

Course 1 -

Deep Learning & Neural Networks

WEEK 2

Binary Classification

Total number of training examples is denoted by m . The number of features for any input x is denoted by n_x . A particular training example is defined by $\{x^i, y^i\}$. All the training examples are combined into a single matrix X where x^i is a column of the matrix. Therefore X is a $n \times m$ matrix. All the training set labels y^i are also arranged into a single matrix Y where an output y^i is a single column of the matrix. Therefore Y will be a $1 \times m$ dimensional matrix.

For the hypothesis we use the sigmoid function. The output y is the probability that the true label is 1. So

$$y = \text{sigmoid}(z)$$

Where

$$z = w'x + b$$

w is the matrix of weights, which is an $n \times 1$ column vector (for the non-vectorized case), and b is a constant. So simplifying,

$$y = 1/(1 + e^{-(w'x + b)})$$

Our job is then to learn the matrix of weights w and the constant b such that y becomes a good measure of the probability that the true label y is 1.

Since we want to make it as close to the true label as possible, we calculate a loss/cost function and then try to minimize the cost function. We could use the squared error as our cost function, but it turns out that squared error can become non-convex and gradient descent can sometimes fail. So we define a different loss function. The loss function we use is

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

Where y is the true output label and \hat{y} is the output we get from our hypothesis. The cost function is the average of the loss function applied to all the training examples. The cost function is

$$J(w, b) = (1/m) * \sum_{i=1}^m L(y^{(i)}, \hat{y}^{(i)})$$

Or

$$J(w, b) = -(1/m) * \sum_{i=1}^m \{y^{(i)} \log(\hat{y}^{(i)}) + (1 - y^{(i)}) \log(1 - \hat{y}^{(i)})\}$$

Gradient Descent For Logistic Regression

The gradient descent update to minimize the cost function is

$$\begin{aligned} & \text{repeat till convergence} \{ \\ & \quad w_i := w_i - \alpha * (\partial/\partial w) J(w, b) \\ & \quad b := b - \alpha * (\partial/\partial b) J(w, b) \\ & \} \end{aligned}$$

The updates are always simultaneous updates.

Substituting the values of J and calculating the derivative, we can simplify the updates as

$$\begin{aligned} & \text{repeat till convergence} \{ \\ & \quad w_i = w_i - \alpha * x_i * (\hat{y}^{(k)} - y^{(k)}) \\ & \quad b = b - \alpha * (\hat{y}^{(k)} - y^{(k)}) \\ & \} \end{aligned}$$

Where w_i is the weight of the i th input feature x_i , and $x_i * (\hat{y}^{(k)} - y^{(k)})$ is $(\partial/\partial w)L(w, b)$, not $(\partial/\partial w)J(w, b)$. This is gradient descent for just 1 training example. Since $(\partial/\partial w)J(w, b)$ is just the average of $(\partial/\partial w)L(w, b)$ for all the m training examples, we take the average of $x_i * (\hat{y}^{(k)} - y^{(k)})$ for all k and for all input features x_i . So we implement this (in python) as,

```
J = 0, dw1 = 0, dw2 = 0, ... , db = 0
for i in range(m):
    #Do this for all the m training examples. The
    #superscript (i) means the ith example, which you
    #will pull from the training set.

    z(i) = w'X(i) + b
    a(i) = sigmoid(z(i))
    J += -(y(i)log(a(i)) + (1-y)log(1-a(i)))
    dz(i) = a(i) - y(i)

    for j in range(n):
        #There are n input features
        dwj += xj(i)*dz(i)

    db += dz(i)

J /= m
dw1 /= m
dw2 /= m
...
b /= m

#Now you perform the simultaneous updates
w1 := w1 - α*dw1
w2 := w2 - α*dw2
...
b := b - α*b
```

This was one step of gradient descent. Repeat this thousands of times to converge to the minima for J . We rarely use the for loops, however. We use vectorization to convert the calculations into matrix multiplications. Also always avoid explicit for loops whenever possible, use built in functions which do the same calculations wherever available. For performing element wise simple operations on arrays check if numpy has built in functions do it first.

Vectorizing Gradient Descent

To eliminate the inner for loop for calculating dw_j , we define dw as an $n \times 1$ column vector of all zeros. The inner for loop can then be replaced by

$$dw += X^{(i)} * dz$$

$X^{(i)}$ is the column vector which represents a single i th training example which has values from x_1 to x_n . Check the for loop again and verify that this does the same thing. Also the last operation for all elements of dw (dividing by m) can be done as

$$dw /= m$$

w is already an $n \times 1$ column vector containing the weights w_i . For the forward propagation step when we calculate the outputs for all of the training examples, we can replace the for loop iterating through all the examples by vectorizing as follows. Define w as the $n \times 1$ column vector containing the weights of the input features. Define Z as a $1 \times m$ row vector where z_i is $w^T X + b$ for the i th example. Since X is the matrix of all training examples stacked as columns, this is represented as

$$Z = [z^{(1)} \ z^{(2)} \ z^{(3)} \ \dots \ z^{(m)}] = w^T X + [b \ b \ b \ \dots \ b]$$

#Which translates to python as -

$$Z = \text{np.dot}(w.T, X) + b$$

Now we perform element wise exponentiation to calculate $\text{sigmoid}(Z)$ and store it as A . That gives us all of our outputs without using a single for loop. So, after this implementation of vectorization, we have the following variables defined.

```
#X is an nxm dimensional matrix containing all the training
#examples.
```

```

#Y is a 1xm dimensional row vector containing the true
labels
#for the corresponding training examples.
#Z = [z(1) z(2) z(3) ... z(m)] = wTX + [b b b ... b], an
intermediate
#A = [a(1) a(2) a(3) ... a(m)] = sigmoid(Z), the outputs of our
program,
#which we need to optimize.

```

Now to actually run gradient descent on these variables, we need vectors

```

#dz(i) = a(i) - y(i)
#dZ = [a(1) - y(1), a(2) - y(2), a(3) - y(3), ... , a(m) - y(m)]

#dZ = A - Y
#db = np.sum(dZ)/m
#dw = np.dot(X,dZ.T)

```

Therefore, the final implementation of 'nstep' steps of gradient descent in python is -

```

for i in range(nstep):
    Z = np.dot(w.T,X) + b
    A = sigmoid(Z)
    dZ = A - Y
    dw = np.dot(X,dZ.T)/m
    db = np.sum(dZ)/m

    #Simultaneous updates ----
    w := w - alpha*dw
    b := b - alpha*db

```

That's it.

WEEK 3

Shallow Neural Networks

Now that we have multiple layers of neurons, we introduce some new notation -

$X^{[i]}$ refers to the X matrix (matrix containing all the training examples) for a neuron in the i th layer. This $X^{[i]}$ matrix will consist of the activations of the neurons in the $i-1$ th layer.

$x^{(i)}$ refers to the i th training example in any X matrix.

$a_i^{[j]}$ refers to the i th neuron inside the j th layer.

Each layer has its own weight matrix $W^{[i]}$ associated with it. $W^{[i]}$ is constructed as follows - each neuron in the i th layer is activated by a Z calculated as $w.T * X + b$. The weights w for a single neuron in the i th layer is laid out as a row of the $W^{[i]}$ matrix. Therefore, $W^{[i]}$ will have as many rows as there are neurons in the i th layer, and each of those neurons will need a weight associated with all the inputs that the neuron gets. Each neuron will get as many inputs as there are neurons in the last $i-1$ th layer, since all the outputs of the previous layer are fed to the next layer. Therefore $W^{[i]}$ will have $n(i-1)$ columns. Therefore, $W^{[i]}$ will be an $[n(i) \times n(i-1)]$ matrix. This $W^{[i]}$ matrix can also be thought of as a vectorized implementation where each row is $w_j^{[i]}$ transpose (that's why it is $n(i) \times n(i-1)$ dimensional and not $n(i-1) \times n(i)$ dimensional).

