Backpropagation[*]

March 13, 2025

Contents

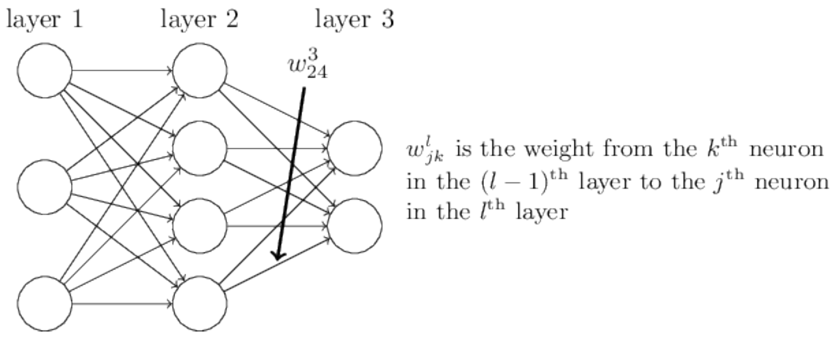
[*]This lecture notes were written by Nail Ibrahimli and were heavily inspired by and adapted from:
- Mickael Nielsen. Neural Networks and Deep Learning

1 Notations and Preliminaries

To describe backpropagation we adopt the following notation:

- w_{jk}^l denotes the weight for the connection from the k-th neuron in the $(l-1)$-th layer to the j-th neuron in the l-th layer.
- b_j^l is the bias for the j-th neuron in the l-th layer.
- a_j^l is the activation of the j-th neuron in the l-th layer.

For example, the diagram below shows the weight from the fourth neuron in the second layer to the second neuron in the third layer:


layer 1
layer 2
layer 3
w_{24}^3
w_{jk}^l is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

In vectorized form we define for each layer l:

- A weight matrix w^l, where the entry in the j-th row and k-th column is w_{jk}^l.
- A bias vector b^l, with components b_j^l.

The activation of the j-th neuron in layer l is given by:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right), \tag{1}$$

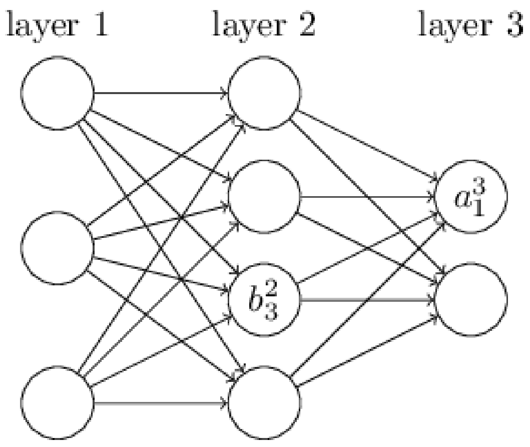
or, in compact vectorized notation,

$$a^l = \sigma\big(w^l a^{l-1} + b^l\big). \tag{2}$$

Here, σ is the activation function (typically the sigmoid), and we define the **weighted input** to layer l as:

$$z^l \equiv w^l a^{l-1} + b^l,$$

so that $a^l = \sigma(z^l)$.



2 Cost Function

Our goal is to compute the partial derivatives $\partial C/\partial w$ and $\partial C/\partial b$ for a given cost function C, so that we can update the network parameters during training. For simplicity, we consider the quadratic cost function:

$$C = \frac{1}{2n}\sum_x \left\|y(x) - a^L(x)\right\|^2, \tag{3}$$

where:

- n is the number of training examples,
- x denotes an individual training example,
- $y(x)$ is the desired output for x,
- L is the total number of layers,
- $a^L(x)$ is the network's output for x.

Again, the aim of backpropagation is to compute the partial derivatives $\partial C/\partial w$ and $\partial C/\partial b$ of the cost function C with respect to any weight w or bias b in the network. For backpropagation to work we need to make two main assumptions about the form of the cost function.

1) It can be written as an average over training examples, i.e., $C = \frac{1}{n}\sum_x C_x$, where C_x is the cost for a single example. (i.e. dependency on input)

2) It is a function of the output activations only (with the desired output y treated as a fixed parameter). For the quadratic cost,

$$C_x = \frac{1}{2}\sum_j \left(y_j - a_j^L\right)^2, \tag{4}$$

which depends solely on a^L. (i.e. we need to tune the weights of the model such that the output layer is steered towards to yjr desired output)

3 The Fundamental Equations of Backpropagation

Backpropagation is designed to efficiently compute the gradients of the cost function with respect to every weight and bias in the network. To do so, we introduce an **error** term for each neuron. For the j-th neuron in layer l, define:

$$\delta_j^l \equiv \frac{\partial C}{\partial z_j^l}. \tag{5}$$

This quantity measures how a small change in the weighted input z_j^l affects the overall cost.

The following four equations form the backbone of the backpropagation algorithm:

1. Error in the Output Layer (BP1)

For the output layer ($l = L$), the error is given by:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L}\, \sigma'(z_j^L). \tag{BP1}$$

In vectorized form, using the Hadamard (elementwise) product $\odot$, we write:

$$\delta^L = \nabla_a C \odot \sigma'(z^L). \tag{BP1a}$$

For example, when using the quadratic cost function we have $\nabla_a C = (a^L - y)$, so that:

$$\delta^L = (a^L - y) \odot \sigma'(z^L). \tag{6}$$

2. Error Backpropagation (BP2)

For any layer l (with $2 \leq l < L$), the error is propagated backward from the layer ahead:

$$\delta^l = \Big((w^{l+1})^T\, \delta^{l+1}\Big) \odot \sigma'(z^l). \tag{BP2}$$

Here, $(w^{l+1})^T$ is the transpose of the weight matrix for the next layer. This equation shows how the error “moves backward” through the network.

