CSci 4270 and 6270
Computational Vision, Spring 2021
Lecture 15: The Basics of Neural Networks
March 18, 2021

Overview

Lectures are based on the first two chapters of the on-line "book"

`neuralnetworksanddeeplearning.com`

last accessed March 2021.

- Motivation
- Artificial neurons
- Activation functions
- Network architectures
- Neural network computations
- Learning and backpropagation

Motivation

- How would you describe recognition of digits to a novice?
  - Most people would start with a set of rules, but...
  - Hard to program, and many exceptions.
- What about extracting a description of a region for pedestrian detection?
  - Combination of standard techniques and experience-based intuitions about methods that might work
  - Implementation and parameterization of options
  - Large data set and extensive, controlled experiment
- Artificial neural networks instead:
  - Simple computational units
  - Operating in parallel
  - Connected in layers
  - Trained using optimization over massive data sets.

A Single Artificial Neuron

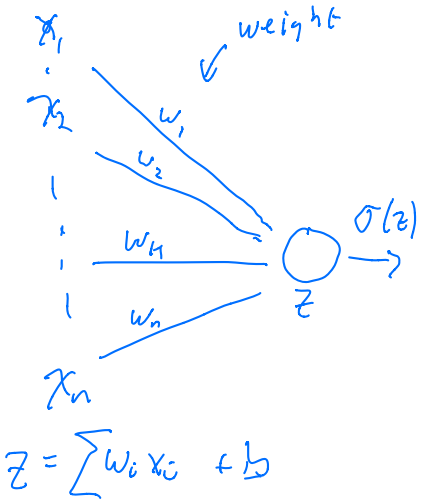
- Input values x_i, $i = 1, \ldots n$, formed into vector $\mathbf{x}$
- Weight values w_i, $i = 1, \ldots n$, formed into vector $\mathbf{w}$
- Bias value, b
- Combined input to the neuron:

$$z = \mathbf{w}^\top \mathbf{x} + b = \mathbf{w} \cdot \mathbf{x} + b = \sum_{i=1}^{n} w_i x_i + b$$

- Output from the neuron:

$$\sigma(z) = \sigma(\mathbf{w}^\top \mathbf{x} + b)$$

- We'll draw a simple picture in class to illustrate.



Activation Functions

- The first is a binary activation function:

$$\sigma(z) = \begin{cases} 1 & z \geq 0 \\ 0 & z < 0 \end{cases}$$

  The result is called a *perceptron* — one of the oldest types of artificial neuron.

  - Note the mathematical similarity to the decision rule of a linear SVM.
  - We will not use it because it is not differentiable — an important property for training ("learning") a network.

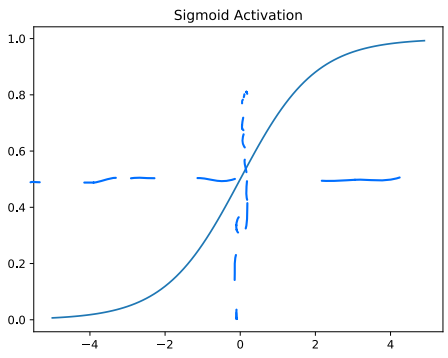
- Sigmoid activation:

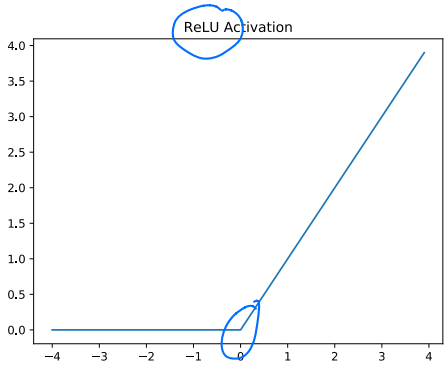
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

  which is 0.5 at $z = 0$, goes to 0 for large negative z, and goes to 1 for large positive z.

