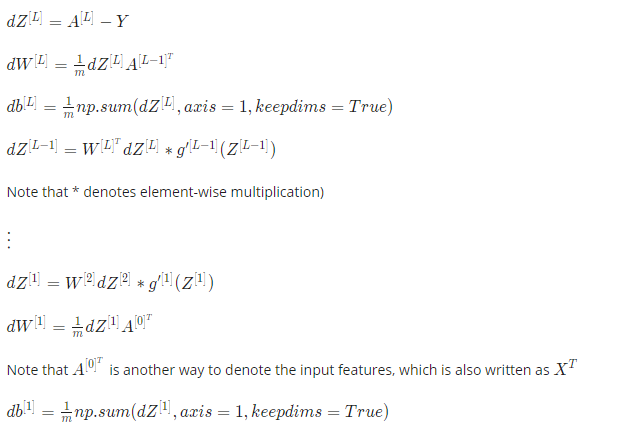
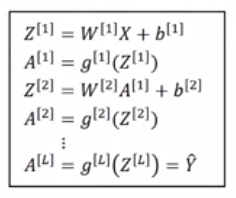
Kevin Chen

Course 1: Neural Networks and Deep Learning

Important note: the notes for this course are condensed, as there is a lot of repeat with the course “Machine Learning” by Andrew Ng. Please refer to the notes for the other course for more details on the presented concepts.

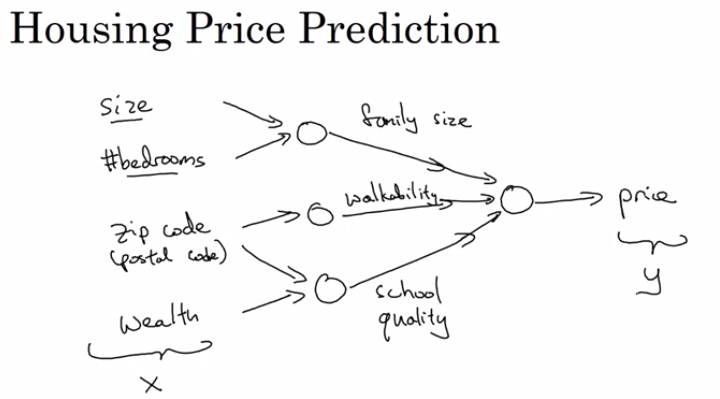
**Quick Reference: Forward and backward propagation of deep network:**

Forward prop: Back propagation:

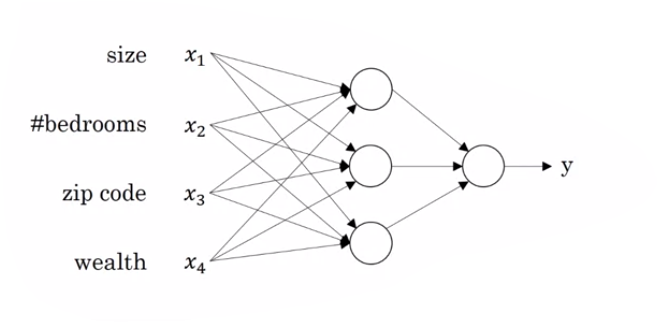


**Introduction**

* A neural network is a set of neurons that take in inputs and outputs a particular value. Each neuron is a function (e.g. ReLU function).
  + (Rectified Linear Unit (ReLU) function: if for some constant . if .)
* Example of a learned neural network (each circle is a neuron):



* In actuality, you begin by giving it a set of training examples (x, y), and make it fully connected in the middle. Input layer is (on left). Neurons in the middle form the hidden layers. Output layer is (on right).