Vectorizing SNNs

Each layer i has some $n^{[i]}$ nodes. Each of these nodes needs to perform 2 calculations - a $z_j^{[i]}$ calculation and $\text{sigmoid}(z_j^{[i]})$ to calculate its activation.

$$z_j^{[i]} = w_j^{[i].T} * x^{[i]} + b_j^{[i]}$$

#Therefore,

$$z_1^{[1]} = w_1^{[1].T} * x^{[1]} + b_1^{[1]}$$

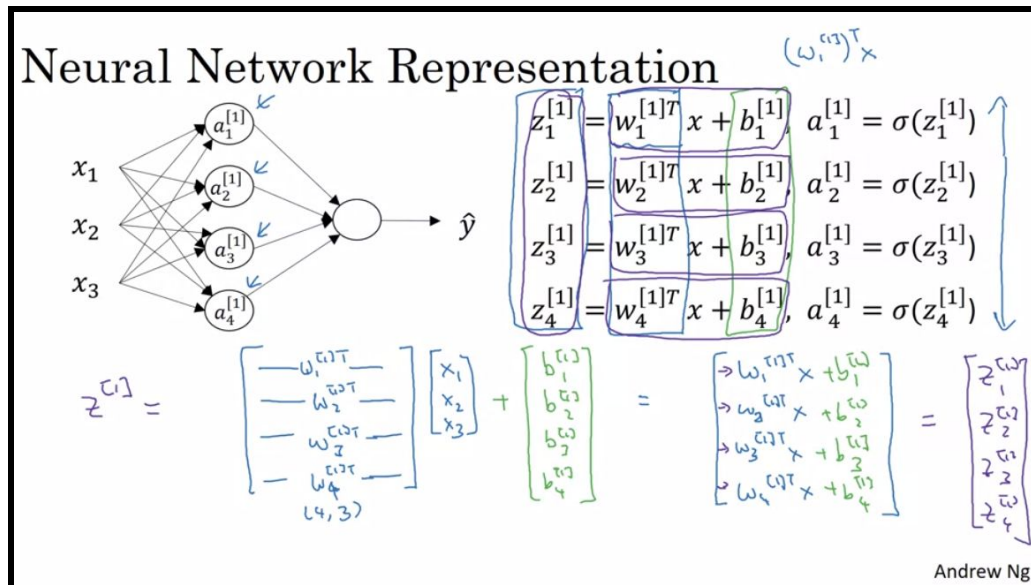
$$z_2^{[1]} = w_2^{[1].T} * x^{[1]} + b_2^{[1]}$$

...

$$z_n^{[1]} = w_n^{[1]T} * x^{[1]} + b_n^{[1]}$$

Instead of iterating through the n neurons with a for loop, we vectorize.

First, we stack all the weight matrices $w_j^{[i]T}$ into a single weight matrix $W^{[i]}$. We then make a matrix for input features $X^{[i]}$ by stacking the activations of the neurons in the previous layer into a column matrix. Then we make another column matrix for the offsets b. We can then calculate $Z^{[i]}$ by doing (see $z^{[i]}$)



Now we have vectorized the calculation of the activation of the neurons for a single training example. Now we must vectorize over all the training examples.

We make a very small change - instead of defining

$$z_j^{[i]} = w_j^{[i]T} * x^{[i]} + b_j^{[i]}$$

We do

$$Z_j^{[i]} = w_j^{[i]T} * X^{[i]} + b_j^{[i]}$$

the $x^{[i]}$ becomes $X^{[i]}$. Which means that instead of $z^{[i]}$ being an $n \times 1$ matrix, it becomes an $n \times m$ matrix, since $X^{[i]}$ is just all m training x stacked as columns (we end up with m columns). After applying the sigmoid/activation function element wise to $Z^{[i]}$, we get $A^{[i]}$. This $A^{[i]}$ is used as the input for the next layer.

Vectorizing across multiple examples

for $i = 1$ to m :

$$z^{[1]}(i) = W^{[1]}x^{(i)} + b^{[1]}$$

$$a^{[1]}(i) = \sigma(z^{[1]}(i))$$

$$z^{[2]}(i) = W^{[2]}a^{[1]}(i) + b^{[2]}$$

$$a^{[2]}(i) = \sigma(z^{[2]}(i))$$

$$X = \begin{bmatrix} x^{(1)} & x^{(2)} & \dots & x^{(m)} \\ \uparrow & & & \uparrow \\ & (n_x, m) & & \end{bmatrix}$$

$$\begin{aligned} z^{[1]} &= W^{[1]}X + b^{[1]} \\ \rightarrow A^{[1]} &= \sigma(z^{[1]}) \\ \rightarrow z^{[2]} &= W^{[2]}A^{[1]} + b^{[2]} \\ \rightarrow A^{[2]} &= \sigma(z^{[2]}) \end{aligned}$$

$$z^{[1]} = \begin{bmatrix} z^{[1]}(1) & z^{[1]}(2) & \dots & z^{[1]}(m) \\ 1 & 1 & & 1 \end{bmatrix}$$

$$A^{[1]} = \begin{bmatrix} a^{[1]}(1) & a^{[1]}(2) & \dots & a^{[1]}(m) \\ 1 & 1 & & 1 \end{bmatrix}$$

Andrew Ng

Done!

Activation Functions

If your activation function is linear, the entire neural network simplifies down to a single neuron, since the hidden neurons don't do any 'processing', they just pass on their z values to the next layer, which essentially is like applying a different set of weights and running a single neuron.

Gradient Descent for SNNs

Let's start by introducing the parameters we are going to apply gradient descent on. For a shallow neural network with just one hidden layer and one neuron in the output layer, we have the following parameters we have to optimize -

$W^{[1]}$ is the weight matrix for the first layer. It is $(n^{[1]}, n^{[0]})$ dimensional (since it is formed by stacking w^T as rows, see above).

$b^{[1]}$ is a column vector of the constant terms.

$W^{[2]}$ is the weight matrix for the second layer.

$b^{[2]}$ is a column vector of the constant terms.

We also have the cost function or the loss function.

The steps involved in a single step of gradient descent are -

1. Compute the prediction \hat{y} .
2. Calculate the derivatives $(\partial/\partial w)J(w, b)$ for all w and all b . w and b will be matrices not real numbers.
3. Update your parameters according to the simultaneous update equation.

The derivatives will be -

$$\begin{aligned}dZ^{[2]} &= A^{[2]} - Y \\dW^{[2]} &= (1/m) * dZ^{[2]} A^{[1]T} \\db^{[2]} &= (1/m) * np.sum(dZ^{[2]}, axis=1, keepdims=True) \\dZ^{[1]} &= W^{[2]T} dZ^{[2]} * g^{[1]'}(Z^{[1]}) \\dW^{[1]} &= (1/m) * dZ^{[1]} X^T \\db^{[1]} &= (1/m) * np.sum(dZ^{[1]}, axis=1, keepdims=True)\end{aligned}$$

Random Initialization

Logistic regression works if you initialize your weights to 0, but a neural network with even one hidden layer won't work if you set all your parameters to be 0. It doesn't work because since you initialize all the weights to be the same number, all of the neurons in that layer compute the exact same value. Hence, even after gradient descent is run on them, their weights end up being the same. Therefore, all the neurons in that layer function effectively as a single neuron. Therefore we use random initialization.

WEEK 4

Deep Neural Networks

New notation we use is

We use L to denote the number of layers in the network.

We use $n^{[L]}$ to denote the number of neurons in layer L .

Forward propagation for a deep neural network is represented by these equations

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

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

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$A^{[2]} = g^{[2]}(Z^{[2]})$$

...

...

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

$$A^{[L]} = g^{[L]}(Z^{[L]})$$

Forward and Backward Propagation

For the forward propagation step, we input $A^{[L-1]}$ and the formulae output $A^{[L]}$. They also output $Z^{[L]}$, $W^{[L]}$, and $b^{[L]}$ as cache to use during backward propagation.

For the backward propagation step, we input $dA^{[L]}$ and the formulae output $dA^{[L-1]}$, $dW^{[L]}$, and $db^{[L]}$.