- ReLU, short for "rectified linear unit":

$$\sigma(z) = \begin{cases} z & z \geq 0 \\ 0 & z < 0 \end{cases}$$



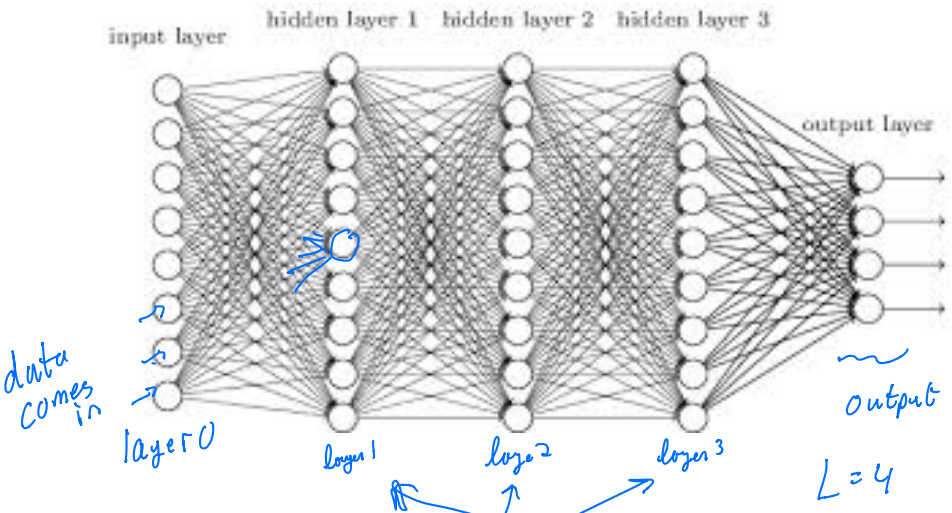


  Note the different units on the y axis!

- These last two, and variations on them, are the most-commonly used activation functions. They have the important properties of being continuous and differentiable (minor exception of ReLU at 0).

Layered, Feed-Forward Networks

- Multiple layers of neurons in a network: input, hidden and output



- Notation on the layers;
  - Layer 0 is the *input layer*; also called the *bottom layer*
  - Layers 1 to $L-1$ are *hidden layers*
  - Layer L is the *output layer*
  - The number of neurons at layer l is n^l.

superscripts mean layers
subscript are neuron indices

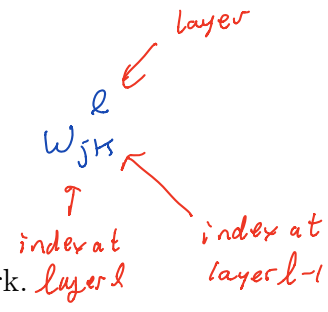
- Each neuron at layer l, $l \geq 1$, is connected to each neuron at layer $l-1$.
- Computatation proceeds layer by layer — this is called *feed forward* or *forward propagation*.
- The result of the network is produced by interpreting the activations of the output layer neurons.
  - Example we will soon see: 10 output neurons, one for each digit

Notation on the Feedforward Computation — Component Form

- x_k is the input to the k-th neuron at layer 0 — the k-th value from an input data vector (or image).
- a_k^l is the "activation" — the output — of neuron k at layer l
  - As a special case, we note that $a_k^0 = x_k$ since, by convention the input layer reproduces the input as its activation
- w_{jk}^l, for $l \geq 1$, is the weight of the connection from neuron k at layer $l-1$ to neuron j at layer l.
- We write the combined input to neuron j at layer l as

