Deep Learning

Labix

January 12, 2024

Abstract

Contents

1 Foundations of Deep Learning

Deep learning involves mostly neural networks and using given inputs and outputs to find out the intermediate process that determines the relation. In other words, we are given a set of inputs and a set of outputs, and we want to find out the function relating these sets.

Deep learning is used in a lot of different places such as photo recognition, speech recognition and real state price estimations. There are different neural network models that allows a more accurate prediction for different data structures and maybe specific to the questions itself.

Inputs can be roughly separated into two types: Structured data and unstructured data. Structured data usually can be aligned into a table while unstructured data include audio, images and texts.

Deep learning is now taking off mainly because of the size of data that we possess as compared to a few years ago.

1.1 Neural Networks

We begin with the notion of a node.

Definition 1.1.1: Nodes

A node takes a vector x of n inputs and outputs a single value a using the follow equation

$$\varphi(wx+b)$$

where w, b, φ are called parameters of the node. w is a $1 \times n$ matrix and b is a real number. $\varphi : \mathbb{R} \to \mathbb{R}$ is called an activation function.

Definition 1.1.2: Layers

A layer is a collection of ordered nodes of the same depth. The depth is denoted l, which means that it takes l steps to transform all the initial inputs to a node in layer l.

Layer 0, consisting of all the initial inputs are called the input layer. The final layer L in which there are no further outputs for all nodes are called the output layer is called the output layer. The rest are called the hidden layer.

A neural network consists of inputs, intermediate nodes and outputs. It is also further separated into layers. The input nodes are grouped to called the input layer, similarly for the output layer. The rest of the nodes are part of the hidden layer. Nodes connected immediately from the input nodes are the first layer, nodes connected immediately from the first layer forms the second layer and so on. They are usually called hidden layer 1, hidden layer 2 and so on.

Suppose that on layer l, there are $n^{[l]}$ nodes. Each node has its own w and b labeled $w_1^{[l]}, \dots, w_{n^{[l]}}^{[l]}$ and similarly for b, which allows us to form the matrices

$$W^{[l]} = \begin{pmatrix} | & \cdots & | \\ w_1^{[l]} & \cdots & w_{n^{[l]}}^{[l]} \\ | & \cdots & | \end{pmatrix}^T \quad \text{and} \quad b^{[l]} = \begin{pmatrix} b_1^{[l]} & \cdots & b_1^{[l]} \\ \vdots & \ddots & \vdots \\ b_{n^{[l]}}^{[l]} & \cdots & b_{n^{[l]}}^{[l]} \end{pmatrix}$$

respectively. (We will see the dimensions of the matrices below)

We denote the training examples as

$$\{(x^{(1)}, y^{(1)}, \dots, (x^{(m)}, y^{(m)}))\}$$

where m is the total number of training data. Each $x^{(i)}$ is the input and $y^{(i)}$ is the expected output for $1 \leq i \leq m$. If the neural network has $n^{[0]}$ inputs, then each $x^{(i)}$ is a $n^{[0]} \times 1$ vector. Similarly, if the neural network has $n^{[L]}$ outputs then each $y^{(i)}s$ is a $n^{[L]} \times 1$ vector.

We can concatenate these $n^{[0]} \times 1$ vectors into the training data matrix:

$$A^{[0]} = \begin{pmatrix} | & \cdots & | \\ x^{(1)} & \cdots & x^{(m)} \\ | & \cdots & | \end{pmatrix}$$

Through this, we can easily batch process an entire training data set using the matrix multiplication

$$Z^{[1]} = W^{[1]}A^{[0]} + b^{[1]}$$

and further applying the activation function to get

$$A^{[1]} = \varphi(Z^{[1]})$$

$A^{[1]}$ in this case is of dimension $n^{[1]} \times 1$. If we inductively apply the process, we can write the following:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$

and

$$A^{[l]} = \varphi(Z^{[l]})$$

The dimensions of the matrices are as follows:

Theorem 1.1.3

Denote l the layer of a neural network. Then $W^{[l]}$ has dimension $n^{[l]} \times n^{[l-1]}$. $b^{[l]}$ has dimension $n^{[l]} \times m$. $Z^{[l]}$ and $A^{[l]}$ both has dimension $n^{[l]} \times m$.

