Deep Learning with Python

Alex Pan

2017-11-08

Chapter 1

Course I: Introduction to Neural Networks

1.1 Introduction

1.1.1 What is a Neural Network?

Andrew provides an analogy for how one might think about a neural network. When trying to predict the price of a house, we may look at features like: postal code, size, and number of bathrooms, types of floors, etc.

When we try to determine how much we are willing to pay for a house, we don't necessarily care about these exact parameters, but we feed them forward into new features. For example:

- Postal code may tell us about the neighbourhood, and what schools are in the area.
- The size and number of bathrooms in the house are important relative to the size of your family.
- We may not care about the types of floors.

A neural network, in effect, does the same thing. However, we don't actually tell a neural net anything about what features are important. We let the neural net decide. In practice, this means that:

- Every input neuron is connected every hidden layer neuron.
- The relative importance of each input feature on each hidden layer feature is determine by the weights and biases that are calculated by the neural net.

Supervised Learning

There is a lot of hype about neural nets. Given how well they work, some of this hype may be justified. However, most of the economic gains associated with neural nets has come from *supervised learning*. Neural nets have been useful in:

Application	Neural Net Architechture
real estate	Standard NN
online advertising	Standard NN
image recognition	CNN
speech recognition	RNN
machine translation	RNN
autonomous driving	Complex/hybrid NN architechture

We will talk more about RNNs and CNNs later.

Structured vs. Unstructured Data Different neural nets deal with both structured and unstructured data.

Structured data refers to data that follows the format you may find in a database. I.e., there are columns of variables and rows of observations. A lot of the short-term economic value of neural nets has been using structured data (ex., advertising)

Unstructured data refers to data that do not follow this format. Examples include raw audio files, images, or text. Historically, computers have sucked at interpreting unstructured data compared to humans. But with the rise of deep learning, we are now able to interpret unstructured data much better than before.

Why is Deep Learning taking off?

The base technology of neural nets has been the same for decades. The reason why deep learning has taken off in recent years has to do with the availability of data, due to the digitization of our society. With traditional learning methods, performance does not increase after reaching a certain level of data.

Neural networks seem to improve continuously when given large amounts of *labeled* data. Of course, this is because neural networks can be infinitely complex. Large neural nets can achieve amazing perfomance metrics, but require a large amount of data. Smaller neural nets, similar to traditional methods, will plateau when a certain amount of data is reached (though they may learn 'longer' than traditional methods).

"Scale has been driving deep learning process" This refers to scale of:

- data (ease of collecting data)
- computation (faster computation)
- algorithms (inventing and improving algorithms)

In fact, two simple solutions to improve the performance of a learning algorithm are (1) train a bigger neural net, or (2) get more data. Of course this only works up to a point, because you may run out of data or run into a neural net so large that it takes too long to train.

Additionally, at the "low end" of training data sizes, the 'hierarchy' of learning algorithms is not as well defined. While large neural nets > small neural nets > traditional algos for large data sets, this is not necessarily true for small data sets. In smaller data sets, cleverly hand-engineering features may provide better performance than just throwing a very large neural network at it.

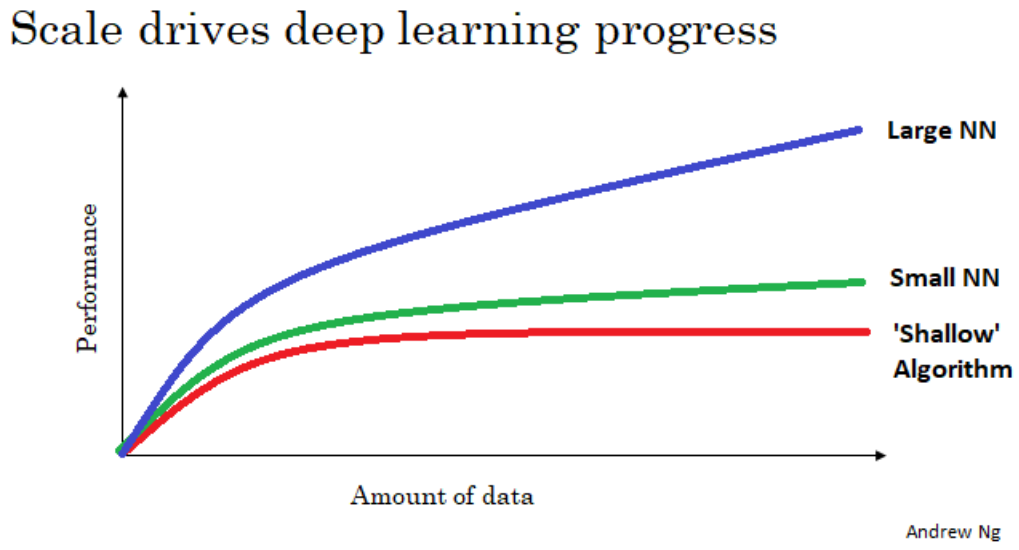


Figure 1.1: Scale drives deep learning progress

Example: Implementation of the ReLU function This is more of a tangent - but it is interesting. Old neural nets used a sigmoid activation function. However, because sigmoid functions have a very shallow slope on their tails, they learn very slowly (i.e., long computation times).

In switching from a sigmoid to a Rectified Linear Unit (ReLU) function speeds this up. ReLU is a linear function that takes on a value of 0 at a certain threshold. Though not as elegant as a sigmoid function, it is computationally much faster.

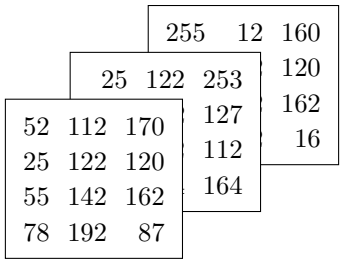
1.2 Logistic Regression as Simple Neural Network

In this section, logistic regression is used as an example to demonstrate certain properties of neural nets. Even if you are familiar with logistic regression you will benefit from this section.

1.2.1 Binary Classification

Basic Image Classification Recall that logistic regression is an algorithm used for binary classification (i.e., cat vs. not cat). In a computer, 2-D images are stored as a tensor consisting of matrices of R, B, and B intensities for each pixel. For example, a 64x64 pixel image would have the dimensions 64x64x3.