3. Partial Derivative with Respect to Biases (BP3)

The rate of change of the cost with respect to a bias is given by:

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l. \tag{BP3}$$

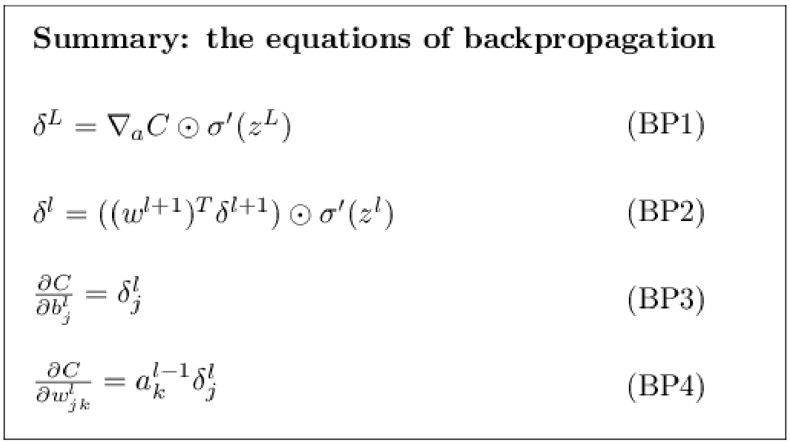
In shorthand, one can write:

$$\frac{\partial C}{\partial b} = \delta.$$

4. Partial Derivative with Respect to Weights (BP4)

For a weight connecting the k-th neuron in layer $(l-1)$ to the j-th neuron in layer l, we have:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\, \delta_j^l. \tag{BP4}$$



Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{BP3}$$

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{BP4}$$

4 The Backpropagation Algorithm

The backpropagation algorithm computes the gradient of the cost function with respect to the network parameters for a single training example. The process can be summarized in the following steps:

1) **Input:** Set the input activation $a^1 = x$.

2) **Feedforward:** For each layer $l = 2, 3, \ldots, L$, compute

$$z^l = w^l\, a^{l-1} + b^l \quad \text{and} \quad a^l = \sigma(z^l).$$

3) **Output Error:** Compute the error in the output layer:

$$\delta^L = \nabla_a C \odot \sigma'(z^L).$$

4) **Backpropagate the Error:** For each layer $l = L-1, L-2, \ldots, 2$, compute

$$\delta^l = \Big((w^{l+1})^T\, \delta^{l+1}\Big) \odot \sigma'(z^l).$$

5) **Gradient Computation:** The gradients for the cost function are given by

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k\, \delta^l_j \quad \text{and} \quad \frac{\partial C}{\partial b^l_j} = \delta^l_j.$$

When training with a mini-batch of m training examples, the above procedure is applied to each example, and the gradients are averaged before updating the weights and biases.

Mini-Batch Gradient Descent Algorithm

1) **Input:** A mini-batch of m training examples.

2) **For each training example x:**

- Set input activation $a^{x,1} = x$.
- Feedforward: Compute $z^{x,l} = w^l\, a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$ for $l = 2, \ldots, L$.
- Compute output error: $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.
- Backpropagate the error for $l = L-1, L-2, \ldots, 2$:

$$\delta^{x,l} = \Big((w^{l+1})^T\, \delta^{x,l+1}\Big) \odot \sigma'(z^{x,l}).$$

3) **Update:** For each layer l, update the parameters by averaging over the mini-batch:

$$w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l}\, (a^{x,l-1})^T, \quad b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}.$$

5 Code Implementation of Backpropagation

The following Python code is a direct translation of the algorithm described above. It shows how to update the network's weights and biases for a mini-batch using backpropagation.

Updating with a Mini-Batch

```
class Network(object):
...
  def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch
   .
    The "mini_batch" is a list of tuples "(x, y)", and "eta"
    is the learning rate."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
      delta_nabla_b, delta_nabla_w = self.backprop(x, y)
      nabla_b = [nb+dnb for nb, dnb in zip(nabla_b, delta_nabla_b
   )]
      nabla_w = [nw+dnw for nw, dnw in zip(nabla_w, delta_nabla_w
   )]
    self.weights = [w-(eta/len(mini_batch))*nw
            for w, nw in zip(self.weights, nabla_w)]
    self.biases = [b-(eta/len(mini_batch))*nb
            for b, nb in zip(self.biases, nabla_b)]
```

Backpropagation Method

```
class Network(object):
...
  def backprop(self, x, y):
    """Return a tuple "(nabla_b, nabla_w)" representing the
    gradient for the cost function C_x.  "nabla_b" and
    "nabla_w" are layer-by-layer lists of numpy arrays, similar
    to "self.biases" and "self.weights"."""
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # feedforward
    activation = x
    activations = [x] # list to store all the activations, layer
   by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
      z = np.dot(w, activation)+b
      zs.append(z)
      activation = sigmoid(z)
      activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) *
   sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-2].transpose())
    # Note that the variable l in the loop below is used a little
    # differently to the notation in the notes.  Here,
    # l = 1 means the last layer of neurons, l = 2 is the
    # second-last layer, and so on.  It's a renumbering of the
    # scheme in the notes, used here to take advantage of the
   fact
    # that Python can use negative indices in lists.
    for l in xrange(2, self.num_layers):
      z = zs[-l]
      sp = sigmoid_prime(z)
      delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
      nabla_b[-l] = delta
      nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
...
  def cost_derivative(self, output_activations, y):
    """Return the vector of partial derivatives \partial{} C_x /
    \partial{} a for the output activations."""
    return (output_activations-y)
  def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))
  def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```