Deep learning involves providing a specified neural network (giving it L layers and $n^{[l]}$ nodes for each layer l) some training data and outputs the linear regression matrix $W^{[l]}$ and bias $b^{[l]}$ on each layer l. Assuming we found out the best W and b for each layer, we use this calculation to calculate our own predicted output, and compare it with the supposed output (provided along with the training data) and find out how big the difference it with a lost function. The goal is then simple: to minimize the lost function, which means to optimize it and find its local minimum.

Definition 1.1.4: Loss Function

The loss function of a neural network calculates the difference between the predicted value of the network and the actual value of in the training data. It is a function which depends on $W^{[l]}$ and $b^{[l]}$ for $1 \leq l \leq L$. It is denoted as

$$\mathcal{L}(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}, Y)$$

Below is a series of code used to implement the calculation of our predicted value. The following code implements the linear part of the of each node.

Algorithm 1.1.5: Linear Forward

Implements the linear part of a layer's forward propagation.

Arguments:
A – activations from previous layer (or input data): (size of previous layer, number of examples)
W – weights matrix: numpy array of shape (size of current layer, size of previous layer)
b – bias vector, numpy array of shape (size of the current layer, 1)

Returns:
Z – the input of the activation function, also called pre-activation parameter
cache – a python tuple containing "A", "W" and "b" ; stored for computing the backward pass efficiently

```python
def linear_forward(A, W, b):

    Z = np.dot(W,A)+b
    cache = (A, W, b)

    return Z, cache
```

1.2 Binary Classification

Binary classification refers to the fact that the outputs are binary. One example is to use deep learning to identify cats. In this case, the output consists of either 1, which means that the picture is a cat, or 0 if not.

Recall that in linear regression, we calculate our estimation of the output $\hat{y}$ using the following equation:

$$\hat{y} = w \cdot x + b$$

where $w \in M_{1\times n^{[0]}}(\mathbb{R})$ is some vector that we are trying to improvise, and b is the bias which we are also trying to improvise. Assuming that our outputs are binary, $\hat{y}$ should be a probability

$$\hat{y} = \mathbb{P}(y = 1|x)$$

so that it generates an estimation between 0 and 1. Unfortunately this does not work using the regular equation for linear regression. Instead, we input the result $z = w \cdot x + b$ further into an activation function. So that the output can 1) be between 0 and 1 in this special case but also more importantly, 2) so that the neural network does not reduce to a simple case of 1 node.

To elaborate, notice that if we take a linear function for the activation function, say the identity, by composing $A^{[l]}$, we get

$$A^{[L]} = W^{[L]}W^{[L-1]}\cdots W^{[1]}A^{[0]} + B$$

This is clearly just the same expression as $z = w \cdot x + b$ which means that no matter how many layers we define the neural network, the network reduces to a one layer network.

The remainder of the codes of the section will be dedicated to how to program an L layer with binary classification at the last layer. The regression model for a binary classification model is called a logistic regression.

1.3 Random Initialization

W and b have to be initialized in order for the first iteration to succeed. However, if the weights are all set to 0, the layer in which the nodes are in will reduce to just being 1 node due to the fact that they have the same weight, and so what each node does is identical.

The code for random initialization is given below:

Algorithm 1.3.1: Initialize Parameters Deep

Program to initialize all weights and bias.

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)

```python
def initialize_parameters_deep(layer_dims):

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims) # number of layers in the network

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

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[l -
            1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))

    return parameters
```

The factor of 0.01 is there to make sure that the weights are close to 0. As seen in the graph of sigmoid, for large weights, z becomes large and thus the gradient becomes close to 0, making gradient descent (the optimization technique slow).