In order to turn this tensor into something machine readable, we *"unrow"* each matrix (i.e., aggregate each matrix into a single column). Ex., for a simple 4x3 image, all of the pixel intensities in all of the channels are converted into a column vector of length $n_x = 36$.



$$\downarrow$$

$$x = \begin{bmatrix} 52 \\ 112 \\ \vdots \\ 25 \\ 122 \\ \vdots \\ 16 \end{bmatrix}$$

We could then use this to make a neural network with 36 input layers, where each one takes in a pixel intensity value.

1.2.2 Notation

Sizes:

- m : number of examples in the data set (m_{train} or m_{test} may be used in some cases)
- n_x : input size (i.e., number of units in input layer, or the number of features)
- n_y : output size (i.e., number of classes)
- L : number of layers in the network
- $n_h^{[l]}$: number of hidden units (h) in the l^{th} layer.
  - Note: you may also denote $n_x = n_h^0$ and $n_y = h_h^{[L+1]}$

Objects:

- $x \in \mathbb{R}^{n_x}$: x is a vector of the inputs for a single feature with length m. (i.e., the red intensities for a particular pixel in all training images)
- $x^{(i)} \in \mathbb{R}^{n_x}$: $x^{(i)}$ is the i^{the} example of the of the colum vector x.
- $X \in \mathbb{R}^{n_x \times m}$: X is matrix of $n_x \times m$ dimensions consisting of real numbers.

  - Note that *in this context* the mtarix X is defined such that each of features is represented as a row and each training example is represented as a row. (i.e., there n_x number of rows and m number of columns).

  - This is different from the usual representation, but this facilitates implementation of the neural net.

- $y \in 0, 1$: y is the true label of the training example.

- $y^{(i)} \in \mathbb{R}^{n_x}$: $y^{(i)}$ is the output label for the i^{th} example.

- $Y \in \mathbb{R}^{n_x \times m}$: Y is the label matrix (is this the matrix of true answers?)

- $W^{[l]}$: is the weight matrix (i.e., a matrix full of weights). $[l]$ indicates the layer. The dimensions of $W^{[l]}$ are:

  - *number of units in next layer* $\times$ *number of units in the previous layer*

- $b^{[l]}$ is the bias vector for the l^{th} layer. Its length is equal to the *number of units in the next layer.*

- $\hat{y} \in \mathbb{R}^{n_y}$: $\hat{y}$ is the predicted output vector (i.e., our hypothesis. It can also be denoted as $a^{[L]}$ (where L is the number of layers in the network)

1.2.3 Logistic Regression

Notation Explained

Let's pretend that we are doing a logistic regression with one feature. (x, y) represents one training example. x is an n_x-dimensional vector (i.e., it is the vector of all of the features). y is the output vector (for binary classification, it takes on the number 0 or 1).

The training set consists of m training examples. So the entire training set looks like: $((x^{(1)}, y^1), (x^{(2)}, y^2), ..., (x^{(m)}, y^m),)$

A more compact way of representing these training sets is using a matrix. Here the matrix X represents all of the inputs for all of the features in the training set:

$$X = \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^1 & x^2 & \dots & x^m \\ \vdots & \vdots & & \vdots \end{bmatrix}$$

With dimensions $n_x \times m$.

Note that this is NOT the way we usually represent matrices. Here we are representing each feature in rows and each training observation in columns. If we apply Pandas `X.shape`, it will output (n_x, m). However, this form makes implementation easier.

1.2.4 Logistic Regression

The basic idea behind logistic regression in binary classification is that it constrains the output to 0 or 1. Linear models can take on values greater than 1 and less than 0, which does not make sense in the context of probability. Logistic regression takes the form of linear regression and wraps it around a logistic function:

$$\hat{y} = \sigma(W^T x + b)$$

Where:

W is the weight matrix,

b is the bias

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

Where:

$z = W^T x + b$

Ignoring the details of the math and looking only at the intution, we see that the sigmoid function is constrained between 0 and 1. We also see that large values for z give bigger outputs and that small values of z give smaller outputs.

1.2.5 Logistic Regression Cost Function

Loss Function

We are trying to predict W and b in order to compute $\hat{y} \approx y$. Mean squared error doesn't work for the logistic function because it may produce multiple local minima, which are not good for gradient descent.

Instead we use the form:

$$Loss(\hat{y}, y) = -(y \log \hat{y}) + (1 - y) \log 1 - \hat{y}$$

Intuitively, we see that if $y = 1$ then the second term is equal to 0. So the $\hat{y}$ that would minimzie this loss function would need to be large.

If $y = 0$, then the first term is equal to 0, and $\hat{y}$ would need to be small.

Of course there are many possible functions that would have similar properties, but there is a reason that this particular form is preferable (i.e., black magic).

Cost Function

The loss function provides a measure of error for a single training example. The cost function summarizes all of the examples:

$$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} Loss(\hat{y}^{(i)}, y^i)$$

i.e., it is the average loss of across all of the training examples.

1.2.6 Gradient Descent

Having just been introduced to to the cost function for logistic regression, we can now discuss gradient descent. Gradient descent is used to 'learn' w and b in order to minimize $J(w,b)$.

$J(w,b)$ is convex (i.e., it has one global optimum), which is why we use this cost function for gradient descent. We initalize some value for w and b as a first guess (for logistic regression this usually 0, but random selection also works). The initial first guess doesn't really matter, because if the solution is convex we will arive at the same solution regardless.

Minimization of $J(w,b)$ is an iterative process. In psuedo-code we would write (assuming there is no bias parameter for simplicity):

Repeat:

$$w := w - \alpha\frac{\delta J(w)}{\delta w}$$

Where:

α is the learning rate (i.e., step size),

$\frac{\delta J(w)}{\delta w}$ is the derivative.

Note: b would be learned the same way.

Note: In Python we denote $\frac{\delta J(w)}{\delta w}$ as `dw`.

Intution Behind Derivatives

This section was pretty high level. There were only two main points:

- The derivative is the slope of an infinitesimal nudge in the input parameter
- The derivative can change depending on 'where' the function is (i.e., the value of the input parameter).