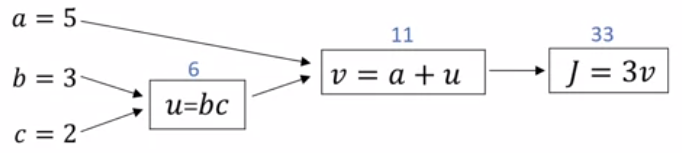
* Supervised learning is a type of machine learning, where you want to learn a function mapping some input x to some output y. Example applications:
  + Input: home features, output: price, application: real estate. Usually use standard neural network.
  + Input: ad and user info, output: click on ad?, application: online advertising. Usually use standard neural network.
  + Input: image, output: object category, application: photo tagging. Usually use convolutional neural network (CNN)
  + Input: audio, output: text transcript, application: speech recognition. Often use recurrent neural network (RNN).
  + Input: English, output: Chinese, application: machine learning. Often use recurrent neural network (RNN).
  + Input: image and radar info, output: position of other cars, application: autonomous driving.
* Supervised learning may have structured or unstructured data.
  + Structure data is essentially a database. Each feature has a well-defined meaning (e.g. house size in ft^2, #bedrooms)
  + Unstructured data examples include audio, images, and text. Historically has been harder to for computers to understand unstructured data, but this has changed with neural networks and deep learning.
* Deep learning has been taking off recently due to:
  + A big increase in amount of labeled data available as the world becomes more and more digital.
  + Increase in computational resources to be able to train large neural networks.
  + Algorithmic innovations allow neural networks to run much faster (e.g. ability to move from Sigmoid function to ReLU function for the activation function – ReLU function works better with gradient descent since Sigmoid has near-zero derivatives at ends, which means small steps and thus longer training times).
* As the amount of labeled data increases, the performance of large neural networks becomes better than smaller neural networks and traditional learning algorithms (e.g. SVM, logistic regression, etc.)
* With the ability to run neural networks faster, it is faster to iterate among the cycle of ideation, coding, and experimentations.

**Neural Network Basics**

* Binary classification: output is either or .
* Notation:
  + dimension of input feature vector . For brevity, let .
  + A training example is , where .
  + training examples. is the th training example.
  + Combine training example inputs together, where each training examples is its own column: .
  + Combine training example outputs together: .
* Logistic regression is used when the output labels in a supervised learning problem are all either 0 or 1 (i.e. binary classification).
  + Given , we want to output a hypothesis such that .
  + To achieve this, we output after learning the parameters of weights and intercept/bias term , where is the logistic function: .
  + As . And as .
  + Note: in some notation, becomes incorporated into the weights by adding an extra feature and . This course does not use this notation.
* The decision boundary is the line/plane that separates positive examples from negative examples. For logistic regression, the decision boundary is linear across inputs unless if the inputs contain nonlinear terms (e.g. or ).
* Loss function is a measure of how good our hypothesis is against for a single training example
  + We want the hypothesis on the th example to be as close as possible to the true output of the th example. In other words, we want .
  + The smaller the output of the loss function, the better our hypothesis is.
  + Using sum of squared errors, , is not a good idea for logistic regression since this produces a non-convex function that can result in local optima when performing gradient descent.
  + Hence, for logistic regression, we use the following loss function: .
  + This loss function has the property that if , we want to be large (close to 1), and if , we want to be small (close to 0).
* Cost function is a measure of how well our hypotheses are for all the training examples.
  + Goal of finding parameters and to minimize the cost function.
* Cost function justification:
  + Recall we want our prediction . This means and .
  + We can create a probability distribution of given :

This has the desired behavior of if , then and if , then

* + Note that optimizing is equivalent to optimizing because the function is monotonically increasing.
  + . When training a learning algorithm, we want to make to be large (and thus large) by minimizing the loss function , so we negative the loss function to achieve this behavior (i.e. decreasing increases ).
  + Assuming the training examples are drawn i.i.d (identically, independently distributed), then:
  + The principle of maximum likelihood estimation will choose the parameters that maximize and thus .
  + To make sure our quantities are better scaled, we multiply by to get our cost function of .
* Computation graphs: a graph with intermediate steps as nodes and edges showing the dependencies of each step. Example, if , that forms the following computation graph (example values in blue):



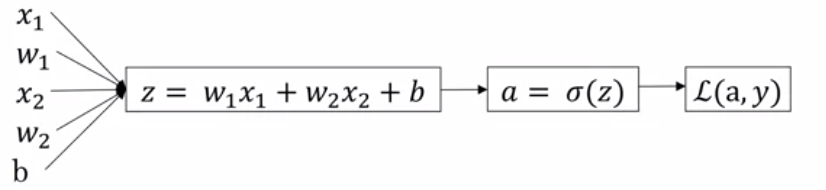
* + Use backpropagation and the chain rule to calculate the derivative of with respect to all the variables (i.e. ):

* Gradient Descent
  + Initialize values to some values (e.g. all 0’s, or small random initial values)
  + Takes a step in the direction of greatest decrease.
  + General gradient descent formula:

Repeat {

}

* + (Use partial derivative if function is a parameter of multiple variables, or the normal derivative if function is parameter of single variable)
* Logistic regression gradient descent
  + In logistic regression, usually initialize values to 0, but random small values also work. Doesn’t affect result since cost function is convex and will thus find global optimum regardless.
  + Create the computation graph for the logistic regression loss function (i.e. the cost of a single training example):
  + Computation graph for 2 features and and bias term:



* + Our goal for gradient descent is to find the derivative of the loss function against the weights and bias term: We compute the following values:
  + The cost function is the average of derivative of the loss function across each example:
  + Logistic regression gradient descent algorithm (using computation graph):

Initialize weights and intercept/bias term to zero (or small random values)

For each iteration of gradient descent:

Initialize for all ,

For to :

// is the hypothesis for , similar to .

For

End For

// divide by for average

for

End For

* + Two shortcomings of above formula: need for-loop for each example and for-loop for each feature . For deep-learning algorithms, you notice this is inefficient. Vectorization techniques gets rid of explicit for-loops.
* Vectorization allows us to eliminate for-loops in our code, which is necessary for large data sets to run efficiently.
  + With this approach, we assume that linear algebra libraries can matrix (and/or vector) multiplication much more quickly than using for-loops.
  + This is because the libraries can take advantage of parallel instructions (SIMD calculations – single instruction multiple data) with CPUs and GPUs.
  + The Python numpy library has a lot of element-wise operations for vectors/matrices, such as log, abs, maximum, minimum, exp, v\*\*k (raises each element in to the th power), 1/ (takes reciprocal of each element in ).
  + Whenever possible, avoid explicit for-loops.
* Vectorized notation:
  + is a vector equal to ( number of ’s).
* Vectorizing logistic regression:
  + We want for all . This can be vectorized by . (Note that Python broadcasting allows you to use just here.)
  + Next, we want for all . This can be vectorized by allowing to take in a vector and apply logistic function element-wise.
  + Next, we want for all . This can be vectorized by
  + To obtain for and , we can instead perform for . Note that this means we initialize to be a vector of zeros, and that we perform when taking the average. See next expression to avoid having to loop over training examples .
  + To perform for , we can instead perform , eliminating the need to later take the average.
  + To perform for , instead just do , where return the sum of the elements in the matrix. This eliminates the need to later take the average.
* Vectorized logistic regression algorithm:

For each iteration of gradient descent: // Can’t get rid of this for-loop.

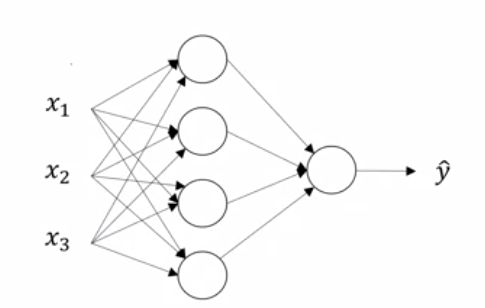
// Python broadcasting lets you use here instead (see below)

* For an numpy matrix , .sum(axis = 0) will sum the values in matrix vertically, returning a vector.
* Numpy broadcasting examples in Python

  + bsxfun in Matlab/Octave achieves something similar to broadcasting in Python.
* Python/numpy vectors advice:
  + Don’t use rank 1 arrays, as they can lead to unexpected behavior. Instead, explicitly use a or matrix.
  + If you are unsure if the variable a rank 1 array or matrix, then calling shape will return something like for some dimension , where you’ll get
  + a = np.random.randn(5) returns a rank 1 array, whereas a = np.random.rand(5, 1) returns a matrix. Asserting a.shape == (5, 1) ensures a is a matrix.
  + a = a.reshape((5, 1)) converts a from a rank 1 array to a matrix.