Parameters and Hyperparameters

Parameters are values that the neural net needs to learn by itself - W and b .

Hyperparameters are the extra values we decide and that we give it that it needs to run - the learning rate, the number of iterations of gradient descent, the number of hidden layers, the number of hidden units in each layer, and the choice of activation function. We call these hyperparameters since these influence the actual parameters that the neural net is responsible for calculating.

Course 2

Improving Neural Networks, Hyperparameter Tuning, Regularization, Optimization

WEEK 1

When building a network, we take our data and split it into 3 parts - the training set, the development set, and the test set. We could be just splitting them into 2 sets - training and test, but that would give an inaccurate accuracy value for the final model you select.

We make multiple architectures for our neural net and train them all on the training set. We then test them on the dev set to see how they perform. After doing this for many models, we choose the best one and run it on the test set. The accuracy we get on the test set is the final performance rating of the model.

The ratio in which you split your data into train/dev/test depends on the amount of data you have. If you have a relatively small data set, say, 1,000 to 10,000 examples, split your data as 60% train, 20% dev, 20% test. However, if you have something like a

1,000,000 examples, set aside a small number of examples that you feel will give you a good enough measure of how well your model is doing, around 10,000 or lesser, and use the rest of the data to train. The more training examples you have, the better your model will perform.

Also try to make sure that your test and dev sets come from the same distribution of data. For example, if you are building a cat classifier, make sure that your dev and test set have the same quality and resolution. Sometimes you can gather data for your training set from different sources so that you will have a bigger training set, and that's okay, but make sure that your dev and test sets come from the same distribution to at least get an accurate measure of your model's performance.

Also, it might be okay to not have a separate test set if you don't need an unbiased measure of your model's performance. If you are satisfied with the performance of your model on the dev set, you can take that as a measure of your model's accuracy. However, this measure won't be unbiased.

Bias and Variance

We use the term bias to describe how well the model does on the training set. We use the term variance to describe how well the model does on the test/dev set, as compared to the training set. If the model does poorly on the training set, we say that it has high bias. If the model does poorly on the test set as compared to the training set, we say it has high variance.

Bias and Variance

Cat classification



Train set error:	1%	15%	15%	0.5%
Dev set error:	11%	16%	30%	1%
	high variance	high bias	high bias & high variance	low bias & low variance
Human:	20%			

If the model does well on the training set but does poorly on the test set, the model is overfitting for the training set and we say that it has high variance.

If the model does bad on the training set but not much worse on the test set, it is because the model is underfitting to the training set, and we say that it has high bias.

If the model does bad on the training set and even worse on the test set, we say that it has high variance and high bias, which is the worst of both worlds. This can happen if your model under fits the data in some region and over fits the data in some other region.

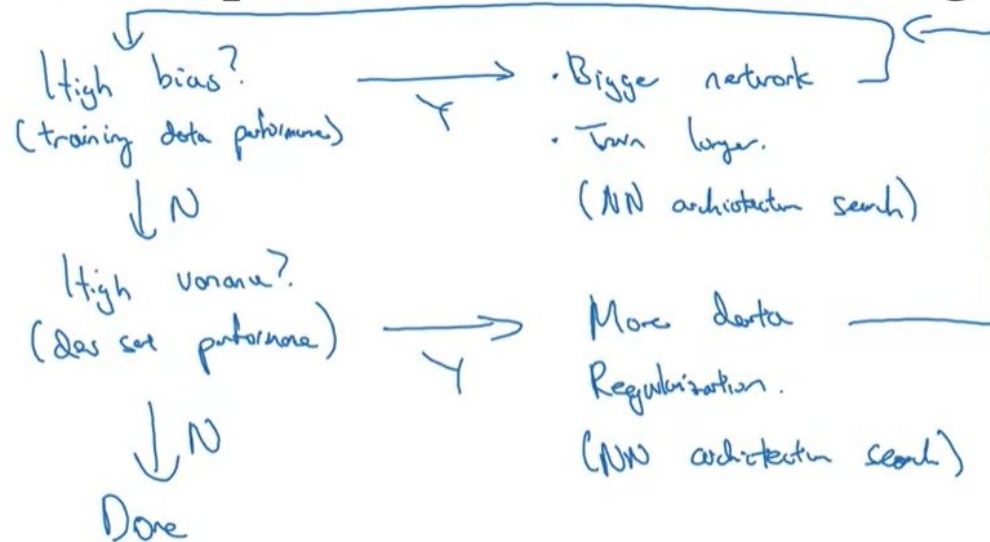
If the model does well on the training set as well as the test set, we say that it has low variance and low bias, and we have a good model.

How 'good' a model is doing is judged on how good it is possible to do on the particular problem (as compared to humans, or as compared to pre existing models).

If you have high bias, try a bigger network, or running more iterations of gradient descent, or search for an architecture that will fit your problem better.

If you fix your high bias problem but then have high variance, try getting more training examples, or regularizing your data, or search for an architecture that will fit your problem better.

Basic recipe for machine learning



Regularization

If you think your model is overfitting your data, you should first try regularization.

To regularize your parameter W for logistic regression, add this small term to the final cost function

$$J(w, b) = (1/m) * \sum_{i=1}^m L(y^{(i)}, \hat{y}^{(i)}) + (\lambda/2m) * \sum_{j=1}^{n_x} w_j^2$$

The last term is the regularization term called the L_2 regularization. It can also be written as $w^T w$ as a vectorized implementation. λ is called the regularization parameter and is a hyperparameter you will have to tune.

For a neural network, since $W^{[l]}$ is a matrix and not a vector, we use the Frobenius norm instead of the L_2 norm (which is almost the same thing).

$$J(w, b) = (1/m) * \sum_{i=1}^m L(y^{(i)}, \hat{y}^{(i)}) + (\lambda/2m) * \sum_{j=1}^L \|W^{[j]}\|_F^2$$

Where $\|W^{[L]}\|_F^2$ is the sum of squares of all the elements in the matrix.

$$\|W^{[L]}\|_F^2 = \sum_{i=1}^{n_L} \sum_{j=1}^{n_{L-1}} W_{ij}^2$$

Since we added this new term, we need to change our definition of dW . $dW^{[L]}$ now becomes

$$dW^{[L]} = (1/m) * dZ^{[L]} A^{[L-1]T} + (\lambda/m) * W^{[L]}$$

Sometimes L_2 regularization is also called weight decay.

There is another type of regularization called dropout regularization in which you randomly choose some neurons to be deactivated while training a single example. You then choose different neurons to be deactivated for a different example. Look up the lecture on dropout regularization if you want to implement it.

Normalization

If X is the input matrix (n_x, m), normalize it by performing the following operation

$$X = (X - \mu) / \sigma^2$$

Where μ is a ($n_x, 1$) dimensional column vector containing the mean of each feature x_i over all m training examples. σ^2 is the variance of the input matrix.

$$\mu = (1/m) * \sum_{i=1}^m X^{(i)}$$

$$\sigma^2 = (1/m) * \sum_{i=1}^m X^{(i)} ** 2 \quad (\text{element wise square})$$

Also remember to store the values of μ and σ^2 and use the same values to normalize the test or dev set even if they won't be the real μ and σ^2 values for these matrices. This is because you want to use the same transformation on both the training and test

set, and since this transformation is just to make computation easier and reduce the time taken to train, you must use the same transformation since the model has learned parameters specific to this transformation.

You can always use normalization, but it works best when your input features have widely different scales.

Exploding/Vanishing Gradients

If you have a very deep network, depending on the value of your weight matrices, the values of the activations of each layer can increase or decrease exponentially. This makes their gradients also increase or decrease exponentially. This makes gradient descent converge very slowly or diverge really fast.

This problem only has a partial solution, which involves biasing your weights when you randomly initialize them.