Power Rule: The derivative of $f(a) = a^2$ is $\frac{d}{da}f(a) = 2a$

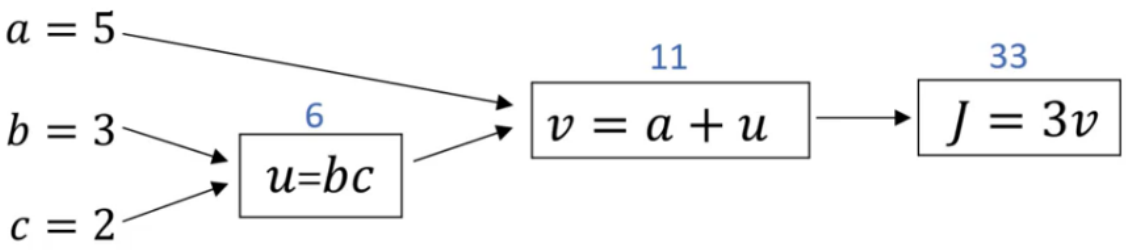


Figure 1.2: Example computation Graph with Derivatives

Logarithms: The derivative of $f(a) = ln(a)$ is $\frac{d}{da}f(a) = \frac{1}{a}$

Chain Rule: For a function that looks like: $f(J) = 3v$ where $v = a + u$:

$\frac{df(J)}{da} = \frac{df(J)}{dv}\frac{dv}{da}$

Computation Graphs To Visualize Back Propagation

For the purposes of visualization, we will construct a computation graph of our function. Rather than using the logistic regression loss function, for now we will use the simple function.

$$J(a, b, c) = 3(a + bc)$$

Let:

$u = bc$,

$v = a + u$,

$J = 3v$.

Figure 1.2 provides a graphical representation of the function. We may use 'forward propagation' in order to find the value of J given $a = 5$, $b = 3$, and $c = 2$. It turns out that 'back propagation' provides a simpler way to compute the derivatives of the function.

As a convention, whenever we are writing $\frac{dJ}{dVariable}$ in code we write `dvar`. Where 'var' is the name of the variable (e.g.,`da, db, dc, du, dv`. We ignore the J because it is assumed that the derivative is for this variable.

What is the derivative of J with respect to v?

If we use Andrew Ng's heuristic technique where we nudge v by 0.001:

Recall: $J = 3v$

$v = 11 \rightarrow 11.001$

$J = 33 \rightarrow 33.003$

J increased 3 times the amount that v increased. Therefore:

$$\frac{dJ}{dv} = 3$$

What is the derivative of J with respect to a?

Recall: $v = a + u$

$a = 5 \rightarrow 5.001$

$v = 11 \rightarrow 11.001$

$$\frac{dv}{da} = 1$$

We can apply the chain rule:

$$\frac{dJ}{da} = \frac{dJ}{dv}\frac{dv}{da} = 3 \times 1 = 3$$

What is the derivative of J with respect to b?

Recall: $u = bc, c = 2$

$b = 3 \rightarrow 3.001$

$u = 6 \rightarrow 6.002$

$$\frac{du}{db} = 2$$

Once again, we apply the chain rule:

$$\frac{dJ}{db} = \frac{dJ}{du}\frac{du}{db} = 3 \times 2 = 6$$

c is solved the same way as b, and you will find that it works out to 9.

1.2.7 Gradient Descent for Logistic Regression

In the last two sections, we discussed some rules for calculus. In this section, we will discuss how to apply those sample principles to the logistic regression cost function. We will represent logistic regression with a computation graph (note that we use a as the notation for $\hat{y}$ for Python purposes).

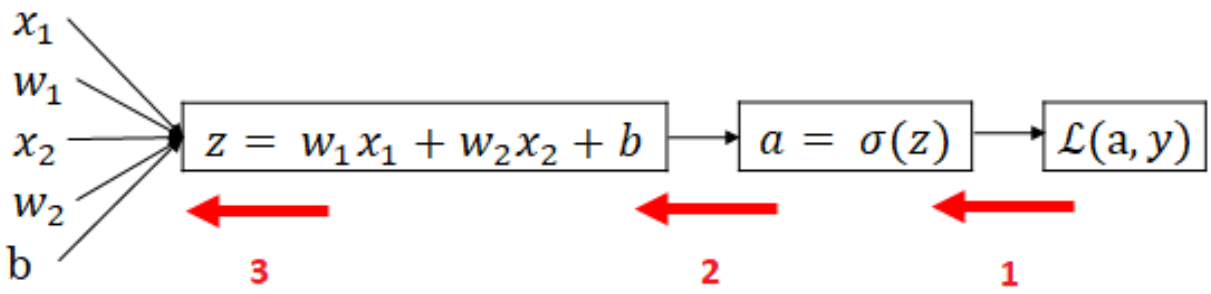


Figure 1.3: Logistic Regression Computation Graph

Recall the following equations for logistic regression:

$z = w_1x_1 + w_2x_2 + b$

$a = \sigma(z)$

$Loss(a, y) = -(y \log a) + (1 - y) \log 1 - a$

We can perform backprogagation simply by evaluating each equation using the Power Rule and then combining them using the Chain Rule (Figure 1.3). See if you can work out these derivatives yourself from the equations.