$$z_j^l = \sum_{k=1}^{n^{l-1}} w_{jk}^l a_k^{l-1} + b_j^l$$

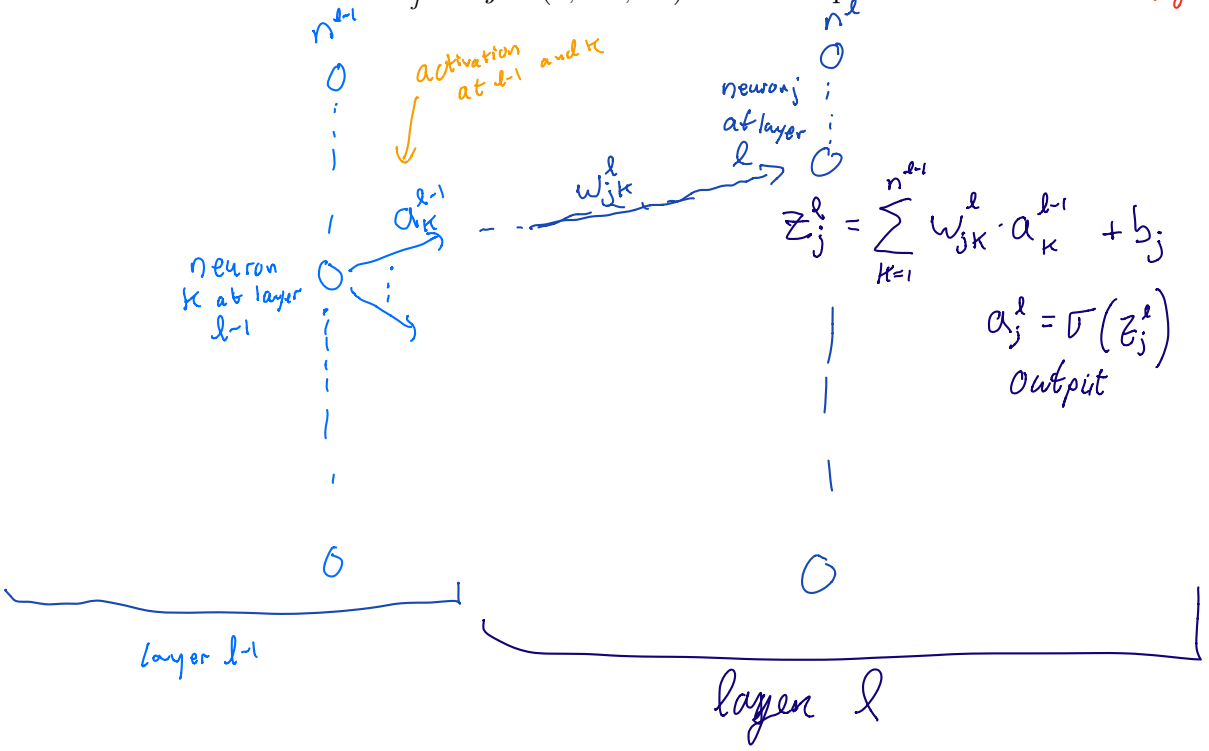
  (we will usually not write the bounds on k in the summation).



- The activation (output) from neuron j at layer l is

$$a_j^l = \sigma(z_j^l).$$

- The values a_j^L for $j \in (1, \ldots, n^L)$ at the output from the network.



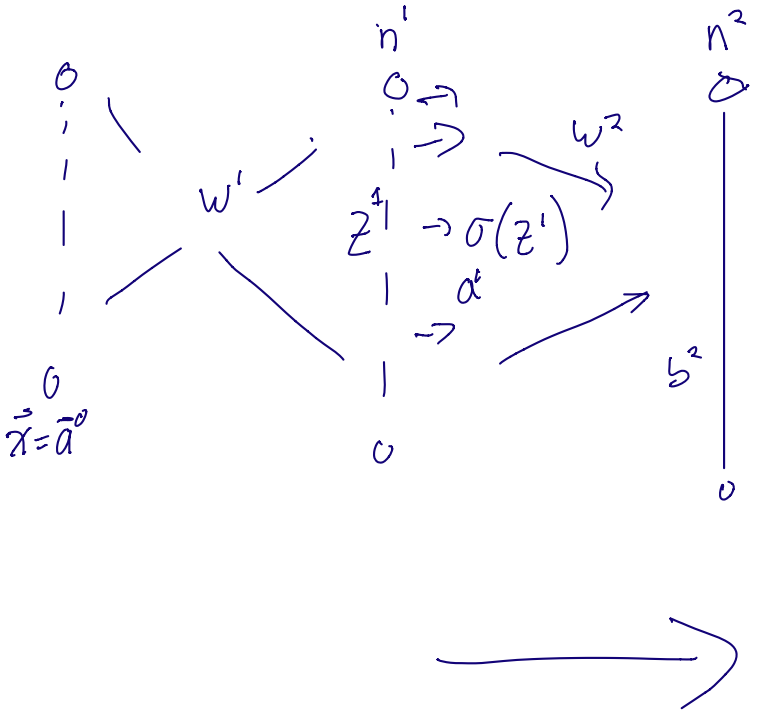
Notation on the Feedforward Computation — In Matrix Form

