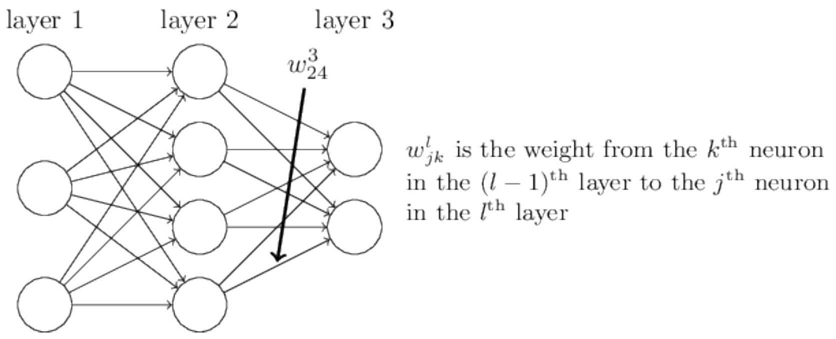
Backpropagation[*]

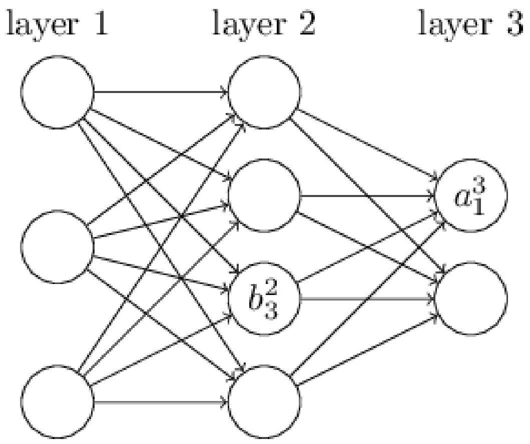
March 21, 2024

1 Notations

We'll use w^l_{jk} to denote 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. So, for example, the diagram below shows the weight on a connection from the fourth neuron in the second layer to the second neuron in the third layer of a network:


layer 1
layer 2
layer 3
w^3_{24}
w^l_{jk} 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

We use a similar notation for the network's biases and activations. Explicitly, we use b^l_j for the bias of the j-th neuron in the l-th layer. And we use a^l_j for the activation of the j-th neuron in the l-th layer. The following diagram shows examples of these notations in use:



[*]References
- Mickael Nielsen. Neural Networks and Deep Learning

With these notations, the activation a^l_j of the j-th neuron in the l-th layer is related to the activations in the $(l-1)$-th layer by the equation

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right), \tag{1}$$

where the sum is over all neurons k in the $(l-1)$-th layer. To rewrite this expression in a matrix form we define a *weight matrix* w^l for each layer, l. The entries of the weight matrix w^l are just the weights connecting to the l-th layer of neurons, that is, the entry in the j-th row and k-th column is w^l_{jk}. Similarly, for each layer l we define a *bias vector*, b^l. You can probably guess how this works – the components of the bias vector are just the values b^l_j, one component for each neuron in the l-th layer. With these notations in mind, Equation (1) can be rewritten in the beautiful and compact vectorized form

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

This expression gives us a much more global way of thinking about how the activations in one layer relate to activations in the previous layer: we just apply the weight matrix to the activations, then add the bias vector, and finally apply the σ function.

When using Equation (2) to compute a^l, we compute the intermediate quantity $z^l \equiv w^l a^{l-1} + b^l$ along the way. This quantity turns out to be useful enough to be worth naming: we call z^l the weighted input to the neurons in layer l. We'll make considerable use of the weighted input z^l later in the lecture notes. Equation (2) is sometimes written in terms of the weighted input, as $a^l = \sigma(z^l)$. It's also worth noting that z^l has components $z^l_j = \sum_k w^l_{jk} a^{l-1}_k + b^l_j$, that is, z^l_j is just the weighted input to the activation function for neuron j in layer l.

2 Cost function

The goal 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. Before stating those assumptions, though, it's useful to have an example cost function in mind. We'll use the quadratic cost function for simplicity.