If you're using a ReLU activation function, initialize your weights as

$$W^{[1]} = \text{np.random.randn}(\text{\#shape}) * \text{np.sqrt}(2/n^{[1-1]})$$

If you're using a tanh activation function, initialize your weights as

$$W^{[1]} = \text{np.random.randn}(\text{\#shape}) * \text{np.sqrt}(1/n^{[1-1]})$$

or

$$W^{[1]} = \text{np.random.randn}(\text{\#shape}) * \text{np.sqrt}(2/(n^{[1-1]}+n^{[1]}))$$

This mostly helps when your network is very deep, maybe 10s of layers deep. Remember that different initializations lead to different results. Sometimes it results in a huge difference in accuracy. Use the above initializations even if your network isn't that deep.

Gradient Checking

Gradient checking helps you to check if your implementation of back propagation is correct. This is only a helpful debugging tool.

Take all of your parameter vectors $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, W^{[3]}, b^{[3]}, \dots, W^{[L]}, b^{[L]}$ and concatenate them into one single large vector called θ . Then your cost function $J(w, b)$ becomes $J(\theta)$.

Gradient check for a neural network

Take $W^{[1]}, b^{[1]}, \dots, W^{[L]}, b^{[L]}$ and reshape into a big vector θ .

concatenate

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = J(\theta)$$

Take $dW^{[1]}, db^{[1]}, \dots, dW^{[L]}, db^{[L]}$ and reshape into a big vector $d\theta$.

concatenate

Is $d\theta$ the gradient of $J(\theta)$?

Andrew Ng

Then define a function `grad_check` which calculates the following and compares it with $d\theta$.

Gradient checking (Grad check)

$$J(\theta) = J(\theta_1, \theta_2, \dots)$$

for each i :

$$\rightarrow \underline{d\theta_{\text{approx}}[i]} = \frac{J(\theta_1, \theta_2, \dots, \theta_i + \epsilon, \dots) - J(\theta_1, \theta_2, \dots, \theta_i - \epsilon, \dots)}{2\epsilon}$$

$$\approx \underline{d\theta[i]} = \frac{\partial J}{\partial \theta_i} \quad \Bigg| \quad d\theta_{\text{approx}} \approx d\theta$$

Check

$$\frac{\|d\theta_{\text{approx}} - d\theta\|_2}{\|d\theta_{\text{approx}}\|_2 + \|d\theta\|_2}$$

$$\epsilon = 10^{-7}$$

$$\approx \frac{10^{-7}}{10^{-5}} - \text{great!}$$

$$\rightarrow 10^{-3} - \text{worry.}$$

Andrew Ng

Where the epsilon is very very small.

Don't use gradient checking during training, only use it once to check if your implementation of back propagation is correct.

If grad_check fails, check the individual values where the difference between the true and calculated values is large. This won't always tell you where the problem is but sometimes might.

Remember to include regularization in both your backpropagation and gradient checking algorithm if you decide to use regularization.

Grad_check doesn't work if you're using dropout regularization since dropout randomly eliminates some neurons. If you are using dropout, run grad_check before you turn dropout on while training.

Example Neural Network Framework

This is an example. The supporting functions are not included, which are most of the work. This is the skeleton of the model which puts it all together.

```
In [ ]: def model(X, Y, learning_rate = 0.01, num_iterations = 15000, print_cost = True, initialization = "he"):
        """
        Implements a three-layer neural network: LINEAR->RELU->LINEAR->RELU->LINEAR->SIGMOID.

        Arguments:
        X -- input data, of shape (2, number of examples)
        Y -- true "label" vector (containing 0 for red dots; 1 for blue dots), of shape (1, number of examples)
        learning_rate -- learning rate for gradient descent
        num_iterations -- number of iterations to run gradient descent
        print_cost -- if True, print the cost every 1000 iterations
        initialization -- flag to choose which initialization to use ("zeros", "random" or "he")

        Returns:
        parameters -- parameters learnt by the model
        """

        grads = {}
        costs = [] # to keep track of the loss
        m = X.shape[1] # number of examples
        layers_dims = [X.shape[0], 10, 5, 1]

        # Initialize parameters dictionary.
        if initialization == "zeros":
            parameters = initialize_parameters_zeros(layers_dims)
        elif initialization == "random":
            parameters = initialize_parameters_random(layers_dims)
        elif initialization == "he":
            parameters = initialize_parameters_he(layers_dims)

        # Loop (gradient descent)

        for i in range(0, num_iterations):

            # Forward propagation: LINEAR -> RELU -> LINEAR -> RELU -> LINEAR -> SIGMOID.
            a3, cache = forward_propagation(X, parameters)

            # Loss
            cost = compute_loss(a3, Y)

            # Backward propagation.
            grads = backward_propagation(X, Y, cache)

            # Update parameters.
            parameters = update_parameters(parameters, grads, learning_rate)

            # Print the Loss every 1000 iterations
            if print_cost and i % 1000 == 0:
                print("Cost after iteration {}: {}".format(i, cost))
                costs.append(cost)

            # plot the loss
            plt.plot(costs)
            plt.ylabel('cost')
            plt.xlabel('iterations (per hundreds)')
            plt.title("Learning rate =" + str(learning_rate))
            plt.show()

        return parameters
```

WEEK 2

Optimization Algorithms

Mini-Batch Gradient Descent

If you have a huge training set, it can take a long time to train your network. Split the training set into lots of smaller batches, and run forward and backward propagation on them simultaneously. For example, if $m = 5,000,000$, running forward propagation on the 5M examples, then computing the cost, and then running backward propagation on the 5M examples is very slow. Instead, you split the 5M training examples into 5,000 batches of 1,000 examples. Then you run forward propagation on the first minibatch, compute the cost, run backward propagation on the first minibatch, and update the weights. Then move to the second minibatch.

This basically treats your training set as if you have a 1,000 examples and 5,000 training sets. The new notation we use is

$X^{(t)}$ = The input examples for the t^{th} batch.
 $Y^{(t)}$ = The output examples for the t^{th} batch.

If the mini batch size is m , mini batch gradient descent becomes the same as gradient descent. If the mini batch size is 1, it is called stochastic gradient descent, but stochastic gradient descent doesn't ever converge to the global minimum. Use a mini batch size which will give you a reasonable vectorization advantage but will also speed things up by taking small enough chunks of the training set at a time.

If your training set is small (<2,000 or 3,000 examples), just use batch gradient descent.

If your training set is larger than that, use mini batch sizes of powers of 2, starting from 64. Also make sure that your mini batches fit in your CPU/GPU's memory.

Exponentially Weighted Averages

If you have a discrete distribution of values $\theta(t)$ for all values of t , you can define the exponentially weighted average of all the values of $\theta(t)$ by defining another quantity

$$\begin{aligned}v(0) &= 0 \\v(t) &= v(t-1) * \beta + (1 - \beta) * \theta(t)\end{aligned}$$

Where β is another parameter that you decide which ranges from 0 to 1. When you plot $v(t)$ vs t , you will get a curve which will be an exponentially weighted average of $\theta(t)$ over $1/(1 - \beta)$ values of t .

To implement/take the exponentially weighted average of any data, you need the data in your memory as well as one more variable for v . Then do this

```
theta = [...]  
v = 0  
for i in range(len(theta)):  
    v = beta*v + (1-beta)*theta[i]
```

However, this moving average is not accurate for initial values of i , since we just initialised v to 0, the first few values of v become very small compared to their corresponding values of θ . To correct this, we implement a bias. We change the update rule for v as

```
theta = [...]  
v = 0  
for i in range(len(theta)):  
    v = (beta*v + (1-beta)*theta[i]) / (1-beta**t)
```

Grad Descent with Momentum

Gradient descent with momentum only differs from gradient descent in the update rule for the parameters. Here is the implementation

```

vdW[1] = np.zeros((dW[1].shape))
vdb[1] = np.zeros((db[1].shape))
for i in range(number_iterations):
    calculate dW[1], dB[1] for the current mini batch or batch
    vdW[1] = beta*vdW[1] + (1-beta)*dW[1]
    vdb[1] = beta*vdb[1] + (1-beta)*db[1]

    W[1] = W[1] - alpha*vdW[1]
    b[1] = b[1] - alpha*vdb[1]

```

Now alpha and beta are both hyperparameters.