Moreover, bias take no effect in random initialization nor are they penalized in the algorithm.

1.4 The First Hypervariables

Definition 1.4.1: Hypervariables

The hypervariables of a program refers to variables that controls the output of other variables which further affects the output of the program.

Namely, the activation function and the lost function are both hypervariables of a neural network. They control what W and b will look like for each node. Common choices of activation functions include the following

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

$$\text{ReLU}(z) = \max\{0, z\}$$

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

Commonly, we will use ReLU for its far from 0 derivative (as we will see, lost functions are optimized by using gradient descent, which depends on the derivative). However, in the case of binary classification, we may want to use the sigmoid function so that its outputs are between 0 and 1. tanh are sometimes useful because the the function lies between -1 and 1 so that the mean of the data can be closer to 0.

The following code the function of a single node, using the linear forward module.

Algorithm 1.4.2: Linear Activation Forward

Implements the forward propagation for the LINEAR to ACTIVATION layer.
Arguments:
A_prev – activations from previous layer (or input data): (size of previous layer, number of examples)
W – weights matrix: numpy array of shape (size of current layer, size of previous layer)
b – bias vector, numpy array of shape (size of the current layer, 1)
activation – the activation to be used in this layer, stored as a text string: ”sigmoid” or ”relu”

Returns:
A – the output of the activation function, also called the post-activation value
cache – a python tuple containing "linear_cache" and "activation_cache"; stored for computing the backward pass efficiently

```python
def linear_activation_forward(A_prev, W, b, activation):

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)

    return A, cache
```

We then use linear activation forward to calculate the final prediction result using the following module:

Algorithm 1.4.3: L Model Forward

Implements forward propagation for the [LINEAR $\rightarrow$ RELU]*$(L-1)$ $\rightarrow$ LINEAR $\rightarrow$ SIGMOID computation.

Arguments:
X – data, numpy array of shape (input size, number of examples)
parameters – output of initialize_parameters_deep()

Returns:
AL – activation value from the output (last) layer
caches – list of caches containing: every cache of linear_activation_forward() (there are L of them, indexed from 0 to $L-1$)

```python
def L_model_forward(X, parameters):

    caches = []
    A = X
    L = len(parameters) // 2            # number of layers in the neural network

    # The first n-1 layers using ReLU
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters["W" + str(l)],
        ↪   parameters["b" + str(l)], "relu")
        caches.append(cache)

    # The output layer using sigmoid
    AL, cache = linear_activation_forward(A, parameters["W" + str(L)],
    ↪   parameters["b" + str(L)], "sigmoid")
    caches.append(cache)

    return AL, caches
```

Similarly, there is a choice for the loss function. Loss functions are calculated independently from each training set. They are combined into an average $J = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}$. The square error

$$\mathcal{L} = \frac{1}{2}(\hat{Y} - Y)^2$$

is often not useful in linear regression due to having no vertex. It is common to use the loss function

$$\mathcal{L} = -(Y\log\left(\hat{Y}\right) + (1-Y)\log\left(1-\hat{Y}\right))$$

for logistic regression.

It is easy to see that if a training data $y = 1$, then $\mathcal{L} = -\log(\hat{y})$ which means that our optimization should be based on $\hat{y}$ being as large as possible. Since $0 \le \hat{y} \le 1$ we must have $\hat{y} = 1$ for best optimization. Similarly, if $y = 0$, $\mathcal{L} = -\log(1-\hat{y})$ will result in optimization towards $\hat{y} = 0$.

We finally have the cost function.

Definition 1.4.4: Cost Function

The most common cost function for deep learning is given by

$$J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}, Y) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{Y}, Y)$$

($\hat{Y}$ is determined by $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ so that this notation makes sense)

The cost function for a single iteration is calculated as below, given the prediction:

Algorithm 1.4.5: Compute Cost

Implements the cost function defined as above.