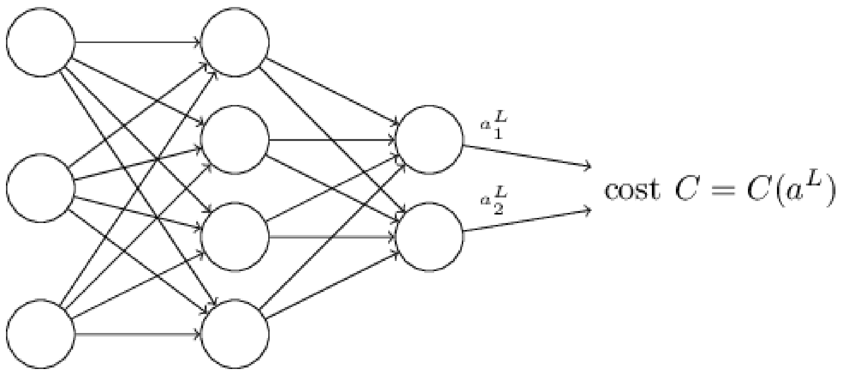
$$C = \frac{1}{2n} \sum_x \left\|y(x) - a^L(x)\right\|^2, \tag{3}$$

where: n is the total number of training examples; the sum is over individual training examples, x; $y = y(x)$ is the corresponding desired output; L denotes the number of layers in the network; and $a^L = a^L(x)$ is the vector of activations output from the network when x is input.

The first assumption we need is that the cost function can be written as an average $C = \frac{1}{n}\sum_x C_x$ over cost functions C_x for individual training examples, x. This is the case for the quadratic cost function, where the cost for a single training example is $C_x = \frac{1}{2}\|y - a^L\|^2$. This assumption will also hold true for all the other cost functions we'll meet in this notes.

The reason we need this assumption is because what backpropagation actually lets us do is compute the partial derivatives $\partial C_x/\partial w$ and $\partial C_x/\partial b$ for a single training example. We then recover $\partial C/\partial w$ and $\partial C/\partial b$ by averaging over training examples.

The second assumption we make about the cost is that it can be written as a function of the outputs from the neural network:



For example, the quadratic cost function satisfies this requirement, since the quadratic cost for a single training example x may be written as

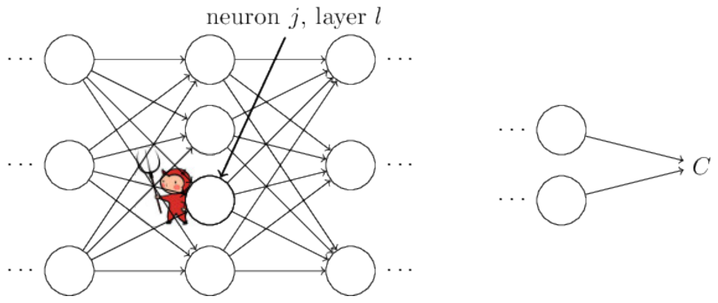
$$C = \frac{1}{2}\left\|y - a^L\right\|^2 = \frac{1}{2}\sum_j \left(y_j - a_j^L\right)^2, \tag{4}$$

and thus is a function of the output activations. Of course, this cost function also depends on the desired output y, and you may wonder why we're not regarding the cost also as a function of y. Remember, though, that the input training example x is fixed, and so the output y is also a fixed parameter. In particular, it's not something we can modify by changing the weights and biases in any way, i.e., it's not something which the neural network learns. And so it makes sense to regard C as a function of the output activations a^L alone, with y merely a parameter that helps define that function.

3 The four fundamental equations behind backpropagation

Backpropagation is about understanding how changing the weights and biases in a network changes the cost function. Ultimately, this means computing the partial derivatives $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$. But to compute those, we first introduce an intermediate quantity, δ_j^l, which we call the *error* in the j-th neuron in the l-th layer. Backpropagation will give us a procedure to compute the error δ_j^l, and then will relate δ_j^l to $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$.

To understand how the error is defined, imagine there is a demon in our neural network:



The demon sits at the j-th neuron in layer l. As the input to the neuron comes in, the demon messes with the neuron's operation. It adds a little change Δz_j^l to the neuron's weighted input, so that instead of outputting $\sigma(z_j^l)$, the neuron instead outputs $\sigma(z_j^l + \Delta z_j^l)$. This change propagates through later layers in the network, finally causing the overall cost to change by an amount $\frac{\partial C}{\partial z_j^l}\Delta z_j^l$.