1. Derivative *wrt* to *Loss*: $da = \dfrac{dLoss}{da} = -\dfrac{y}{a} + \dfrac{1-y}{1-a}$

2. Derivative *wrt* z: $dz = \dfrac{dLoss}{dz} = \dfrac{dLoss}{da} \cdot \dfrac{da}{dz} = a - y$

3. Derivative *wrt* w or b: $dw_1 = \dfrac{dLoss}{dw_1} = \dfrac{dLoss}{dz}\dfrac{dz}{dw_1} = x_1 \cdot dz$

1.2.8 Implementing Gradient Desecent for Logistic Regression

In the previous section we went through an example of how a single iteration of gradient descent may be applied to a single training example with two features. We will need to iterate the same process over all training examples in order to compute the cost function and the derivative for each parameter. Conceptually, we can describe the implementation of gradient descent as follows (in reality this is a very inefficient way, so we seek to vectorize it):

```
# This code doesn't actually work. It's just for illustration.
import numpy as np

# w is an n_x-dimensional column vector with values that are already defined
# Y is an m-dimensional column vector with values that are already defined
# X is an nx by m matrix
# sigma(z) is the previously defined logistic function

J = 0
db = 0
```

```
z = np.zeros((1, m))
a = np.zeros((1, m))
dw = np.zeros((1, m))

for i in range(m):
   # Forward prop: Compute z and a for the training example
   z[i] = w.T[i] * X[i] + b
   a[i] = sigma(z[i])
   # Add the loss from this training example to the cost function
   J += y[i]*log(a[i]) - (1-y[i]) * log(1-a[i])

   # Back Prop: Compute the dz, dw, and db
   dz[i] = a[i] - y[i]
   db += dz[i]
   for j in range(nx)
      dw[i] += X[j][i] * dz[i]
   # ... ETC. (compute weights for each of the features)

# Outside of the loop, take the average for each of the parameters
J /= m
dw1 /= m
dw2 /= m
db /= m
```

Notice how we have to iterate over every feature for every training example. This is very inefficient. Luckily, it is pretty easy to implement vectorization in Python.

1.2.9 Python for Logistic Regression

Vectorizing Backward Propagation
In the previous section we wrote a piece of code to run one iteration of gradient descent. We could achieve the same result much faster using vectorized code:

```
import numpy as np

# w is an n_x-dimensional column vector with values that are already defined
# Y is an m-dimensional column vector with values that are already defined
# X is a nx by m matrix
# sigma(z) is the previously defined logistic function

J = 0
Z = np.zeros(1, m)
A = np.zeros(1, m)
dZ = dW = np.zeros(1, m)
dW = np.zeros(nx, 1)  # empty array with dimensions n_x by 1
dB = 0

# Forward prop: Compute z and a for the training example
Z = np.dot(w.T, X) + b
A = sigma(Z)

# Add the loss from this training example to the cost function
J = 1/m * np.sum(y*log(A) - (1-y) * log(1-A))

# Back Prop: Compute the dz, dw, and db
dZ = A - Y
```

```
dB = 1/m * np.sum(dZ)
dW = 1/m * X * np.transpose(dZ)
```

Explanation

Vectorizing Forward Propagation

We are trying to solve $z^{(1)}$ through $z^{(m)}$.

Recall:

$z^{(i)} = W^T X^{(i)}$ for all of $1, ..., m$

Let:

Z be a a $1 \times m$ dimensional row vector ($Z = [z^{(1)}, ..., z^{(m)}]$).

X be a $n_x \times m$ dimensional matrix

W^T be a $1 \times n_x$ dimensional row vector (i.e., $W^T \in \mathbb{R}^{1 \times n_x}$).

B be a $1 \times m$ dimensional row vector containing b broadcasted m times.

A be a a $1 \times m$ dimensional row vector ($A = [a^{(1)}, ..., a^{(m)}]$).

Then:

$Z = W^T X + B$

$A = \sigma(Z)$

Vectorizing Backward Propagation

Recall:

$dz^{(i)} = a^{(i)} - y^{(i)}$ for all of $1, ..., m$

$$db = \frac{1}{m} \sum_{i=1}^{m} dz^{(i)}$$

$$dw = \frac{1}{m} \sum_{i=1}^{m} x^{(i)} dz^{(i)}$$

Let:

dZ be a $1 \times m$ dimensional row vector ($dZ = [dz^{(1)}, ..., dz^{(m)}]$).

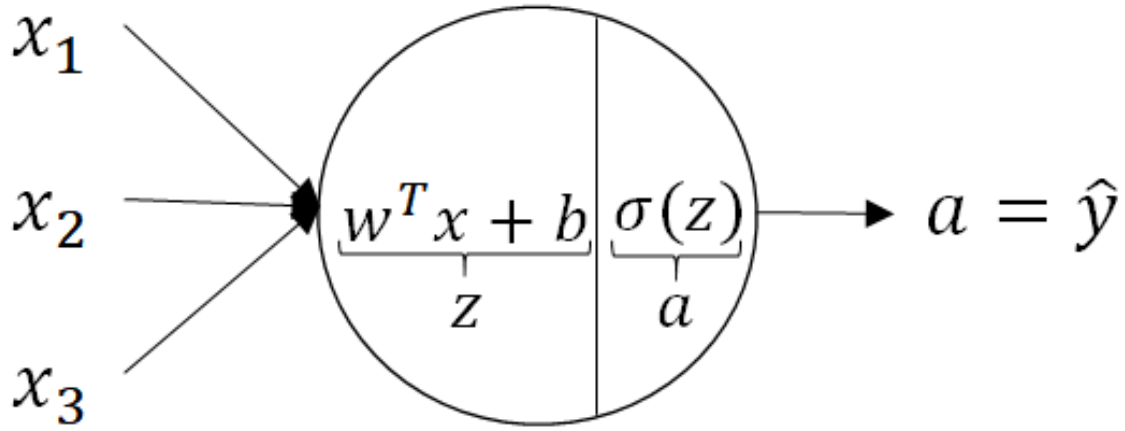


Figure 1.4: Neural Network Representation of Logistic Regression

dB be a $1 \times m$ dimensional row vector ($dB = [db^{(1)}, ..., db^{(m)}]$).

dW be a $1 \times m$ dimensional row vector ($dW = [dw^{(1)}, ..., dw^{(m)}]$).

Then:

$dZ = A - Y$

$dB = \frac{1}{m} \sum(dZ)$

$$dW = \frac{1}{m} \sum(X \cdot dZ^T) = \frac{1}{m} \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix} \cdot \begin{bmatrix} dz^{(1)} \\ dz^{(2)} \\ \vdots \\ dz^{(m)} \end{bmatrix}$$

An implementation of this in cat image recognition can be found at: Documents/Python/DeepLearning/Assignment_1-Logistic_Regression_2

1.3 Standard Neural Networks

Neural Networks are like logistic regression repeated a bunch of times. Compare the neural net representation of a logistic regression compared to a the representation of a true neural net (Figure 1.4 and 1.5).

1.3.1 Neural Network Notation

For the most part, the notation used in 1.2.2. Notation for logistic regression applies here.