- $\mathbf{x}$ is the vector of input values
- $\mathbf{a}^l$ is the vector of activations at layer l, with
  - $\mathbf{a}^0 = \mathbf{x}$ is the input and
  - $\mathbf{a}^L$ is the output.
- $\mathbf{w}^l$ is the $n^l \times n^{l-1}$ matrix of weights coming into layer l.
- $\mathbf{b}^l$ is the vector of biases at layer l
- $\mathbf{z}^l$ is the vector of inputs to layer l, with

$$\mathbf{z}^l = \mathbf{w}^l \mathbf{a}^{l-1} + \mathbf{b}^l.$$

  We'll study this carefully in class.

- $\mathbf{a}^l = \sigma(\mathbf{z}^l)$ is the vectorized activation function computation.



Example: MNIST Data Set

See Figure ??

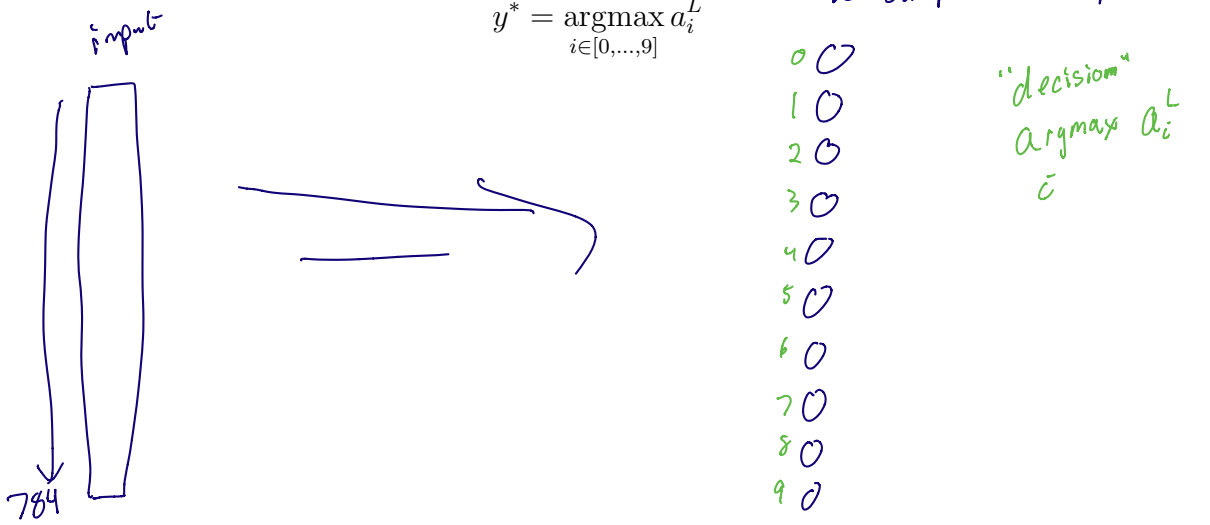
- Handwritten digit recognition
- 28x28 binary images
- 60,000 training and 10,000 test
  - A training set is the set of images on which the algorithm "learns" — in this case the weights and biases.
  - The test set is the set of images that are used to measure the performance of the algorithm after learning is complete.
  - The training set is usually split into the actual training set and the validation set.
- For the purposes of our discussion we will ignore the problem of segmenting digits; assume it is solved...
- The current best error rate (Wikipedia page, November 2019) is 0.21%.