Now, this demon is a good demon, and is trying to help you improve the cost, i.e., they're trying to find a Δz_j^l which makes the cost smaller. Suppose $\partial C/\partial z_j^l$ has a large value (either positive or negative). Then the demon can lower the cost quite a bit by choosing Δz_j^l to have the opposite sign to $\partial C/\partial z_j^l$. By contrast, if $\partial C/\partial z_j^l$ is close to zero, then the demon can't improve the cost much at all by perturbing the weighted input z_j^l. So far as the demon can tell, the neuron is already pretty near optimal[1]. And so there's a heuristic sense in which $\partial C/\partial z_j^l$ is a measure of the error in the neuron.

Motivated by this story, we define the error δ_j^l of neuron j in layer l by

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

As per our usual conventions, we use δ^l to denote the vector of errors associated with layer l. Backpropagation will give us a way of computing δ^l for every layer, and then relating those errors to the quantities of real interest, $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$.

Here's a preview of the ways we'll delve more deeply into the equations later in the lecture notes: I'll give a short proof of the equations, which helps explain why they are true; we'll restate the equations in algorithmic form as pseudocode, and see how the pseudocode can be implemented as real, running Python code; and, in the final section of the lecture notes, we'll develop an intuitive picture of what the backpropagation equations mean, and how someone might discover them from scratch. Along the way we'll return repeatedly to the four fundamental equations, and as you deepen your understanding those equations will come to seem comfortable and, perhaps, even beautiful and natural.

An equation for the error in the output layer, δ^L: The components of δ^L are given by

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

This is a very natural expression. The first term on the right, $\partial C/\partial a_j^L$, just measures how fast the cost is changing as a function of the j-th output activation. If, for example, C doesn't depend much on a particular output neuron, j, then δ_j^L will be small, which is what we'd expect. The second term on the right, $\sigma'(z_j^L)$, measures how fast the activation function σ is changing at z_j^L.

Notice that everything in Eq. (BP1) is easily computed. In particular, we compute z_j^L while computing the behaviour of the network, and it's only a small additional overhead to compute $\sigma'(z_j^L)$. The exact form of $\partial C/\partial a_j^L$ will, of course, depend on the form of the cost function. However, provided the cost function is known there should be little trouble computing $\partial C/\partial a_j^L$. For example, if we're using the quadratic cost function then $C = \frac{1}{2}\sum_j (y_j - a_j^L)^2$, and so $\partial C/\partial a_j^L = (a_j^L - y_j)$, which obviously is easily computable.

Equation (BP1) is a componentwise expression for δ^L. It's a perfectly good expression, but not the matrix-based form we want for backpropagation. However, it's easy to rewrite

[1]This is only the case for small changes Δz_j^l, of course. We'll assume that the demon is constrained to make such small changes.

the equation in a matrix-based form, as

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

Here, $\nabla_a C$ is defined to be a vector whose components are the partial derivatives $\partial C/\partial a^L_j$. You can think of $\nabla_a C$ as expressing the rate of change of C with respect to the output activations. $\odot$ is elementwise product (Hadamard product) operation. It's easy to see that Equations (BP1a) and (BP1) are equivalent, and for that reason from now on we'll use (BP1) interchangeably to refer to both equations. As an example, in the case of the quadratic cost we have $\nabla_a C = (a^L - y)$, and so the fully matrix-based form of (BP1) becomes

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

As you can see, everything in this expression has a nice vector form, and is easily computed using a library such as Numpy.

An equation for the error δ^l in terms of the error in the next layer, δ^{l+1}: In particular

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

where $(w^{l+1})^T$ is the transpose of the weight matrix w^{l+1} for the (l+1)-th layer. This equation appears complicated, but each element has a nice interpretation. Suppose we know the error δ^{l+1} at the (l+1)-th layer. When we apply the transpose weight matrix, $(w^{l+1})^T$, we can think intuitively of this as moving the error *backward* through the network, giving us some sort of measure of the error at the output of the l-th layer. We then take the Hadamard product $\odot\sigma'(z^l)$. This moves the error backward through the activation function in layer l, giving us the error δ^l in the weighted input to layer l.