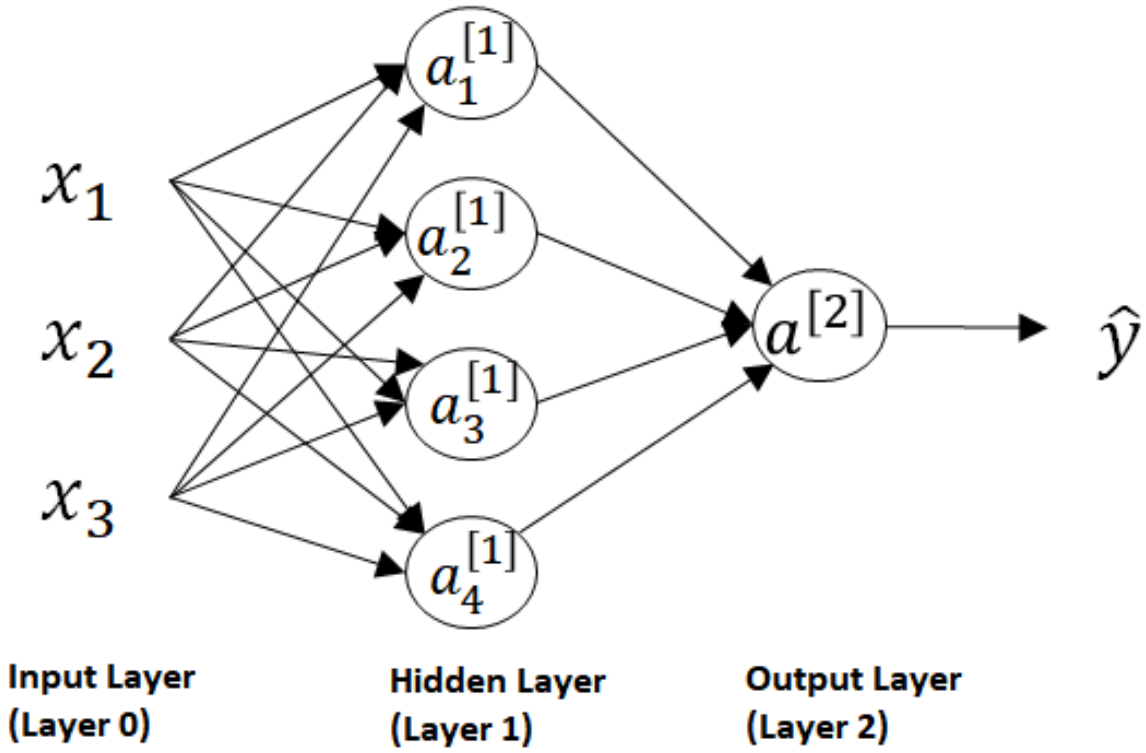


Figure 1.5: Shallow Neural Network Representation

Neural Network Notation

- Each circle is a 'node' or a 'unit', and is essentially a feature.
- Each column of nodes is a layer.
- The input layer is the 0^{th} layer.
- The first hidden layer is the 1^{st} layer, etc.
- The output layer is also number (e.g., layer 2 in a neural net with 1 hidden layer)

Mathematical Notation

- n_x : Vector containing the number of units in each layer.

  e.g., $n_x = [n^{[0]}, n^{[1]}, n^{[2]}, ...]$
- $a_j^{[l]}$: Activation value for the j^{th} unit of the l^{th} layer.
- $A^{[l]}$: Activation vector for the $l^{[th]}$ layer; has the dimensions

  $(n^{[l-1]}) \times (m)$.

- $x_j^{[l](i)}$: The x value for the j^{th} unit of the l^{th} layer for the i^{th} training example.
- $W^{[l]}$: The weight matrix for the l^{th} layer; has the dimensions

  $(n^{[l]}) \times (n^{[l-1]})$.
- $b^{[l]}$: The bias vector for the l^{th} layer; has the dimensions

  $(n^{[l]}) \times 1)$, but is broadcasted to $(n^{[l]}) \times m)$ by Python
- $Z^{[l]}$: The linear combination of $W^{[l]} * a^{[l-1]} + B^{[l]}$ for the l^{th} layer; has the dimensions

  $(n^{[l]}) \times m$.

Alternate Notation

- $x = a^{[0]}$.
- $\hat{y} = a^{[2]}$, which is equivalent to our use of "$\hat{y} = a$" in logistic regression.

1.3.2 Computing the Neural Net Activation for a Single Example

The neural net represented in figure 1.5 will be used to illustrate how to compute the activation of a neural net.

Neural net activation is just logistic regression with more layers and more nodes. We can easily show that:

$a_1^{[1](i)} = \sigma(z_1^{[1](i)}) = \sigma(w_1^{[1](i)T} x + b_1^{[1](i)})$

$a_2^{[1](i)} = \sigma(z_2^{[1](i)}) = \sigma(w_2^{[1](i)T} x + b_2^{[1](i)})$

$a_3^{[1](i)} = \sigma(z_3^{[1](i)}) = \sigma(w_3^{[1](i)T} x + b_3^{[1](i)})$

$a_4^{[1](i)} = \sigma(z_4^{[1](i)}) = \sigma(w_4^{[1](i)T} x + b_4^{[1](i)})$

We can represent this much more efficiently. For example, we can 'stack up' each of the w vectors and b scalars into rows and come up with:

$$W^{[1](i)} = \begin{bmatrix} . & w_1^{[1](i)T} & . \\ . & w_2^{[1](i)T} & . \\ . & w_3^{[1](i)T} & . \\ . & w_4^{[1](i)T} & . \end{bmatrix} \qquad B^{[1](i)} = \begin{bmatrix} b_1^{[1](i)} \\ b_2^{[1](i)} \\ b_3^{[1](i)} \\ b_4^{[1](i)} \end{bmatrix}$$

$W^{[1]}$ has dimensions $(4, 3)$ and $B^{[1]}$ with dimensions $(4, 1)$. Note that when we do this, we no longer need to transpose W in order to get the dot product with X.