You can scale it up by (1-beta) as above if you want or not, since the bias disappears in a few iterations anyway.

Gradient descent with momentum will almost always work faster than normal gradient descent.

RMS Prop

RMS prop does the same thing as gradient descent with momentum, only the updates are a little different.

```

SdW[1] = np.zeros((dW[1].shape))
Sdb[1] = np.zeros((db[1].shape))
for i in range(number_iterations):
    calculate dW[1], dB[1] for the current mini batch or batch
    SdW[1] = beta*SdW[1] + (1-beta)*(dW[1])**2 #elementwise sq
    Sdb[1] = beta*Sdb[1] + (1-beta)*(db[1])**2 #elementwise sq

    W[1] = W[1] - alpha*dW[1]/np.sqrt(SdW[1] + epsilon)
    b[1] = b[1] - alpha*db[1]/np.sqrt(Sdb[1] + epsilon)

```

The epsilon is there to make sure we don't divide by zero. It can be a very small value (around 1e-8).

Adam Optimization

Adam optimization combines both RMS prop and momentum. The implementation is as follows

```
vdW[1] = np.zeros((dW[1].shape))
vdb[1] = np.zeros((db[1].shape))
SdW[1] = np.zeros((dW[1].shape))
Sdb[1] = np.zeros((db[1].shape))

for i in range(number_iterations):
    vdW[1] = beta1*vdW[1] - (1-beta1)*dW[1]
    vdb[1] = beta1*vdb[1] - (1-beta1)*db[1]
    SdW[1] = beta2*SdW[1] - (1-beta2)*dW[1]**2
    Sdb[1] = beta2*Sdb[1] - (1-beta2)*db[1]**2

    vdW_corrected[1] = vdW[1]/(1-beta1**i)
    vdb_corrected[1] = vdb[1]/(1-beta1**i)
    SdW_corrected[1] = SdW[1]/(1-beta2**i)
    Sdb_corrected[1] = Sdb[1]/(1-beta2**i)

    W[1] = W[1] -
alpha*vdW_corrected[1]/np.sqrt(SdW_corrected[1] + epsilon)
    b[1] = b[1] -
alpha*vdb_corrected[1]/np.sqrt(Sdb_corrected[1] + epsilon)
```

The hyperparameters you have now are

- alpha - tuneable, try a range of values and see which one is best.
- beta1 - recommended value is 0.9
- beta2 - recommended value is 0.999
- epsilon - recommended value is 1e-8

Learning Rate Decay

This is pretty self-explanatory. You slowly decrease the value of alpha as the number of iterations or the mini batch number increases. Implement it using

```
alpha = alpha/(1+decay_rate*batch_number)
or
alpha = (decay_rate2)**batch_number*alpha
or
#any definition in which alpha decreases over time. Can be
a discrete decrease too.
```

WEEK 3

Hyperparameter Tuning

There are a lot of hyperparameter choices for you to make in a large neural network, especially if you're using ADAM Optimization, etc. The order of importance of the hyperparameters, and thus, the order in which you should tune them is

1. Learning rate α
2. Momentum term β , the number of hidden units in each layer, mini batch size.
3. Number of hidden layers/Architecture, learning rate decay.

To tune a set of hyperparameters, set a range for each parameter, decide on the number of different values of all the hyperparameters you want to try (n), and then choose n different points from a random sampling. If you want to tune p different parameters, think of it as choosing n points from a p dimensional space, where each axis has its own range.

Once you choose and try out n different points in the range you selected, you will most likely get a smaller subset or range in which your model gives you the results you want. Then reduce your range to that smaller subset and sample n more points and train again to get the best results.

When the range over which you want to choose values for a hyperparameter is large, for example from 0.0001 to 1 for the learning rate, don't choose random values directly

from 0.0001 to 1, instead choose values from -4 to 0 and take 10^{ans} as the value for the learning rate. In other words, use a logarithmic scale instead of a linear scale.

Do the same when you're sampling for the momentum term beta.

Batch Normalization

When we normalized the input matrix X , if you have a deep neural network, the effect of that normalization can diminish throughout the layers. To fix that, we use batch normalization.

To implement batch norm,

```
#Do this for all training examples
 $\mu = (1/m) * \sum_{i=1}^m Z^{[1]}(i)$ 
 $\sigma^2 = (1/m) * \sum_{i=1}^m (Z^{[1]}(i) - \mu)^2$  #elementwise
 $Z_{\text{normalized}}^{(i)[1]} = (Z^{[1]}(i) - \mu) / \text{np.sqrt}(\sigma^2 + \epsilon)$ 
 $Z_{\text{tilde}}^{[1]}(i) = \gamma * Z_{\text{normalized}}^{[1]}(i) + \beta$ 

#We then use  $Z_{\text{tilde}}^{[1]}(i)$  everywhere instead of  $Z^{[1]}(i)$ 
```

Now β and γ also become learnable parameters for your network. There are different betas and gammas for each layer.

If you apply batch norm, it effectively cancels out the b parameter. So now your parameter set is W , beta, and gamma.

$\beta^{[l]}$ and $\gamma^{[l]}$ will have dimensions the same as $Z^{[l]}$. $Z^{[l]}$ has dimensions $(n^{[l]}, 1)$ or $(n^{[l]}, m)$, depending on whether you're vectorizing over all training examples or not. Beta and gamma will therefore have the same dimensions.

To then apply the network on a test example, you will need a value for μ and σ^2 for your test example. But since the test example won't be a matrix, rather a vector, the mean and variance across different rows doesn't make sense. So you just take an exponentially weighted average of the mean and variance of the Z value for that layer

for all the mini batches and use that as the μ for your test example, and you take the same exponentially weighted average of the σ^2 of the Z values for that layer for all the mini batches and use that as the σ^2 for your test example.

Multi Class Classification & Softmax Regression

When you have c classes to classify into instead of just 2, the output layer L has c units instead of 1. The only change that we make then is to the activation function of the output layer. We use an activation function $g^{[L]}(Z^{[L]})$ where $g^{[L]}$ is

$$g^{[L]}(Z^{[L]}) = e^{**}(Z^{[L]}) / \sum_{i=1}^c e^{**}(Z_i^{[L]})$$

Where $Z_i^{[L]}$ is the i th class or the i th output unit of the output layer. This is effectively like raising e to the power of each element in $Z^{[L]}$ and then dividing each element in Z by the sum of all elements in Z (which we do to make the sum of all elements of $Z = 1$).

The loss function we use for softmax regression is

$$L(y, \hat{y}) = \sum_{i=1}^c y_c * \log(\hat{y}_c)$$

Where both y and \hat{y} are $(c,1)$ dimensional row vectors.

Then the cost function becomes

$$J(W, b) = (1/m) * \sum_{j=1}^m L(y^{(j)}, \hat{y}^{(j)})$$

That is the average of the loss for all training examples.

For running gradient descent, the derivative of the last layer becomes

$$dZ^{[L]} = \hat{y} - y$$

Course 3

Structuring Machine Learning Projects

WEEK 1

Performance Evaluation Metrics

To measure how well our model is doing on a data set, we could use two numbers - precision and recall.

Precision is defined as the percentage of examples that were classified as 1 which were actually 1, and recall is defined as the percentage of examples which were 1 which were classified as 1.

But using two performance metrics is counterproductive when one model performs better at one metric and worse at another. So, we combine these two metrics by taking their harmonic mean and use that as the final metric. This metric is called the F_1 score.

You can also decide that one metric is particularly important to you (usually the classifying accuracy), and that the others don't really matter as long as they are above some threshold. You can then call the most important metric your optimizing metric and all the others as your satisficing metrics. The satisficing metrics are the metrics which just have to be above or below some level and then you wouldn't care how better they are. The optimizing metric is the metric which you want to have the best performance.