**Neural Network**

* Notation:
  + is the number of layers in the neural network.
  + is the number of nodes in the th layer. Note that , the number of input features. is the number of output units of the neural network.
  + is the activation function for the th layer (e.g. sigmoid function).
  + vector of linear combination of weights and inputs for the th layer of the neural network. is the linear combination of the th node in layer .
  + vector of activations for the th layer of the neural network. Note that . is the activation of the th node in layer . Also note (the training example) and (the neural network’s hypothesis).
  + vector of intercept terms for the th layer of the neural network. is the intercept term for the th node in the th layer.
  + is a matrix of weights mapping nodes in the th layer to the nodes in the th layer. is a vector of weights mapping nodes in the th layer to the th node in the th layer.
  + and are the respective values for the th training example. (Note that weights and are independent of the examples, so and doesn’t exist.)
  + We combine values across training examples into a single matrix for vectorization across all examples (more details below): , and
* Terminology:
  + The th layer of the neural network is the th hidden layer. The 0th layer are the inputs, and the last layer is the th layer. An -layer neural network has hidden layers, 1 input layer, and 1 output layer.
  + Example, this is a 2 layer neural network:



* + “Hidden layer” contains data not seen in the training set (i.e. data not in the input: and data not in the output ).
* Neural network is basically applying logistic regression many times: it computes and for forward propagation and and for backward propagation once per layer of the neural network.
* The activation of the 0th layer are the inputs, i.e. .
* Forward propagation: calculating the hypothesis , non-vectorized:

For :

For :

For :

// is equal to

Return // should a “vector” as

* Forward propagation: calculating the hypothesis across all training examples, vectorizing features:

For :

For :

// is equal to

Return

* Forward propagation: calculating the hypothesis across all training examples, vectorizing features and training examples:

For :

// .

Return // this represents the hypothesis .

* + Justification for the vectorized can be done by expanding the matrices and showing the above algorithm is the same as the un-vectorized one.
* Activation function
  + The function almost always works better than the sigmoid function for the inner layers. It is a scaled sigmoid function, going between -1 and 1 and crossing the origin. . Make sure to still use the sigmoid function for the output layer so that since .
  + Issue with and sigmoid function is that the derivative is roughly 0 if is small or large, which means gradient descent takes much smaller steps and thus takes much longer to converge.
  + ReLU function solves above issue. . (One small issue is that the derivative is not defined when . In the rare event where you’re trying to take the gradient of the function and is exactly 0, you can hardcode the derivative to be 0 or 1.) One issue is that if , then the derivative is 0, but it is uncommon for in practice.
  + Leaky ReLU is equal to for some small constant , such as – you can make this another parameter of the learning algorithm, but most people don’t do that. This solves the problem of ReLU’s derivative being 0 when . While Leaky ReLU does work better than ReLU, this is used less.
  + General rule of thumb: use sigmoid for node in last layer, use ReLU/Leaky ReLU for all nodes in the hidden layer (this is what most people use).
* A non-linear activation function is required by the neural network to learn interesting functions.
  + Let’s try setting (i.e. an “identity activation function”). This means
  + This shows that no matter how many hidden layers you have, the neural network will only learn linear functions (so you might as well not have any hidden layers)
  + A linear hidden layer is useless since the composition of two linear functions is a linear function.
  + The one place you might use a linear function is the output layer if the output can take on more values than just 0 or 1 (e.g. housing prices).
* Derivatives of activation functions
  + Derivative of sigmoid function : . If we use the sigmoid activation function , then .
  + Derivative of tanh function : . If we use the tanh activation function , then
  + Derivative of ReLU function : . (Technically is undefined if , but setting it to 0 or 1 in code is okay.)
  + Derivative of Leaky function : . (Technically is undefined if , but setting it to or 1 in code is okay.)
* Backpropagation algorithm, vectorized across all features:

For :

For :

If :

Else:

EndFor

EndFor

For :

* Note that the above implementation skipped computing , which can be done as intermediate step for computing :

If :

Else:

* Backpropagation algorithm, vectorized across all training examples and features:

For :

If :

Else:

// is Element-wise multiplication, both matrices are .