A Vectorized Implementation

With the parameters w and b now represented in matrix/vector form, one can to show that a vectorized computation of Z is possible.

$$Z^{[1](i)} = \begin{bmatrix} z_1^{[1](i)} \\ z_2^{[1](i)} \\ z_3^{[1](i)} \\ z_4^{[1](i)} \end{bmatrix} = \begin{bmatrix} . & w_1^{[1](i)T} & . \\ . & w_2^{[1](i)T} & . \\ . & w_3^{[1](i)T} & . \\ . & w_4^{[1](i)T} & . \end{bmatrix} \cdot \begin{bmatrix} x_1^{(i)} \\ x_2^{(i)} \\ x_3^{(i)} \end{bmatrix} + \begin{bmatrix} b_1^{[1](i)} \\ b_2^{[1](i)} \\ b_3^{[1](i)} \\ b_4^{[1](i)} \end{bmatrix} F$$

$$Z^{[1](i)} = \begin{bmatrix} w_1^{[1](i)T} x + b_1^{[1](i)} \\ w_2^{[1](i)T} x + b_2^{[1](i)} \\ w_3^{[1](i)T} x + b_3^{[1](i)} \\ w_4^{[1](i)T} x + b_4^{[1](i)} \end{bmatrix}$$

$$a^{[1](i)} = \begin{bmatrix} a_1^{[1](i)} \\ a_2^{[1](i)} \\ a_3^{[1](i)} \\ a_4^{[1](i)} \end{bmatrix} = \sigma(Z^{[1](i)})$$

1.3.3 Computing the Neural Net Activation for Many Examples

In the previous section, we went over how to vectorize activation for a given layer for a single training example. An inefficient approach might loop the steps above for all of 1:m training examples. However, we can implement a vectorized approach:

Let:

$$X = A^{[0]} \begin{bmatrix} \vdots & \vdots & & \vdots \\ x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}, \text{ a feature matrix with dimensions } (n^{[0]}, m)$$

$$Z^{[1]} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ z^{[1](1)} & z^{[1](2)} & \dots & z^{[1](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}, \text{ with dimensions } (n^{[1]}, m)$$

$$A^{[1]} = \begin{bmatrix} \vdots & \vdots & & \vdots \\ a^{[1](1)} & a^{[1](2)} & \dots & a^{[1](m)} \\ \vdots & \vdots & & \vdots \end{bmatrix}, \text{ with dimensions } (n^{[1]}, m)$$

(Note that in our 'sideways' matrix representation our features line up with the way we represent our neural network units (i.e., features))

Then, the entire neural net can be calulated by:

$$Z^{[1]} = W^{[1]} A^{[0]} + B^{[1]} \qquad A^{[1]} = \sigma(Z^{[1]})$$

$$Z^{[2]} = W^{[2]} A^{[1]} + B^{[2]} \qquad A^{[2]} = \sigma(Z^{[2]})$$

1.3.4 Activation Functions

Sigmoid Activation

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

$$g'(z) = g(z)(1-g(z)$$

$$g'(z) = a(1-a)$$

The sigmoid function is useful for binary classification. Otherwise, it is a slow learner and is outclassed by the *tanh* function.

Tanh Activation

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

$$g'(z) = 1 - (\tanh(z))^2$$

$$g'(z) = 1 - a^2$$

Tanh is a shifted version of the sigmoid function. It is preferable in hidden layers because it has a mean of zero, and so it centres the data better for the next layer (i.e., faster learning)

ReLU Activation

$$g(z) = \max(0, z)$$

$$g'(z) = \begin{cases} 0, & \text{if } z < 0 \\ 1, & \text{if } z > 0 \end{cases}$$

The ReLU activation function has a derivative 1 if $z > 0$ and has a derivative 0 otherwise. This leads to fast learning, and ReLU is popular for this reason.

Technically, the $\frac{dg(z)}{dz}$ is undefined when $z = 0$, but we can implement our code so that the derivative is 0 in such edge cases (or you could make it 1, it really doesn't matter).

Leaky ReLU Activation

$$g(z) = \max(0.01z, z)$$

$$g'(z) = \begin{cases} 0.01, & \text{if } z < 0 \\ 1, & \text{if } z > 0 \end{cases}$$

The Leaky ReLU function exists to deal with cases where a derivative exactly equal to 0 will mess things

up. I am not sure of practical situations where this might come up, though.

1.3.5 Gradient Descent for Neural Networks

In our $3 \times 4 \times 1$ dimensional neural network in Figure 1.5, we have:

The parameters: $W^{[1]}, b^{[1]} W^{[2]}, b^{[2]}$

The cost function: $J(W^{[1]}, b^{[1]} W^{[2]}, b^{[2]})$

We initialize our parameters randomly and then compute gradient descent:

Repeat:

Compute $\hat{y}^{(i)}$ for $i = 1 : m$

Compute the derivatives:

$dZ^{[2]} = A^{[2]} - Y$ (note: this assume a logistic activation)

$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]T}$

$db^{[2]} = \frac{1}{m}$`np.sum(`$dZ^{[2]}$`, axis = 1, keepdims = True)`

$dZ^{[1]} = dW^{[2]T} dZ^{[2]} \times g^{[1]'}(Z^{[1]})$ (note: this is element-wise multiplication)

$dA^{[1]} = dW^{[2]T} dZ^{[2]}$

$dZ^{[1]} = dA^{[1]} \times g^{[1]'}(Z^{[1]})$

$dW^{[1]} = \frac{1}{m} dZ^{[1]} A^{[0]T}$

$db^{[1]} = \frac{1}{m}$`np.sum(`$dZ^{[1]}$`, axis = 1, keepdims = True)`

Update the parameters:

$W^{[1]} := W^{[1]} - \alpha dW^{[1]}$

$b^{[1]} := W^{[1]} - \alpha db^{[1]}$

$W^{[2]} := W^{[2]} - \alpha dW^{[2]}$

$b^{[2]} := b^{[2]} - \alpha db^{[2]}$

Note: `keepdims = True` ensures that db does not end up as a rank 1 numpy array with dims $(m,)$.

1.3.6 Initializing Parameters

Unlike logistic regression, we can't initialize W to be all zeroes. It is possible to show that using a matrix of zeroes for your W parameter results in hidden layer units that are identical to each other (i.e., symmetric) and that subsequent layers are computing the exact same function as the first.