Train/Dev/Test Sets

Always choose your dev set and test sets from the same distribution of data - the data which you expect to see in the field and which is important to do well on. Choosing dev and test sets from different distributions is very counterproductive as you obviously won't get the performance you want on the test set if you work on fitting the dev set well and it contains data from a completely different distribution.

When you have a small total data set, say, less than 10,000 examples, use a 70/15/15 or 60/20/20 split for your train/dev/test set. Just remember to shuffle your data and randomly select examples to go into the different sets. If you have millions of examples, choose a certain amount of examples that would give you a satisfactory level of accuracy of the performance of your model for the dev/test set and put all the remaining examples in the training set.

WEEK 2

If you find that your (training) dataset itself has a few mislabelled data points, then you should go in and try to fix the dataset only if the errors are systematic. That is, if a certain type of data point has consistently been labelled incorrectly, then your network will learn to label that type of data point incorrectly too. In that case, go into your dataset and try to fix the mislabelled data points.

If the mislabelled data points are just random and have been mislabelled due to human error, then you can train your network on the slightly incorrectly labelled dataset anyway, as deep learning algorithms are quite robust to a few random errors.

If you also have mislabelled examples in your dev/test set, you should fix them only if the number of mislabelled examples makes it harder for you to get an accurate measure of your model's performance. Otherwise let it be.

Training/Dev/Test Set Mismatch

If your training set, dev set, and test set come from different distributions, you may get a sudden jump between your training set error and dev set error. To be sure if this jump

is because of the data distribution mismatch or because of a variance problem, take a small portion of your training set and call it the training-dev set. Don't train on this set, but use it to calculate the error. This error is your variance problem. The difference between training-dev error and dev error is the data distribution error.

Transfer Learning

Once you have trained a model for a particular task, you can then switch your goals and use the same network or a part of the same network to perform a different task. For example, once you have trained a network to recognize cats, you can take the same architecture, and take the same weights of most of the layers, and just randomly initialize the weights of the last few layers and retrain the network on a dogs dataset. This helps your network get trained much faster as it already recognizes the main parts of a face, and now only needs to learn the different features of a dog's face compared to a cat's face.

This technique is mainly used if you have a very small data set. Train your model on a very similar but large dataset and then change just the last few layers and retrain the changed layers with your small dataset.

You should use transfer learning from task A to task B if -

1. A and B have the same type/format of input.
2. You have much more data for A than for B.
3. The basic/overall features of the input for A are similar to the features for B.

End-to-End Learning

End to end deep learning is a process applied to huge neural nets which skips the intermediate steps required in conventional approaches to build an application and directly maps the input to the output.

Course 4

Convolutional Neural Networks

WEEK 1

Convolution Operation

The main problem with using regular deep neural networks for image processing or computer vision is that the number of input parameters grows very fast as the resolution of the input image grows. For a standard 1 megapixel image, we get 3 billion input parameters for the first hidden layer alone.

We instead use convolutional neural networks. Convolutional neural networks make use of the convolution operation. The convolution operation is implemented in python as `conv-forward`, in tensorflow as `tf.nn.conv2d`, and in keras as `conv2D`.

Convolving an image matrix with a 3x3 matrix of columns 1,0, and -1 performs vertical edge detection.

Convolving an image matrix with a 3x3 matrix of rows 1,0, and -1 performs horizontal edge detection.

A vertical edge is defined as a region with bright pixels on the left and dark pixels on the right. A horizontal edge is defined as a region with bright pixels on top and dark pixels at the bottom.

These 3x3 matrices are called filters or kernels, and their elements don't have to be 1, 0, and -1. They can be different combinations of numbers which allow you to detect different edges.

The point of convolutional neural networks is to treat the 9 numbers in the filters as parameters and use back propagation to learn them. The network then learns the best possible values to detect edges at different rotations.

However, there are a few problems with the convolution operation. Firstly, the image shrinks a little every time you apply the operation. If you apply an $f \times f$ filter on an $n \times n$ image, you get back an $(n-f+1, n-f+1)$ image. Moreover, the pixels at the very edges of the image are considered in the calculation much less than the pixels at the center of the image. This results in the edge detection being weak near the borders of the image. To solve both these problems, we apply a small padding to the image. If you're using a 3×3 filter, extend the image by 1 pixel on both sides. This new pixel is usually filled with a 0. More generally, you can apply a padding of p pixels to neutralize the shrinking effect of the $f \times f$ filter. This causes the output image to also be $n \times n$ and also eliminates the bias against corners problem. The output image now becomes $(n+2p-f+1, n+2p-f+1)$ dimensional.

There are two types of convolution operations - valid convolutions and same convolutions. Valid convolutions are convolutions which have no padding and shrink the image a little. Same convolutions are convolutions which have a padding such that the same dimensional image is returned.

You can choose to take a stride of more than one when you perform the convolution operation. If you take a stride of s , the dimensions of the output matrix will be

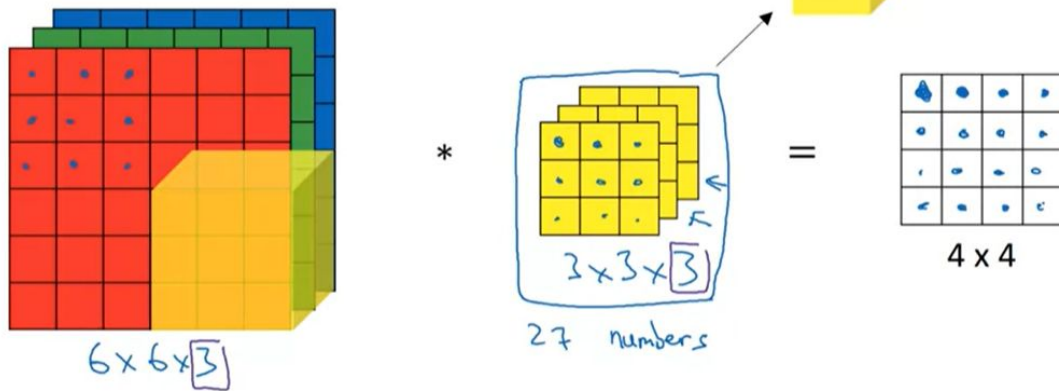
$$((n + 2p - f)/s + 1, (n + 2p - f)/s + 1)$$

If the fraction is not an integer, then we take its floor.

Convolution Over 3 Dimensions

This was the convolution operation on 2 dimensions. To apply it to colored images, which have dimensions (height, width, #channels), we use a filter or kernel of dimensions $(f, f, \text{\#channels})$. The operation still outputs a 2D matrix. The operation is similar to the 2D operation. You place the filter on the top left corner, multiply element wise the elements of the filter and the matrix, then shift the filter over by one stride length.

Convolutions on RGB image



If you want to use multiple filters on the image at the same time, you can convolve all the filters you want to apply to the image separately and then stack the outputs in layers. The output will then be of the dimensions

$$((n + 2p - f)/s + 1, (n + 2p - f)/s + 1, \#filters)$$

And this output will be the input for the next layer of the convolutional neural network.

To apply this concept to a neural network, we treat neurons as performing two operations - convolution and activation. Each neuron convolves the input matrix with a filter, adds a bias real number b to it, and then applies a non-linear activation function to it. This is one output. We then stack the outputs of all the neurons in layers and use that as input for the next layer.