Figure 1: MNIST Examples

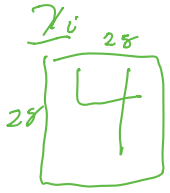
MNIST In a Network

- The 28×28 binary image becomes a $28 \times 28 = 784$ component vector input to the bottom layer
- The top/output layer contains 10 neurons, one per digit.
- Given an input, the computation "feeds forward" through the network until reaching the top layer,
- The neuron at this layer with the greatest activation is the "decision" by the network i.e.,

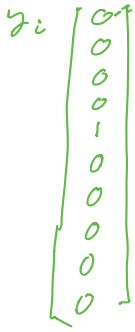
$$y^* = \underset{i \in [0,\ldots,9]}{\operatorname{argmax}}\, a_i^L$$



Learning: Training Data



- Many examples to learn from. Each has
  - $\mathbf{x}_i$ is the i training image
  - y_i is the manually labeled decision about the digit in image $\mathbf{x}_i$.
    - y_i may be a *"one hot"* vector: a vector of nine 0's and a single 1 at the index of the manual label



- As we have discussions, we represent y_i as a vector $\mathbf{y}_i$ of 0's with just a single 1 at the location corresponding to the desired digit.
- 60,000 pairs of input / output training (and test) data in MNIST
- **Goal:**
  - "Learn" to get the best set of weights and biases on the training data set
  - "Test" or evaluate on the test set to determine how well the result works.

classification accuracy

- Even though performance on the test set feels like our real goal, we aren't allowed to use the test data to modify the weights and biases. Why might this be?

Learning: Minimize Cost Function

- Strategy: formulate a cost function and then minimize it.
- Here is one simple cost function for the error in one input/output pair (after the feedforward computation):

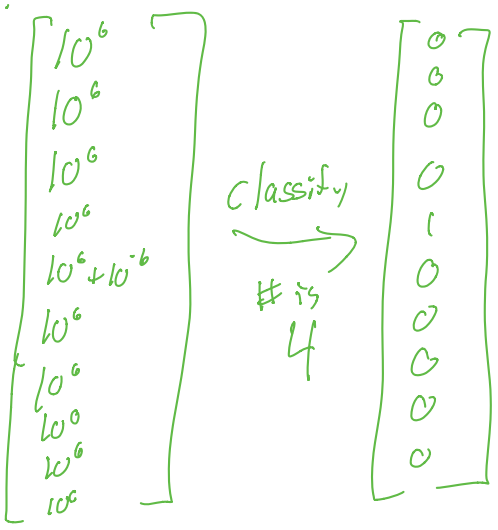
$$C(\mathbf{x}_i, \mathbf{y}_i) = \frac{1}{2}\|\mathbf{y}_i - \mathbf{a}^L\|^2$$

- Notes:
  - the cost function does not appear to depend on the weights and biases, but this dependence is implicit through $\mathbf{a}^L$,
  - $\mathbf{a}^L$ also depends on $\mathbf{x}_i$, and
  - to be sure things are clear, make sure you understand why $\mathbf{y}_i$ does not depend on the weights and biases.

- Issues to think about:
  - Why don't we write a cost (objective) function that simply counts 1 if the network made a mistake, and 0 otherwise? Then we could search for the weights and biases that minimize this? does not give clean enough error signal
  - In what ways is this cost function similar to and different from other least-squares type object functions we've discussed before?
- We will discuss other cost functions in the next lecture.

How far can a^L be from y_i and still give correct classification?

Hugely



huge error but still right!

Aside: Minimization Through Gradient Descent

- Objective function is far too complicated to solve in a closed form like we did for least-squares line fitting or transformation matrix parameter estimation.
- Minimize instead through a conceptually-simple method called *gradient descent*.
- Abstract description follows...