Instead we should initialize parameters randomly:

```
W1 = np.random.randn((4, 3)) * 0.01
b1 = np.zero((4, 1))
W2 = np.random.randn((1, 4)) * 0.01
b2 = np.zero((1, 1))

# We multiply by 0.01 in order to get a smaller number. For functions like tanh or sigmoid, this
    helps keep the initial parameters close to the middle where the slopes are greater (i.e.,
    learn faster)
```

1.3.7 Deep Neural Networks

Deep neural nets have the same implementation as shallow neural nets, except there are more hidden layers in the middle. It also becomes tedious to compute derivatives for back propagation by hand, as the chaining becomes increasingly complex as you increase layers.

Initializing Parameters

```
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer in our network

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":
    Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
    bl -- bias vector of shape (layer_dims[l], 1)
    """


    parameters = {}
    L = len(layer_dims)         # number of layers in the network

    for l in range(1, L):
    parameters['W' + str(l)] = np.random.rand(layer_dims[l], layer_dims[l - 1]) * 0.01
    parameters['b' + str(l)] = np.zeros(shape=(layer_dims[l], 1))

    return parameters
```

Conceptualizing gradient descent for deep networks

Figure 1.6 provides an outline for how we perform gradient descent. In pseudo-code this might look like:

```
Initialize parameters [W1, ..., WL] and [b1, ..., bL]

# Forward Propagation:

# Forward prop for all layers 1 to L-1, which use ReLU
Loop for each layer 1:(L-1):
```

```
   Compute the forward linear combination (Z)
      cache Z, A_previous, W, b
   Compute the forward activation (A) via ReLU using cache
      Add A to the cache

# Forward prop for layer L, which uses sigmoid
Compute AL (aka yhat) via sigmoid using A_L-1, WL, bL
(Optional) Compute the Cost

# Backward Propagation:

# Backprop for layer L, which uses sigmoid
dW = 1/m * np.dot(dZ, A[l-1].T)
db = 1/m * np.sum(dZ, axis=1, keepdims = True)
dA_previous = np.dot(W.T, dZ)

dZ = dA_previous * g*(Z) # Where g(Z) is sigmoid

# Backprop for layers L-1 to 1, which use ReLU
Loop for each layer l in (L-1):1:
   # Compute dL/dA (aka dA) for the layer (l - 1):
   dW = 1/m * np.dot(dZ, A[l-1].T)
   db = 1/m * np.sum(dZ, axis=1, keepdims = True)
   dA_previous = np.dot(W.T, dZ) # dA_previous is dA[l-1] where dW and db are layer [l]
      cache dA_previous, dW, db

   # Compute dA/dZ (aka dZ) for the layer (l - 1):
   dZ = dA_previous * g*(Z) # Where g(Z) is sigmoid

# Update parameters
Loop for each layer l in 1 to L:
   parameters['W' + str(l)] -= alpha * gradients['dW' + str(l)]
   parameters['b' + str(l)] -= alpha * gradients['db' + str(l)]
```

