers = len(sizes)
    self.sizes = sizes
    self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
    self.weights = [np.random.randn(y, x)
                    for x, y in zip(sizes[:-1], sizes[1:])]  
    print(self.biases)
    print(self.weights)

Console Output:

C:\Users\Mai\Anaconda3\python.exe "C:\Users\Mai\Dropbox\Teaching\Neural Networks\neural- 
array([[-0.07255204,
       [-0.87602306]], array([[-1.39621772]]))
[ array([[0.49732565, -1.34532419, -1.64812572],
       [1.09648866, -1.00056592, 0.39153363]], array([[2.79351196, -0.03665709]]))

Process finished with exit code 0
What is NumPy and NumPy Array?

NumPy is:

- An extension package to Python for multi-dimensional arrays
- Designed for scientific computation
- Also known as array oriented computing

NumPy `array()` object:

```python
>>> b = np.array([[0, 1, 2], [3, 4, 5]])  # 2 x 3 array
>>> b
array([[0, 1, 2],
       [3, 4, 5]])
>>> b.ndim
2
>>> b.shape
(2, 3)
>>> len(b)  # returns the size of the first dimension
2
```
net.weights[0] ➔ the set of weights connecting the input layer with the first hidden layer
net.weights[1] ➔ the set of weights connecting the first hidden layer with the second hidden layer
```python
def feedforward(self, a):
    """Return the output of the network if `a` is input."""
    for b, w in zip(self.biases, self.weights):
        a = sigmoid(np.dot(w, a) + b)
    return a

def sgd(self, training_data, epochs, mini_batch_size, eta, test_data=None):
    """Train the neural network using mini-batch stochastic gradient descent. The `training_data` is a list of tuples `((x, y)` representing the training inputs and the desired outputs. The other non-optional parameters are self-explanatory. If `test_data` is provided then the network will be evaluated against the test data after each epoch, and partial progress printed out. This is useful for tracking progress, but slows things down substantially."""
    if test_data:
        n_test = len(test_data)
    n = len(training_data)
    for j in range(epochs):
        random.shuffle(training_data)
        mini_batches = [training_data[k:k+mini_batch_size]
                        for k in range(0, n, mini_batch_size)]
        for mini_batch in mini_batches:
            self.update_mini_batch(mini_batch, eta)```
Getting the code and data:


In Console: load the data

```python
>>> import mnist_loader

>>> training_data, validation_data, test_data = \
... mnist_loader.load_data_wrapper()
```
>>> import network

>>> net = network.Network([784, 30, 10])

>>> net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
Convergence

Saturation due to sigma’ (Michael’s book).
HW 3:

Q1: When does the learning become very slow or stops?

Q2: Consider the following Neural Network:
HW3: (Q2 – continued)

Initial weights and biases:

\[ w^1 = 1 , \quad b^1 = 1 , \quad w^2 = -2 , \quad b^2 = 1 . \]

An input/target pair is given to be

\[ ((p = 1), (t = 1)) . \]

i. Find the squared error \( (e)^2 \) as an explicit function of all weights and biases.

ii. Using part (i) find \( \partial (e)^2 / \partial w^1 \) at the initial weights and biases.

iii. Repeat part (ii) using backpropagation and compare results.