Arguments:
AL – probability vector corresponding to your label predictions, shape (1, number of examples)
Y – true "label" vector (for example: containing 0 if non-cat, 1 if cat), shape (1, number of examples)

Returns:
cost – cross-entropy cost

```python
def compute_cost(AL, Y):

    m = Y.shape[1]

    cost = (-1/m) * np.sum(np.multiply(Y, np.log(AL)) + np.multiply(1-Y,
    ↪   np.log(1-AL)))

    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect

    return cost
```

1.5 Optimization via Gradient Descent

Optimization of the loss function involves finding the minimum of the function. Since J is a convex function, we can use the method of gradient descent.

Definition 1.5.1: Gradient Descent

For a function J with partial variable w, we find the local optima with respect to w using the formula

$$w_{n+1} = w_n - \alpha \frac{\partial J(w)}{\partial w}\bigg|_{w=w_n}$$

where $0 \leq \alpha \leq 1$ is called the learning rate. In matrix form this is written as

$$W^{[l]}_{n+1} = W^{[l]}_n - \alpha \frac{\partial J(W^{[l]}}{\partial W^{[l]}}\bigg|_{W^{[l]}=W^{[l]}_n}$$

n here indicates the nth iteration.

The learning rate is another hypervariable that we will come and modify later. We will repeatedly obtain new values of W and b for every node and run more iterations to fine tune their values, hence explaining why iteration appears in the definition.

The process of finding $A^{[l]}$ for each layer is called forward propagation for its iterative use of $A^{[l-1]}$, while finding $\frac{\partial J(w)}{\partial w}|_{w=w_n}$ is called backwards propagation for its iterative use of $\frac{\partial J(w)}{\partial w}|_{w=w_{n-1}}$ AFTER computing $A^{[L]}$.

We establish the following notation.

Definition 1.5.2

Suppose there is a neural network with L layers. Suppose there are $n^{[l]}$ nodes on the lth layer. Denote

$$\frac{\partial \mathcal{L}}{\partial W^{[l]}} = \begin{pmatrix} | & & | \\ \frac{\partial \mathcal{L}}{\partial w^{[l]}_1} & \cdots & \frac{\partial \mathcal{L}}{\partial w^{[l]}_{n^{[l]}}} \\ | & & | \end{pmatrix}^T$$

for the gradient of $\mathcal{L}$ with respect to $w^{[l]}$, where

$$\frac{\partial \mathcal{L}}{\partial w^{[l]}_k} = \begin{pmatrix} \frac{\partial \mathcal{L}}{\partial (w^{[l]}_k)_1} \\ \vdots \\ \frac{\partial \mathcal{L}}{\partial (w^{[l]}_k)_{n^{[l-1]}}} \end{pmatrix} = \nabla_{w^{[l]}_k} \mathcal{L}$$

In particular, the dimension of $\frac{\partial \mathcal{L}}{\partial W^{[l]}}$ is $(n^{[l]}, n^{[l-1]})$. The notation is similar for the gradient of J.

In one layer neural networks, we have the following gradient:

Proposition 1.5.3

For the loss function $\mathcal{L}$ in a neural network with one layer with m training data, the gradient is given by

$$\frac{\partial J}{\partial W} = \frac{1}{m}\frac{\partial \mathcal{L}}{\partial Z}(A^{[0]})^T$$

The gradient for the bias vector is given by

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\partial \mathcal{L}}{\partial Z}\right)_i$$

where the sum runs through the elements of the $1 \times m$ vector. (Sanity check: $\frac{\partial \mathcal{L}}{\partial Z}$ is a $1 \times m$ matrix, and $\frac{\partial J}{\partial W}$ has the same dimension as W).