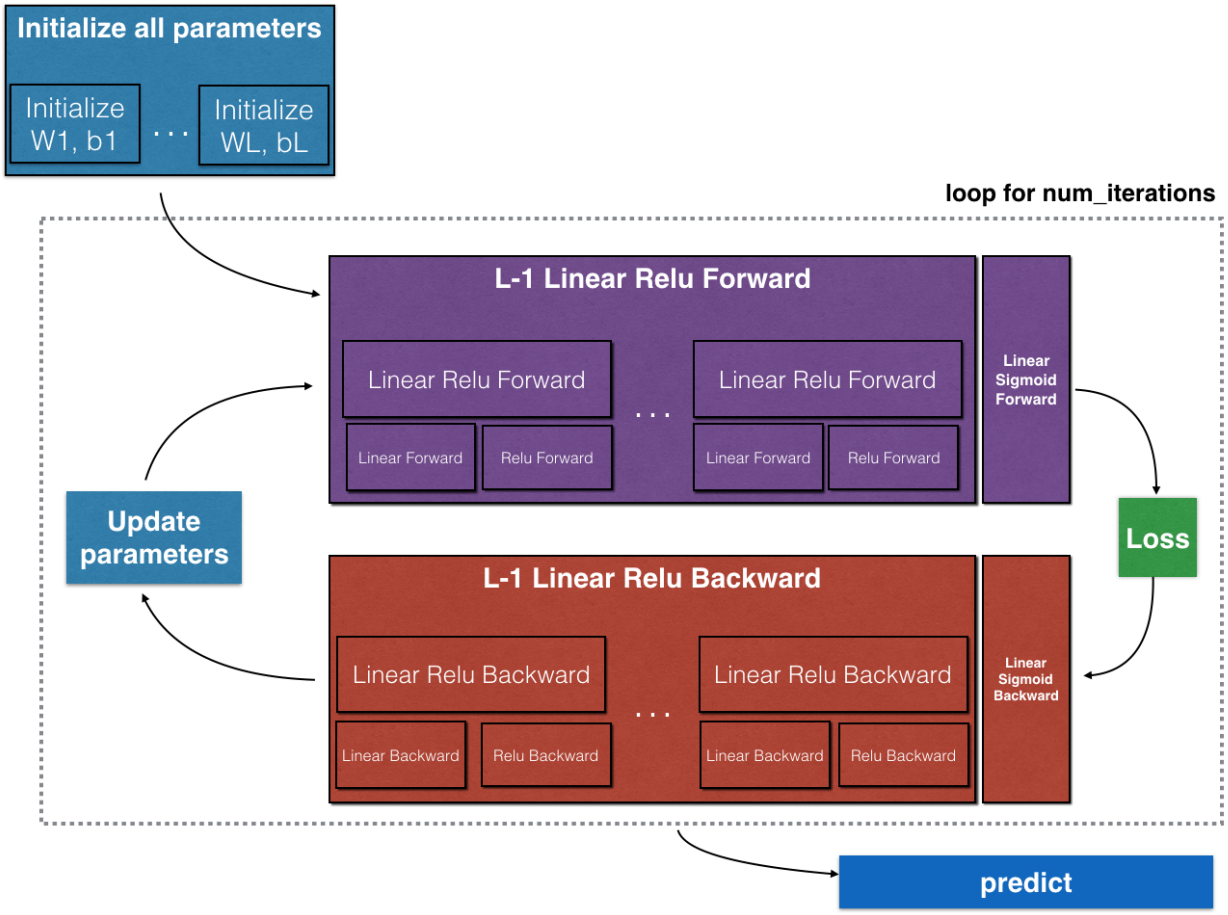


Figure 1.6: Schematic of gradient descent

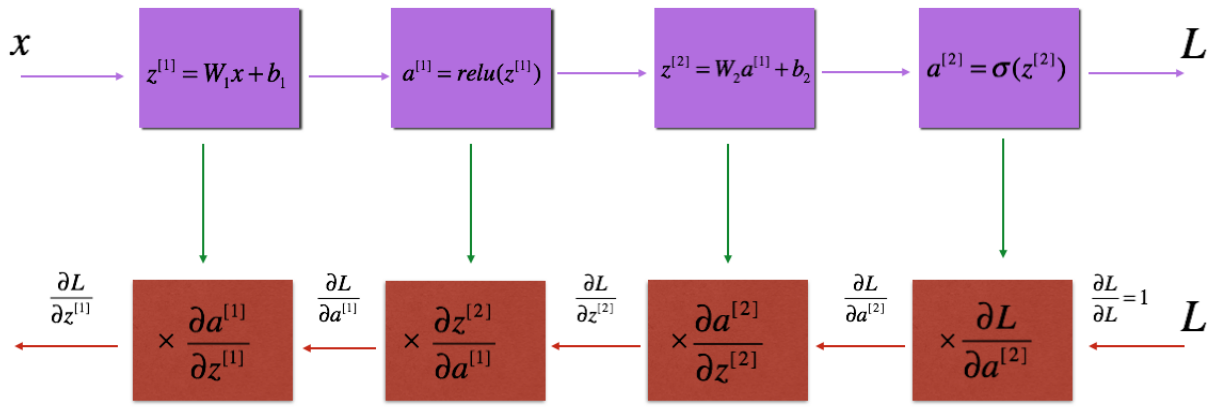


Figure 1.7: Steps in back propagation

Chapter 2

Course II: Improving Neural Networks

Applied ML is an interative process.

- There are a number of hyperparameters that need to be tuned.
- Intutions and expertise from one application in deep learning do not necessarily apply to other applications. The optimal hyperparameters vary by problem, data, and computer configuration.
- The speed of the pipeline is determined by how quickly you can cycle from idea to code to experiment.

2.1

2.1.1 Dividing data sets

In traditional ML, best practice is to divide data sets into training, CV, and test sets (60%, 20%, 20%). For example, you may train several models and evaluate their CV error. After picking the best one, you would test the model against the test set to get an unbiased estimate of the true error. This makes sense when data sets are small to medium sized. However, when data sets become large (e.g., ¿ 1,000,000 observations), a smaller proportion of the data need to be used for CV and testing. For a data set with 1,000,000 observations, only 10,000 (1%) of the data need to be used in each of the CV and test sets (for example).

Chapter 3

Math Review

3.1 Matrix Multiplication

Because I keep forgetting.

$$\begin{bmatrix} a_{1,1} & a_{1,2} \\ a_{2,1} & a_{3,2} \\ a_{3,1} & a_{3,2} \end{bmatrix} \times \begin{bmatrix} b_{1,1} & b_{1,2} & b_{1,3} \\ b_{2,1} & b_{2,2} & b_{2,3} \end{bmatrix} = \begin{bmatrix} a_{1,1} \times b_{1,1} + a_{1,2} \times b_{2,1} & a_{1,1} \times b_{1,2} + a_{1,2} \times b_{2,2} & a_{1,1} \times b_{1,3} + a_{1,2} \times b_{3,1} \\ a_{2,1} \times b_{1,1} + a_{2,2} \times b_{2,1} & a_{2,1} \times b_{1,2} + a_{2,2} \times b_{2,2} & a_{2,1} \times b_{1,3} + a_{2,2} \times b_{3,1} \\ a_{3,1} \times b_{1,1} + a_{3,2} \times b_{2,1} & a_{3,1} \times b_{1,2} + a_{2,2} \times b_{2,2} & a_{3,1} \times b_{1,3} + a_{3,2} \times b_{3,1} \end{bmatrix}$$

Chapter 4

Python

4.1 Useful Built-ins

4.2 Numpy Functions

`np.array([[..., ...], [..., ...]])`: Generates an array (i.e., vector, matrix, or higher order array). Each dimension is enclosed in '`[ ]`'.

`np.random.randn(m, n)`: Generates a data structure of random numbers with dimensions `m` by `n`. (Note: never use rank 1 arrays (i.e., missing a dimension))

`x.shape`: Attribute that retrieves dimensions of data structure `x`.

`x.reshape(m, n), ...`: Reshapes a data structure to the dimensions `m` by `n` by `...`. Used in image-to-vector transformation (im2vec)

`x.squeeze`: Reshapes an array with empty dimensions to a lower order data structure (ex., an array [[17]] will become an int 17).

`x.ravel`: Flattens an array into a vector (i.e., a contiguous rank 1 array)

```
def image2vector(image):
'''
Argument: image -- a numpy array of shape (length, height, depth)

Returns: v -- a vector of shape (length*height*depth, 1)
'''

v = image.reshape((image.shape[0] * image.shape[1]* image.shape[2]), 1)
return v
```

Math Functions

A lot of functions in the `math` package have vectorized counterparts in numpy. For example:

`np.dot(x, y)`: Computes the dot product for the matrices $x \times y$.

`np.multiply(x, y)`: Computes element-wise multiplcation of x by y (same as the `*` in `math`.

`np.exp(x)`: Computes e^x for all elements in x.

`np.sum(x, axis = 0, keepdims = False)`: Computes the sum of x. Can select row-wise, column-wise sums, etc. depending on axis (0 = column-wise).

4.3 Time

`time.process_time()` : Records the system time. Can be used to benchmark run-time