chapter 2: how the backpropagation algorithm works

- backpropagation is a fast algorithm for computing the gradients of the cost function.
- The expression tells us how quickly the cost changes when we change the weights and biases. It actually gives us detailed insights into how changing the weights and biases changes the overall behavior of the network.
- $z_l = w_l * a_{l-1} + b$
- $a_l = \sigma_(w_l * a_{l-1} + b)$.
- We'll need **two assumptions** to be true for backpropagation to work:
  - The first assumption we need is that the cost function can be written as an average over cost functions for individual training examples. The reason we need this assumption is because what backpropagation actually lets us do is compute the partial derivatives $\frac{\partial C_x}{\partial w}$ and $\frac{\partial C_x}{\partial b}$ for a single training example. We then recover $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ 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
- **Hadamard product** or Schur product of two vectors is simply the element wise multiplication of the two vectors. Good matrix libraries usually provide fast implementations of the Hadamard product. we'll represent the Hadamard product as $A \odot B$
- Backpropagation is really just the just disciplined application of the multivariate chain rule to compute the rate of change of cost with each weight in the network.
- Note that we ultimately want to compute $\frac{dC}{dw}$, $\frac{dC}{db}$ for each weight and bias in the network.
- the equations:
  - $\delta_j^l = \frac{\partial C}{\partial z_j^l}$ (we're just defining δ as rate of change of cost with respect to z)
  - For the final layer L, $\frac{\partial C}{\partial z_j^L} = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$ (this is simply the chain rule applied to the cost function, keeping in mind the output of the last layer of the network)
  - Using Hadamard product, we can write the above as:
    - $\delta^L = \nabla_a C \odot \sigma'(z^L)$. **(1)**

    Here, $\nabla_a C$ is defined to be a vector whose components are the partial derivatives ∂C/∂a. You can think of it as expressing the rate of change of C with respect to the output layer activations.
    - For the least squares loss cost function, $\delta^L = (a^L - y) \odot \sigma'(z^L)$
    - $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$. **(2)**

    this just the application of the multivariate chain rule.
    - $\frac{\partial C}{\partial b_j^l} = \delta_j^l$ **(3)**
    - $\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$, **(4)**

    This just means that $\frac{\partial C}{\partial w} = a_{in}\delta_{out}$, where it's understood that a_{in} is the activation of the neuron input to the weight w, and δ_{out} is the error of the neuron output from the weight w.
- Note that the above equations don't depend on the exact form of the cost function. The cost just needs to be a function of the outputs of the neural network.
- The backprop algorithm:
  - get training data.
  - for each learning iteration:

    - make mini-batches
    - for each mini batch:
        - feedforward on each training example
        - compute δ^L for the output layer, using equation (2).
        - apply the backprop equations to compute the $\frac{\partial C}{\partial w}$ for each weight in the previous layer using equation (3). note that this doesn't actually have to be done for each weight in a layer => this is just matrix multiplication so linear algebra libraries and GPUs to the rescue! we'll discuss this later in this chapter. this is the step that propagates the error backward.
        - once we have the derivatives, we just take the average across all samples in this mini-batch. update the network weights accordingly, and move onto the next mini-batch.
- It's possible to modify the backpropagation algorithm so that it computes the gradients for all training examples in a mini-batch simultaneously. The advantage of this approach is that it takes full advantage of modern libraries for linear algebra. As a result it can be quite a bit faster than looping over the mini-batch. Using a GPU, the benefits of this approach are quite apparent, but even without GPUs, good linear algebra libraries have special optimizations for matrix multiplication and this takes advantage of those optimizations.
- We could have estimated $\frac{\partial C}{\partial w}$ by slightly incrementing each weight, computing the resulting cost and then using the basic equation of derivatives to estimate $\frac{\partial C}{\partial w}$. If we have a network with a million weights (which by the way isn't that crazy - just take a net with 1000 neurons in the first layer, and a similar number of them in the hidden layer which are then connected to the output layer and that already exceeds a million weights), we'd have to compute this estimate a million different times (once for each weight). That is pretty ridiculous compared to our backprop approach which inolves a feedforward step and propagating-the-error-back step (which is about just as expensive as the feedforward step).
- What's clever about backpropagation is that it enables us to simultaneously compute *all* the partial derivatives ∂C/∂w using just one forward pass through the network, followed by one backward pass through the network. Roughly speaking, the computational cost of the backward pass is about the same as the forward pass. And so the total cost of backpropagation is roughly the same as making just two forward passes through the network. Compare that to the million and one forward passes we needed for the other approach of estimating these partial derivatives.
- This speedup was first fully appreciated in 1986, and it greatly expanded the range of problems that neural networks could solve.
- Look back at the equations above - they are actually quite intuitive, if you think about the intuitions behind what they are saying/why they are true.