In multiple layers, the gradient is given partially by the following equations:

Proposition 1.5.4

For the cost function $\mathcal{L}$ in a neural network with L layers and m training data, the gradients on the lth layer with activation function φ are given by

$$\frac{\partial J}{\partial W^{[l]}} = \frac{1}{m}\frac{\partial \mathcal{L}}{\partial Z^{[l]}}\left(A^{[l-1]}\right)^T$$

$$\frac{\partial J}{\partial b^{[l]}} = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{\partial \mathcal{L}}{\partial Z^{[l]}}\right)_i$$

where the sum are taken over the rows so that $\frac{\partial J}{\partial b^{[l]}}$ has dimension $n^{[l]} \times 1$ and

$$\frac{\partial \mathcal{L}}{\partial Z^{[l]}} = \frac{\partial J}{\partial A^{[l]}} * \varphi'(Z^{[l]})$$

with $*$ being elementwise multiplication and

$$\frac{\partial J}{\partial A^{[l-1]}} = (W^{[l]})^T\frac{\partial \mathcal{L}}{\partial Z^{[l]}}$$

(Sanity check: Recall that both $Z^{[l]}$ and $A^{[l]}$ are both $n^{[l]} \times m$ matrices. $\frac{\partial J}{\partial A^{[l-1]}}$ would have dimension $n^{[l]} \times m$ as well)

The following code calculates part of the gradient (without $\frac{\partial J}{\partial z}$) at each iteration. It must be done after forward propagation since it makes use of its intermediate values.

Algorithm 1.5.5

Implements the linear portion of backward propagation for a single layer (layer l).

Arguments:
dZ – Gradient of the cost with respect to the linear output (of current layer l)
cache – tuple of values (A_prev,W,b) coming from the forward propagation in the current layer

Returns:
dA_prev – Gradient of the cost with respect to the activation (of the previous layer $l-1$), same shape as A_prev
dW – Gradient of the cost with respect to W (current layer l), same shape as W
db – Gradient of the cost with respect to b (current layer l), same shape as b

```python
def linear_backward(dZ, cache):

    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = (1/m)*(np.dot(dZ,A_prev.T))
    db = (1/m)*(np.sum(dZ, axis=1, keepdims=True))
    dA_prev = np.dot(W.T,dZ)

    return dA_prev, dW, db
```

Helper functions:

Algorithm 1.5.6

We can now combine the backward propagation step of each activation function into the linear activation backward module, which computes our required gradient:

Algorithm 1.5.7

Implements the backward propagation for the LINEAR→ACTIVATION layer.

Arguments:
dA – post-activation gradient for current layer l
cache – tuple of values (linear_cache, activation_cache) we store for computing backward propagation efficiently
activation – the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

Returns:
dA_prev – Gradient of the cost with respect to the activation (of the previous layer $l-1$), same shape as A_prev
dW – Gradient of the cost with respect to W (current layer l), same shape as W
db – Gradient of the cost with respect to b (current layer l), same shape as b

```python
def linear_activation_backward(dA, cache, activation):

   linear_cache, activation_cache = cache

   if activation == "relu":
      dZ = relu_backward(dA, activation_cache)
      dA_prev, dW, db = linear_backward(dZ, linear_cache)

   elif activation == "sigmoid":
      dZ = sigmoid_backward(dA, activation_cache)
      dA_prev, dW, db = linear_backward(dZ, linear_cache)

   return dA_prev, dW, db
```

Recalling that are last layer uses the sigmoid activation function and the rest using ReLU, we implement the backwards propagation using the above module:

Algorithm 1.5.8: L Model Backwards

Implements the backward propagation for the [LINEAR→RELU] * $(L-1)$ → LINEAR → SIGMOID group.

Arguments:
AL – probability vector, output of the forward propagation (L_model_forward())
Y – true "label" vector (containing 0 if non-cat, 1 if cat)
caches – list of caches containing: every cache of linear_activation_forward() with "relu" (it's caches[l], for l in range(L-1) i.e l = 0...L-2) the cache of linear_activation_forward() with "sigmoid" (it's caches[$L-1$])