New notation -

$f^{[1]}$ = dimensions of the filter/ filter size for layer 1

$p^{[1]}$ = padding size for layer 1

$s^{[1]}$ = stride for layer 1

$n_c^{[1]}$ = number of filters in layer 1

filter dimensions for layer 1 = $f^{[1]} \times f^{[1]} \times n_c^{[1-1]}$

activation of layer 1 = $n_h^{[1]} \times n_w^{[1]} \times n_c^{[1]}$

Activation (w/ vectorization) = $m \times n_h^{[1]} \times n_w^{[1]} \times n_c^{[1]}$

weights of layer $l = f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$
 bias for layer $l = n_c^{[l]}$ OR $(1, 1, 1, n_c^{[l]})$

input dimensions for layer $l = n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$
 layer l produces output of dimensions $= n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

dimensions of the last layer and current layer are related
 as -

$$n_{H/W}^{[l]} = \text{floor}((n_{H/W}^{[l-1]} + 2p^{[l]} - f^{[l]}) / s^{[l]} + 1)$$

Summary of notation

If layer l is a convolution layer:

$f^{[l]}$ = filter size

$p^{[l]}$ = padding

$s^{[l]}$ = stride

$n_c^{[l]}$ = number of filters

→ Each filter is: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]}$

Activations: $A^{[l]} \rightarrow n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

Weights: $f^{[l]} \times f^{[l]} \times n_c^{[l-1]} \times n_c^{[l]}$

bias: $n_c^{[l]} = (1, 1, 1, n_c^{[l]})$ ← #f: (f: filters in layer l).

Input: $n_H^{[l-1]} \times n_W^{[l-1]} \times n_c^{[l-1]}$

Output: $n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}$

$$n_{H/W}^{[l]} = \left\lfloor \frac{n_{H/W}^{[l-1]} + 2p^{[l]} - f^{[l]}}{s^{[l]}} + 1 \right\rfloor$$

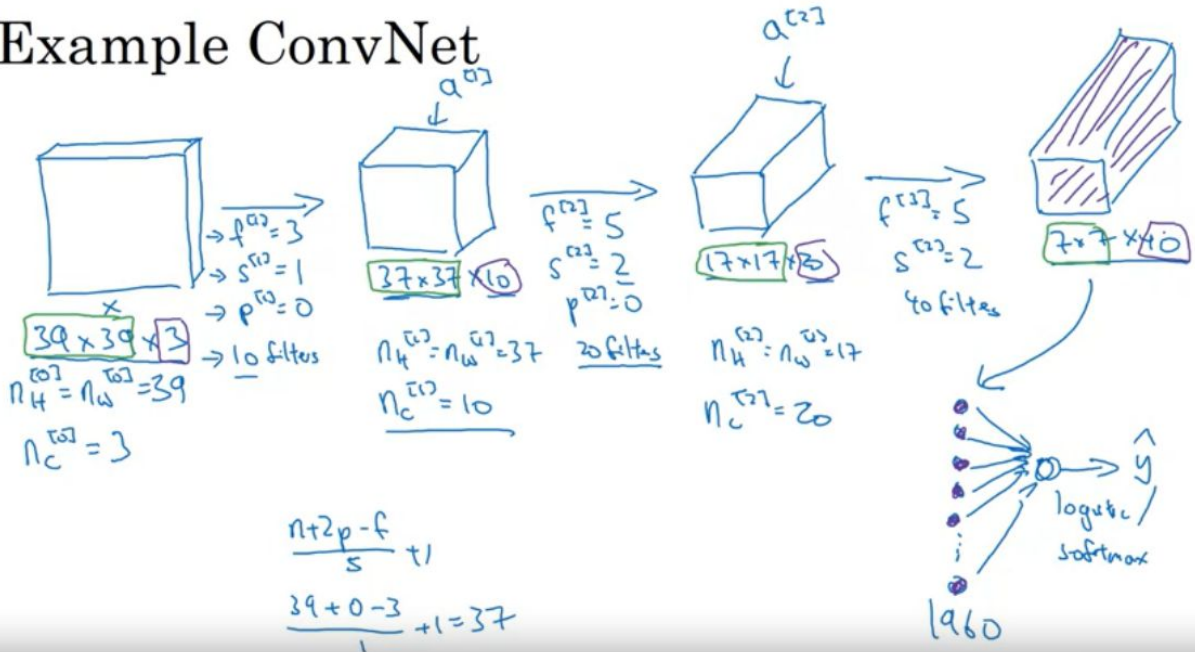
$$A^{[l]} \rightarrow m \times \underbrace{n_H^{[l]} \times n_W^{[l]} \times n_c^{[l]}}_{n_c^{[l]} \times n_H^{[l]} \times n_W^{[l]}}$$

Andrew Ng

Building ConvNets

We use convolutional layers in the network mainly to shrink the dimensions of the image down to a manageable amount, and then flatten the dimensions and feed it into a logistic regression or softmax regression layer to calculate the output.

Example ConvNet



Andrew Ng

Pooling Layers

Pooling layers are layers which help to directly reduce the dimensions of the previous layer. For example, if you want to turn a 4×4 image into a 2×2 image, use max pooling. Max pooling will divide the 4×4 area into a 2×2 area where a single cell in the output will have the maximum of 4 cells in the input image.

So in effect, max pooling is performed just like the convolution operation is performed, except the operation performed is the max operation, or the average operation, or some other simple operation.

The advantage of this type of pooling layer is that it doesn't need to learn any parameters, and that it immediately shrinks the dimensions of the image down.

Max pooling is used much more often than average pooling, and the common values of f and s are $(2,2)$ or $(3,2)$. Padding is generally not used.

WEEK 2

1x1 Convolutions

If you want to preserve the height and width of the previous layer's output and just want to reduce the number of filter layers, use 1x1 convolutions. A 1x1 convolution is effectively taking all the numbers in a particular slice of the input $[x, y, :]$ and adding them together with a weight. You can then keep the height and width dimensions of the previous layer and just change the depth of the output of this layer by changing the number of filters you use for this layer.

You should use 1x1 convolutions if you have too many multiplications to perform.

Residual Networks

Very deep neural networks are very susceptible to the problem of vanishing or exploding gradients. As the number of layers increases, the gradients become very very close to zero, and gradient descent becomes prohibitively slow.

To solve this problem, we use residual networks or ResNets. In a ResNet, the output $A^{[l]}$ is passed on to the layer $l+2$, and is added to its Z value. So the output of each neuron becomes

$$A^{[l]} = g^{[l]}(Z^{[l]} + A^{[l-2]})$$

WEEK 3

Object Detection

Before we do object detection, we have to do object localization. Object localization is when you not only identify the object in the image, but also draw a bounding box around it. To do this, we make the neural network output 4 more numbers - b_x, b_y, b_h, b_w . The first two, b_x and b_y , are the x and y locations of the center of the bounding box,

and their units are percentages of the total height and total width of the image. b_h and b_w are the width and height of the bounding box, and their units are also percentages of the total height and width of the image.

We now change the format of the output label Y a little. Y will now be a row vector where the first element is P_c , the probability that the image contains an object we are trying to detect. The next 4 elements will be b_x , b_y , b_h , and b_w . Then, if $P_c \geq 0.5$, we also output $C_1, C_2, C_3, \dots, C_n$, that is the probability of all the classes being in the image, like a standard softmax output.

If there is no object in the image, the first element is a 0 and all the other elements are NaNs.

We then calculate the loss using standard squared error. We define the loss as -

$$L(\hat{y}, y) = \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad \text{If } y_1 = 1, \text{ and}$$


$$L(\hat{y}, y) = (y_1 - \hat{y}_1)^2 \quad \text{If } y_1 = 0$$

Defining the target label y

{ 1 - pedestrian
 2 - car ←
 3 - motorcycle
 4 - background ←
 }

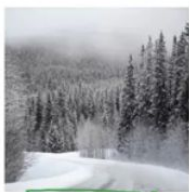
$L(\hat{y}, y) = \begin{cases} (\hat{y}_1 - y_1)^2 + (\hat{y}_2 - y_2)^2 + \dots + (\hat{y}_n - y_n)^2 & \text{if } y_1 = 1 \\ (\hat{y}_1 - y_1)^2 & \text{if } y_1 = 0 \end{cases}$

Need to output b_x, b_y, b_h, b_w , class label (1-4)