- Given function $f(\mathbf{x}; \boldsymbol{\theta})$ of (known) training data $\mathbf{x}$ and (to be estimated) parameters $\boldsymbol{\theta}$, our goal is to find the values of $\boldsymbol{\theta}$ that minimize f for fixed $\mathbf{x}$.
- Suppose we have an initial estimate $\boldsymbol{\theta}_0$ and wish to compute a new $\boldsymbol{\theta}_1$.
  - Usually formulated in terms of finding $\Delta\boldsymbol{\theta}$ and then calculating $\boldsymbol{\theta}_1 = \Delta\boldsymbol{\theta} + \boldsymbol{\theta}_0$.
- Goals can be either
  1. Find $\Delta\theta$ in terms of the step that maximixes the change in f or
  2. The step that guarantees a negative change in (reduction of) our cost function
- Both lead to assigning
$$\Delta\boldsymbol{\theta} = -\eta\nabla_\theta$$
where η is a small positive constant.

$\theta = (w, b)$

$f(x; \theta_1) < f(x; \theta_0)$

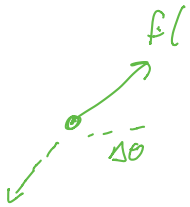
$f(x; \theta_0 + \Delta\theta) < f(x; \theta_0)$

$f(x; \theta_0) + \nabla f_\theta(x; \theta_0)\,\Delta\theta < f(x; \theta_0)$

$\nabla f_\theta(x; \theta_0)\,\Delta\theta < 0$

$\nabla f_\theta(x; \theta_0)$: direction of greatest increase

$\therefore$ go in opposite direction

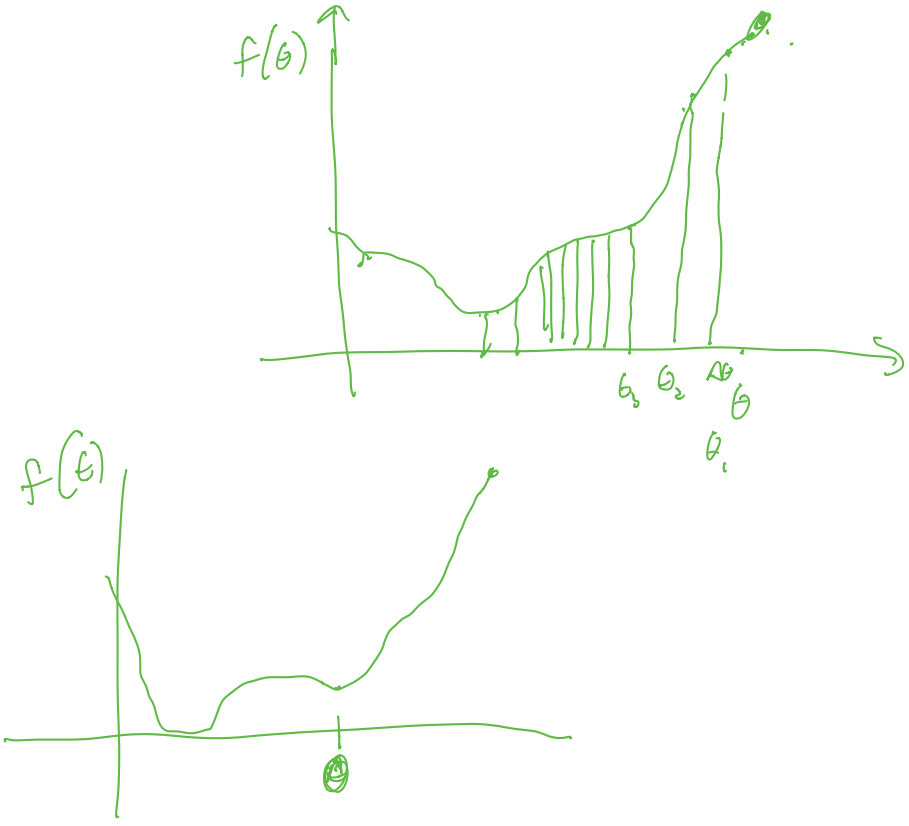


$\Delta\theta = -\eta\,\nabla_\theta f(x; \theta_0)$

η: small size

- This indicates an iterative process that alternates computing the gradient for the current $\boldsymbol{\theta}$ and then making a small change to $\boldsymbol{\theta}$ in the negative gradient direction.
- Two important notes:
  - This requires a starting estimate.
  - The minimum obtained is a local minimum, and is not guaranteed to be a global minimum.