Returns:
grads – A dictionary with the gradients
grads["dA" + str(l)] = ...
grads["dW" + str(l)] = ...
grads["db" + str(l)] = ...

```python
def L_model_backward(AL, Y, caches):

    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    #Initializing the backpropagation
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    # Lth layer (SIGMOID -> LINEAR) gradients

    current_cache = caches[-1]
    dA_prev_temp, dW_temp, db_temp = linear_activation_backward(dAL,
    ↪   current_cache, "sigmoid")
    grads["dA" + str(L-1)] = dA_prev_temp
    grads["dW" + str(L)] = dW_temp
    grads["db" + str(L)] = db_temp

    # L-1 layer to 1 (RELU -> LINEAR) gradients
    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA" +
        ↪   str(l + 1)], current_cache, "relu")
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l+1)] = dW_temp
        grads["db" + str(l+1)] = db_temp

    return grads
```

Finally, we reach the learning step. We upgrade our new $W^{[l]}$ and $b^{[l]}$ on each layer using the gradient descent. In other words, for every iteration, we get closer to the minima. The definition is given at the beginning of the subsection, and the code is given below:

Algorithm 1.5.9: Update Parameters

Updates parameters using gradient descent.

Arguments:
params – python dictionary containing your parameters
grads – python dictionary containing your gradients, output of L_model_backward

Returns:
parameters – python dictionary containing your updated parameters
parameters["W" + str(l)] = ...
parameters["b" + str(l)] = ...

```python
def update_parameters(params, grads, learning_rate):

    parameters = copy.deepcopy(params)
    L = len(parameters) // 2 # number of layers in the neural network

    for l in range(L):
        parameters["W" + str(l+1)] = params["W" +
         ↪   str(l+1)]-learning_rate*grads["dW" + str(l+1)]
        parameters["b" + str(l+1)] = params["b" +
         ↪   str(l+1)]-learning_rate*grads["db" + str(l+1)]

    return parameters
```

Learning rate determines how big the step is per iteration as it goes down to the minima. It is a hyperparameter that may lead to divergence if the rate is too high, or taking long computational time if the rate is too low.

2 Improving the Adaptability of the Prediction

Applied machine learning is highly iterative. This is to find out the best set of hyperparameters to train the parameters. In particular, it is important to find out

- Number of layers
- Number of hidden units per layer
- Learning rates
- Activation functions

through the use of the idea→code→experiment cycle. Moreover, intuition from one application area usually does not transfer to another. In this section, we try to control the data set as well as fine tuning the change in $W^{[l]}$ and $b^{[l]}$ to improve the prediction output of the learning process.

2.1 Data Sets

Suppose we have n training data available. Traditionally, we split the training data into the training set, the dev set and the test set using either the 60%/20%/20% split or the 70%/30% split ignoring the test set. However, as we gather more and more data, it is useful to put in more data into training since we will still have a large amount of data for testing.

While previously data sets maybe of size nearing $10,000$, we now have data sets of size $1,000,000$. Modern deep learning appliers take the 98%/1%/1% split.

It is important to make sure the distribution of the train and test set to be similar. For example, training the cat picture identifier only using pet owner photos, while testing the program using security camera cat photos may result in the program being highly inaccurate. It is therefore important both the training data and test data come from the same distribution.

2.2 Bias and Variance

High bias is when the program unable to give correct predictions for most of the data set. It is also called underfitting. High variance is when the program is trained too specifically for the training data so that when given other data sets, the program becomes inaccurate. It is also called overfitting.

For example, suppose we are training a program to identify photos between cats and dogs. If the train set error is at 1% while the dev set error is at 10%. This indicates that the program is too specific to the training set, hence the high dev set error. In this case the prediction has high variance. Now if both the training set error and the dev set error are both 15%, this means that the program in general is just not performing well. Hence this program is high bias.

An important thing to know is that programs can be simultaneously high bias and high variance. This is when the training set error is high all the while dev set error is even higher.

All this is assuming that human error is almost 0%. If the data with manually provide to train is wrong, it is hard for the program to get right. This is also called Bayes' error in some literature.