// Row-wise sum across the matrix , returning a vector

For :

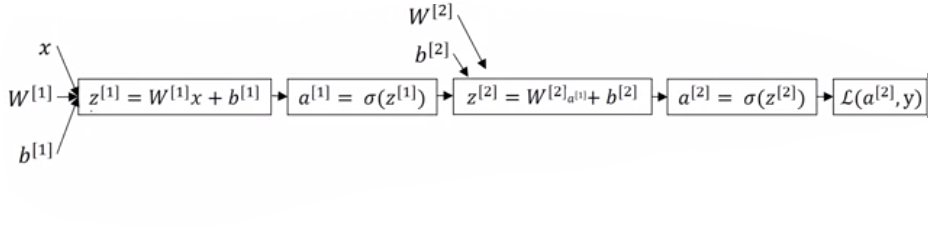
* Note that the above implementation skipped computing , which can be done as intermediate step for computing :

If :

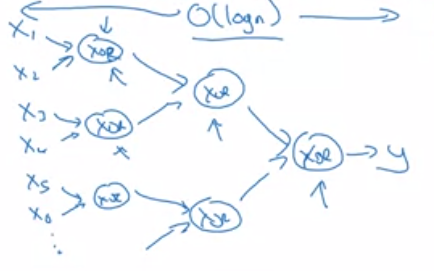
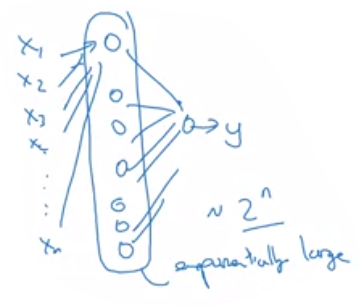
For

Else:

* Backpropagation intuition for a single training example: consider the 2-layer neural network computation graph



* + For the last layer (layer 2), we derive : Since , we get that . Next . This explains the first step of back propagation.
  + For the last layer, we derive . Note that there is a transpose for since consists of transposed vectors.
  + For the last layer, we derive .
  + For hidden layers and the first layer, we derive .
  + For hidden layers and the first layer, the derivation for and are the same for the respective values in the last layer: and .
* We can then convert the backpropagation for a single example to be vectorized across all examples. The approach is similar to how we vectorized logistic regression.
  + There’s an extra term in , which arises from the term in
* Unlike logistic regression where we can initialize the weights to zero, we must initialize the weights randomly for neural network. Otherwise the weights across each layer will always be symmetric (i.e. identical).
  + The rows of will be identical.
  + This defeats the purpose of having more than one hidden unit.
  + Use small random initial values. If the weights are large, then is more likely to be large (positively or negatively), which means the derivative of the sigmoid/tanh function will be small, and gradient descent will take a while.
  + can be initialized to 0’s because the ’s are random and thus should be different than the ’s
* There is no way to avoid the for-loop that loops across layers in a neural network.
* One way to help debug your code is to verify on paper that the matrix dimensions are correct after each step of forward and back propagation. Be sure to use different sized hidden layers to ensure the checks are more robust.
  + The derivative of the cost function with respect to any matrix should be the same dimensions as the matrix itself. For example, should have the same dimensions as .
* Intuition why deep representations (i.e. models with many layers) are powerful: First layers are usually simple components, whereas final layers are complex features. Certain people argue this is similar to how the human brain works: begin with detecting simple components and then building more complex features. Examples of deep networks below:
  + Face detection: input is an image, first layer detects simple components of the image, such as edges. Next layers try to group simple components into parts of human face, such as eyes, nose, mouth. Next layers then try to use those parts of the human face to recognize it.
  + Speech recognition: input is audio, first layers of neural network detect low level audio waveform, which then get composed in the next layers into basic units of sound (known as phonames), which then get composed in the next layers to recognize words, which then get composed in the next layers to recognize sentences/phrases.
* Circuit theory also shows why deep representations can be more powerful: there are functions that you can compute with a “small” L-layer deep neural network that shallower networks require exponentially more hidden units to compute.
  + Example, you want to calculate . With levels, we can compute this with XOR gates. With just 1 level, we need gates to enumerate all possibilities:

* Parameters vs. hyperparameters:
  + Parameters: , where represents for and represents for .
  + Hyperparameters: learning rate , # of iterations, # of hidden layers , # of hidden units for , choice of activation function. Hyperparameters later discussed are the momentum, minibatch size, and regularization term.
* Applied deep learning:
  + Is a very empirical process since so many hyperparameters need to be tuned
  + Has many applications: vision, speech, NLP, online advertising, web search, recommendations, etc.
* The best set of hyperparameters can change over time.
* There are analogies between a single neuron in a neural network and a single neuron in the brain, but it is still unclear today what a single neuron does in the brain.
  + Unclear if we do forward/back propagation in the brain.
  + Analogy used to show we can learn functions for supervised learning: a mapping between input and output .