By combining (BP2) with (BP1) we can compute the error δ^l for any layer in the network. We start by using (BP1) to compute δ^L, then apply Equation (BP2) to compute δ^{L-1}, then Equation (BP2) again to compute δ^{L-2}, and so on, all the way back through the network.

An equation for the rate of change of the cost with respect to any bias in the network: In particular:

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j. \tag{BP3}$$

That is, the error δ^l_j is *exactly equal* to the rate of change $\partial C/\partial b^l_j$. This is great news, since (BP1) and (BP2) have already told us how to compute δ^l_j. We can rewrite (BP3) in shorthand as

$$\frac{\partial C}{\partial b} = \delta, \tag{7}$$

where it is understood that δ is being evaluated at the same neuron as the bias b.

An equation for the rate of change of the cost with respect to any weight in the network: In particular:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j. \tag{BP4}$$

This tells us how to compute the partial derivatives $\partial C/\partial w^l_{jk}$ in terms of the quantities δ^l and a^{l-1}, which we already know how to compute.

Summary: the equations of backpropagation

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

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

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \quad \text{(BP3)}$$

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \quad \text{(BP4)}$$

4 The backpropagation algorithm

The backpropagation equations provide us with a way of computing the gradient of the cost function. Let's explicitly write this out in the form of an algorithm:

1) **Input** x: Set the corresponding activation a^1 for the input layer.
2) **Feedforward:** For each $l = 2, 3, \dots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z_l)$.
3) **Output error** δ^L: Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.
4) **Backpropagate the error**: For each $l = L-1, L-2, \dots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.
5) **Output**: The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

As I've described it above, the backpropagation algorithm computes the gradient of the cost function for a single training example, $C = C_x$. In practice, it's common to combine backpropagation with a learning algorithm such as stochastic gradient descent, in which we compute the gradient for many training examples. In particular, given a mini-batch of m training examples, the following algorithm applies a gradient descent learning step based on that mini-batch:

1) Input a set of training examples
2) For each training example x: Set the corresponding input activation $a^{x,1}$, and perform the following steps:
   - Feedforward: For each l$= 2, 3, \dots, L$ compute $z^{x,l} = w^l a^{x,l-1} + b^l$ and $a^{x,l} = \sigma(z^{x,l})$.
   - Output error $\delta^{x,L}$: Compute the vector $\delta^{x,L} = \nabla_a C_x \odot \sigma'(z^{x,L})$.
   - Backpropagate the error: For each $l = L-1, L-2, \dots, 2$ compute $\delta^{x,l} = ((w^{l+1})^T \delta^{x,l+1}) \odot \sigma'(z^{x,l})$.
3) Gradient descent: For each $l = L, L-1, \dots, 2$ update the weights according to the rule $w^l \to w^l - \frac{\eta}{m} \sum_x \delta^{x,l} (a^{x,l-1})^T$, and the biases according to the rule $b^l \to b^l - \frac{\eta}{m} \sum_x \delta^{x,l}$.

Of course, to implement stochastic gradient descent in practice you also need an outer loop generating mini-batches of training examples, and an outer loop stepping through multiple epochs of training. I've omitted those for simplicity.

5 The code for backpropagation

The code for these methods is a direct translation of the algorithm described above. In particular, the **update_mini_batch** method updates the **Network**'s weights and biases by computing the gradient for the current **mini batch** of training examples:

```
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)]
```

Most of the work is done by the line **delta_nabla_b, delta_nabla_w = self.backprop(x, y)** which uses the **backprop** method to figure out the partial derivatives $\partial C_x/\partial b^l_j$ and $\partial C_x/\partial w^l_{jk}$. The **backprop** method follows the algorithm in the last section closely. There is one small change – we use a slightly different approach to indexing the layers. This change is made to take advantage of a feature of Python, namely the use of negative list indices to count backward from the end of a list, so, e.g., **l[-3]** is the third last entry in a list **l**. The code for **backprop** is below, together with a few helper functions, which are used to compute the σ function, the derivative σ', and the derivative of the cost function. With these inclusions you should be able to understand the code in a self-contained way. If something's tripping you up, you may find it helpful to consult the original description (and complete listing) of the code.

```
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))
```