In general, there are different solutions to whether the prediction is high bias or high variance. In the case that the prediction is high bias, we can use one of the following methods:

- Increasing the size of the network
- Allow the program to train longer (try different hypervariables)
- Test different Optimization Algorithms
- Change the network structure

with increasing the size of the network being almost always efficient. On the other hand, high variance can be solved using the following methods

- Collecting more training data
- Regularization
- Change the network structure

with collecting more training data and regularization being most useful.

One thing to note is the Bias-variance trade off. Usually, different methods while useful to solving one problem, may increase magnitude of the problem. For example, methods that reduced bias may increase variance. In modern day, this problem is not so much apparent. In particular, increasing the network size helps decreasing bias, fixes variance given we perform regularization. Collecting more training data will solve the problem of high variance while fixing the bias.

2.3 Regularization

Recall that in general, the core of deep learning is to optimize the function

$$J(W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(A^{[L](i)}, Y^{(i)})$$

Regularization refers to the fact that we add a regularization term to the cost function.

Definition 2.3.1: The Regularization Term

Suppose that a neural network with 1 layer has n inputs. Regularization refers to the fact that we choose one of the following regularization terms and add it to the cost function:

- L_2-regularization:
  $$\frac{\lambda}{2m}\sum_{k=1}^{n} w_k^2 = \frac{\lambda}{2m}\|w\|_2^2$$
  (most common)
- L_1-regularization:
  $$\frac{\lambda}{m}\sum_{k=1}^{n} w_k = \frac{\lambda}{m}\|w\|_1$$

where λ is the regularization parameter. It is a hyperparameter.

In a neural network with L layers, L_2 regularization is given by

$$\frac{\lambda}{2m}\sum_{l=1}^{L}\|W^{[l]}\|_F^2 = \frac{\lambda}{2m}\sum_{l=1}^{L}\sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}}(w_{ij}^{[l]})^2$$

The back propagation, depending on the cost function, also changes by adding the term

$$\frac{\lambda}{m}W^{[l]}$$

for layer l which is just the derivative of the L_2-regularization term. The effect becomes apparent when we see that the gradient descent step becomes

$$W_{n+1}^{[l]} = W_n^{[l]} - \frac{\alpha\lambda}{m}W_n^{[l]} - \alpha \cdot (\text{normal back prop term})$$

Notice here that W_n decreases by a small amount due to the introduction of the regularization term. Moreover, the larger the hyperparameter β, the larger the decrease in W_n becomes. In the gradient descent step, the importance of the weight decreases, and decays over time. This is why regularization is also sometimes called weight decay.

Since $W^{[l]}$ becomes smaller, this means that some of the nodes in layer l becomes less important. Their contribution becomes very small so that the the dependency on the some of the nodes becomes less. Recalling that high variance, means that the prediction fits the training data too well. By reducing node dependency, we can penalize nodes that the program become way too dependent on. Thus allowing the prediction to fit less of the training data.

Let us use tanh to illustrate. As λ increases, the gradient descent step changes W in to closer to 0. This means that Z follows W and decreases to close to 0. As seen in the graph of tanh, at near 0 the derivative is almost linear. This means that we are punishing too complicate fittings that are non-linear, which prevents overfitting.

There is also the notion of dropout regularization, in which each node is assigned a probability to be disabled, meaning that their contributions is 0. As such it is also a type of regularization, by decentralizing nodes that are overly dependent from the training. We demonstrate this using an implementation of inverted dropout.

We define a variable which is the probability for a node to be kept in the layer. We then standardize it by dividing the output $A^{[l]}$ with the probability so that the expected value remains the same. Otherwise it will be hard for the program to train the weights and bias. During making predictions at test time, we don't want any dropout since we don't want randomization in testing the data. Intuitively, nodes should not rely on any node in the previous layer. So by randomly disabling the nodes, we make sure that the weight are spread out and any strong dependency is penalized and so is lost.

Studies have shown that dropout has an extremely similar effect to that of L_2 regularization.