  Concerns about these impeded progress on artificial neural networks for many years.



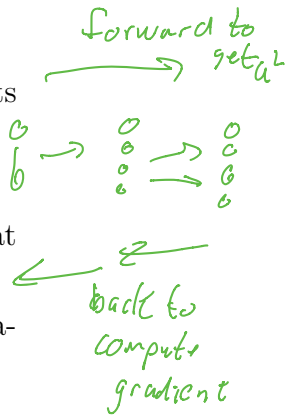
leave off x

Neural Network Training (Learning) Via Gradient Descent

- Parameters to be estimated are the weights and biases at each layer:

$$\{(\mathbf{w}^l, \mathbf{b}^l), \text{ for } l \in [1, \ldots, L]\}$$

- Initialization is through a random Gaussian distribution.
  - More on this in the next lecture.
- Then start gradient descent.
- For each training data instance $\mathbf{x}_i, y_i$:
  1. Apply the feed-forward computation through the network with its current weights and biases.
  2. Compute the cost – the loss at the output layer
  3. Compute the gradient of the cost with respect to all $\mathbf{w}^l$ and $\mathbf{b}^l$ at all layers!
  4. Make a small change in the weights and biases in the negative gradient direction to improve the cost.



- This is repeated many times over many training instances.
  - The true gradient is obtained by summing the individual gradients over the entire training set, but as we will see this is never done in practice.
- The important step to explain is the computation of the gradient using a method called *backpropagation*.

Backpropagation to Compute the Gradient

Think for now in terms of a single input training image $\mathbf{x}$ and its label y (we've dropped the index i)

- Need to compute derivative with respect to weights and biases.
- Done, one layer at a time, going backward through the network
- Introduce additional notation:

  $$\boldsymbol{\delta}^l = \frac{\partial C}{\partial \mathbf{z}^l}$$

  is the partial derivative vector of errors at layer l with respect to the input at that layer. During lecture we will discuss why this is referred to as an "error".
- The back propagation will compute this partial derivative vector recursively and then compute the desired gradients with respect to the weights and from this.

Backpropagation — Table of Equations

Equation	Component	Matrix
(1)	$\delta_j^L = \frac{\partial C}{\partial z_j^L} = (y_j - a_j^L)\sigma'(z_j^L)$	$\boldsymbol{\delta}^L = \nabla_{\mathbf{a}^L} C \odot \sigma'(\mathbf{z}^L)$
(2)	$\delta_j^l = \frac{\partial C}{\partial z_j^l} = \sigma'(z_j^l) \sum_k w_{kj}^l \delta_k^{l+1}$	$\boldsymbol{\delta}^l = \sigma'(\mathbf{z}_j) \odot (\mathbf{w}^{l+1})^\top \boldsymbol{\delta}^{l+1}$
(3)	$\frac{\partial C}{\partial b_j^l} = \delta_j^l$	$\frac{\partial C}{\partial \mathbf{b}^l} = \boldsymbol{\delta}^l$
(4)	$\frac{\partial C}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}$	$\frac{\partial C}{\partial \mathbf{w}^l} = \boldsymbol{\delta}^l (\mathbf{a}^{l-1})^\top$

Notes:

- Recall that $\sigma(z)$ is the activation function and therefore $\sigma'(z)$ is derivative of this function.
- The symbol $\odot$ is the component-wise multiplication ("Hadamard product") of two vectors.
- In equation (1) the component-wise term includes the factor $(y_j - a_j^L)$, which is specific to the quadratic cost (loss) function we are using. The matrix term is correct for all loss functions.
- $(\mathbf{w}^{l+1})^\top$ is the transpose of the layer $l+1$ weight matrix
- $\boldsymbol{\delta}^l (\mathbf{a}^{l-1})^\top$ is an outer product of dimension $n^l \times n^{l-1}$

Summary of Backpropagation Computation

- Applied during learning.
- Formulated here in terms of a single input / output training pair, $\mathbf{x}, \mathbf{y}$.
  - Below we will see how to combine this across multiple pairs.
- Apply the feed forward, layer-by-layer computation using the training pair and the current weights and biases. Record the combined inputs, z_j^l, activations, a_j^l, at each layer and each node in each layer.