Furthermore, we can also vary the probability that keeps the node. In particular, for layers with less node, the probability of keeping the node should be higher so that these already small amount of nodes keep getting trained in a more constant manner. Conversely, layers with more nodes or layers in which overfitting may occur, the probability to keep nodes should be lower. We can also drop out inputs although it is an uncommon practise.

The down side of this is that J is now less well defined since it now randomly depends on different sets of nodes. It becomes hard to debug the cost function itself this way. Optimizing the cost function itself should be seen as a completely different task. (Orthogonalization)

Finally, there are also other regularization methods such as data augmentation, to flip, rotate crop and zoom in or out of images so that more data can be used to train it. Though this is not as good since the images are fundamentally the same, but it is a cheap way to obtain more data. Early stopping is a technique that as we are carrying out the test using the dev set, once we see that the cost function is not minimized or is not minimized the same way as the minimization done in the training set, we halt the process. Early stopping may not be a good way to regularize the program because it contradicts orthogonality. Orthogonality essentially means to do one task at a time / change one variable at a time. We will see more of this later. But early stopping is contrary to orthogonality since we are stopping the optimization of J as well as stopping the prevention of overfitting but halting early. This makes calculating a good λ for regularization harder.

3 Improving the Speed of the Algorithm

3.1 Normalization

Normalization is a process that allows training to be more efficient. This is because we standardize all our data into constant mean and variance each time W and b for each layer is renewed. We do it through the following:

Definition 3.1.1: Normalization

For a neural network at layer l, we normalize the inputs by the following equations:

$$A^{[l]}_{\text{norm}} = \frac{A^{[l]} - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

where μ is the mean of $A^{[l]}$ calculating over the training data and σ is the standard deviation. The ϵ term is extremely small to prevent σ from becoming 0 due to rounding.

For $A^{[l]} = \begin{pmatrix} | & & | \\ A^{[l](1)} & \cdots & A^{[l](m)} \\ | & & | \end{pmatrix}$, the mean is calculated by

$$\mu = \frac{1}{m}\sum_{k=1}^{m} A^{[l](k)}$$

and the variance is calculated by

$$\sigma^2 = \frac{1}{m}\sum_{k=1}^{m}(A^{[l](k)} * A^{[l](k)})$$

where $*$ is element wise multiplication. Finally division in $\frac{A^{[l]}-\mu}{\sigma}$ is given by column wise division of the matrix $A^{[l]} - \mu$ with the column vector σ, where the column divides the column element wise.

This results in the variance in different inputs to be the same, which is 1.

In the case that J is not normalized, the graph of J may be elongated or in weird shapes so that gradient descent becomes difficult. By normalizing J, gradient descent becomes easier, we can take larger steps for each step in the gradient descent. Indeed say x_1 is a value that ranges betwen 1 and 1000 and x_2 is a value that ranges between 0 and 1. Then W_1 and W_2 becomes very different.

3.2 Weight Initialization

Sometimes derivatives get huge or tiny. In the case that the activation function is linear, the weight matrix becomes diagonal so that inputs grow or shrink exponentially. Initializing the weights may solve this problem. Recall that $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$. If we want the size of $Z^{[l]}$ to be appropriate, for larger n we need smaller w_{ij} in general. So we initialize $W^{[l]}$ so that each element is between 0 and $\text{Var}(w_{ij}) = \sqrt{\frac{1}{n^{[l-1]}}}$. This allows the elements of the matrix to have a variance of $\frac{1}{n^{[l-1]}}$.

Other weight initialization equations include the Xavier Initialization:

$$\tanh\left(\sqrt{\frac{1}{n^{[l-1]}}}\right)$$

or

$$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$$

. This could also be a hypervariable.

3.3 Minibatch

One way to lower the computation time is to process the training data in a minibatch instead of one batch, as matrix multiplication becomes computationally costly for large matrices. We break the training data $A^{[0]}$ into

$$A^{\{1\}[0]}, \ldots, A^{\{m/s\}[0]}$$

where s is each size of the minibatch.

3.4 Optimization of the Gradient Descent

3.5 Learning Rates