- Then:

  1. Compute $\boldsymbol{\delta}^L$ at output layer.
  2. For each layer l, starting at L and going down to layer 1:
     (a) Compute $\boldsymbol{\delta}^{l-1}$ for the next layer below, layer $l-1$.
     (b) Compute the gradient for the bias terms $\mathbf{b}^l$ at the current layer from equation (3).
     (c) Compute the gradient for the weight matrix $\mathbf{w}^l$ at the current layer from equation (4).
     (d) Update $\mathbf{b}^l$ and $\mathbf{w}^l$ by taking a small step in the gradient direction:

$$\mathbf{w}^l \mathrel{-}= \eta \frac{\partial C}{\partial \mathbf{w}^l} \qquad \mathbf{b}^l \mathrel{-}= \eta \frac{\partial C}{\partial \mathbf{b}^l}$$

Stochastic Gradient Descent

- Problem:
  - With thousands of training examples, the true gradient of the cost function C with respect to the training set requires summing the training data over each training instance before update.
    - Very expensive
    - Leads to undesirable local minima
  - At the other extreme, the gradient with respect to a single instance is very noisy:
    - Allows escape of local minima, but
    - Very slow convergence
- Solution:
  - Break the M training instances into "mini-batches" of size m.
  - Average the m gradient values at each layer of the network to determine the negative gradient step direction in updating the network parameters.
  - Repeat for all M/m mini-batches.
  - Result is an "epoch" of training.
  - Training instances are randomly ordered before the start of each epoch.
- Typical mini-batch sizes are 16 or 32.

Software Design — Basic Ideas

For our simple hierarchical network it is not very difficult using NumPy, e.g.:

- Form a class for each layer containing:
  - Weight matrix, $\mathbf{w}^l$ and its gradient
  - Bias vector, $\mathbf{b}^l$, and its gradient
  - Computed input vector $\mathbf{z}^l$ and activation $\mathbf{a}^l$ for most recent data value.
- Generate random values to initialize the weight matrices and bias vectors
- Three nested for loops (in simplest form) over (1) epochs, (2) minibatches within an epoch, and (3) training instances within each minibatch:
  - Forward pass for each data point
  - Backward pass to update each gradient; accumulate gradient values.
  - Update the weight matrices and bias vectors at the end of each minibatch.

  Many of these can be vectorized.
- Repeat epochs until some measure of convergence is reached.

But, Don't Roll Your Own

- Network architectures that are effective are much more complicated than this
- Fast computation requires careful mapping onto GPUs
- SGD is not the only optimization method
- Need performance and convergence metrics.
- Why reinvent the wheel when there are many software packages, supported by the big players?

Looking Ahead to Software Packages

- Theano and Lasagne were earlier packages that
- Low level packages support tensor (multi-dimensional array) representations, gradient computations, and mapping onto GPUs. Examples:
  - Tensor Flow (Google)
  - MXNEt (Apache, Amazon)
  - Torch / PyTorch (Facebook)
  - Caffe2 (Facebook)
- High-level packages support network configurations and work with low-level packages "under the hood"
  - Keras — now part of Tensor Flow.
  - Modules within PyTorch / Tensor Flow.
- Somewhat of a distinction between research and production (e.g. PyTorch vs. Caffe2) and flexibility (PyTorch) vs. efficiency (MXNet).
- We are going to use the rapidly developing PyTorch because of its ease of use and tight Python integration.