$x =$ 

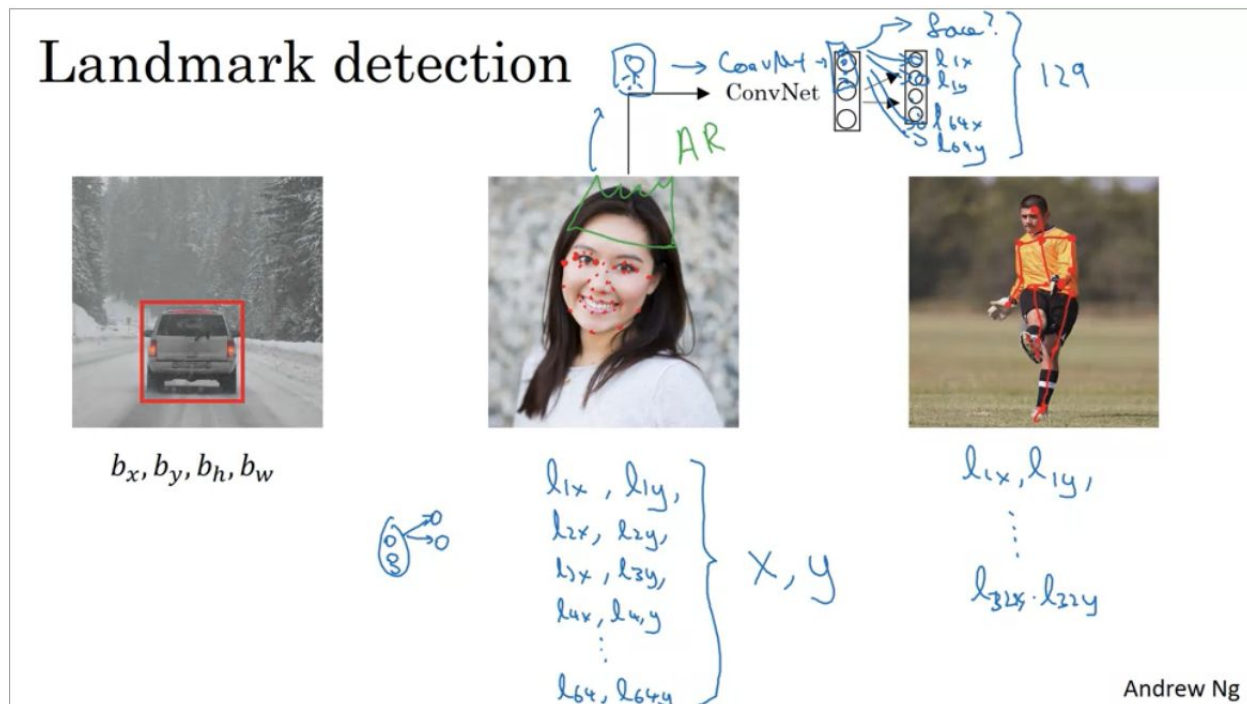
$y =$ $\begin{bmatrix} P_c \\ b_x \\ b_y \\ b_h \\ b_w \\ C_1 \\ C_2 \\ C_3 \end{bmatrix}$ is there any object?

$(x, y) = \begin{bmatrix} 1 \\ b_x \\ b_y \\ b_h \\ b_w \\ 0 \\ 1 \\ 0 \end{bmatrix}$

 $\begin{bmatrix} 0 \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \\ ? \end{bmatrix}$ P_c ← "don't care"

Andrew Ng

Instead of drawing a bounding box around the object, you can also have the network output some specific points on the image, known as landmarks. For example, you can label a considerable number of points on the edge of a person's mouth, and then train the network to identify those points on a face. You can then use those points to detect a smile, for example. This is what snapchat filters use to animate their special effects.



The first element of your output label will be a boolean corresponding to whether or not there is an object you care about in the image, then the next $2n$ elements will be the x and y coordinates of the n landmark points you select.

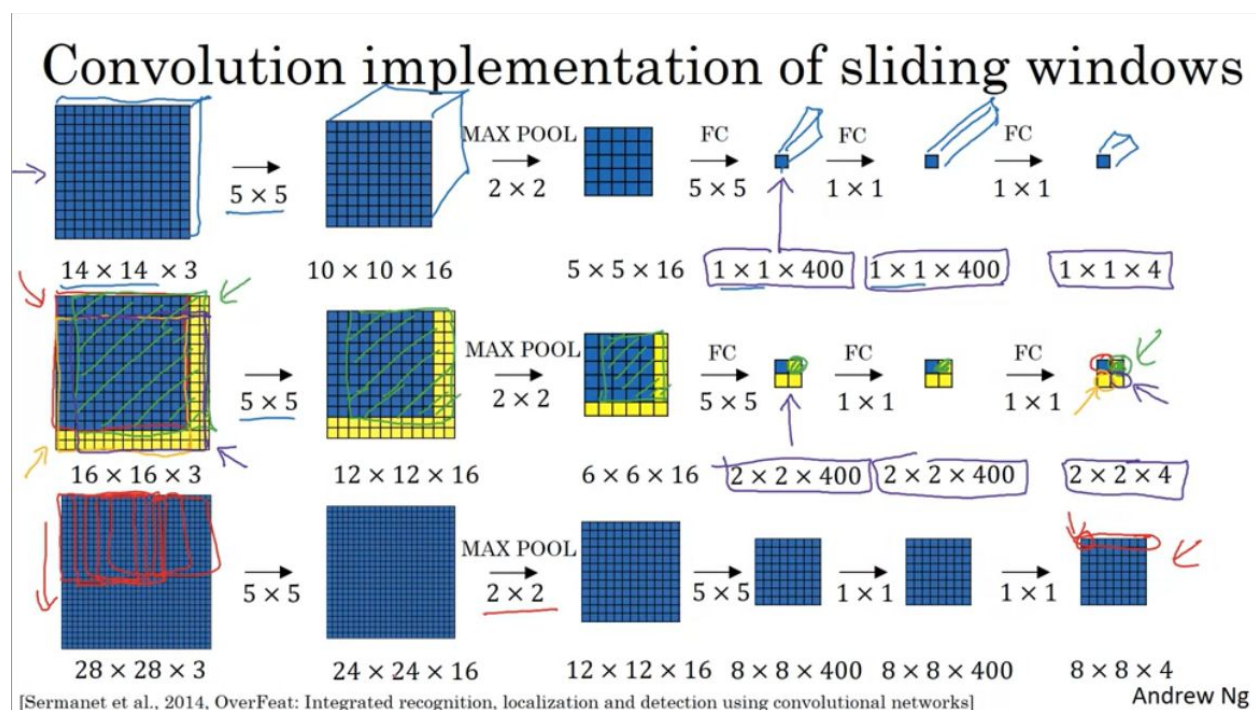
Sliding Windows Object Detection

Another method to detect the locations of objects in an image is to decide on a small window size, and then take small crops of that size all over the image and run a simple softmax classifier to detect if our object is present in the image. If it is, we draw a bounding box over that cropped region. We use many window sizes to detect objects close and far from us. But the problem with this brute force method is that it can be very computationally expensive, and may not work fast enough to be practical in a field such as self driving cars.

The solution for this is a convolutional implementation of sliding windows.

You can turn the last few fully connected layers into convolutional layers. If you have n units in the fully connected layer, use a filter of the same size as the input of the last layer. This will give a 1×1 output. Then use as many filters as there are units in the fully connected layer. This gives a $1 \times 1 \times n$ volume, which you can treat as the output of the fully connected layer.

Now that your fully connected layers are represented as convolutional layers, you can basically calculate all the outputs of all the window positions in a single forward propagation. For example, if your input image is $n \times n$, and you choose the input to your network to be $m \times m$, performing a single forward propagation using that network on the $n \times n$ image anyway will give you a $k \times k$ output instead of the 1×1 output that you would have got using an $m \times m$ image, but the $k \times k$ output corresponds to running a forward pass in all the positions of the $m \times m$ window on the $n \times n$ image.



However, this sliding windows implementation is still not completely accurate. It also can't put a rectangular bounding box around the object if it is not square.

YOLO Algorithm

To deal with these problems, we use the YOLO algorithm. The YOLO algorithm works as follows -

1. Split the image into an $n \times n$ grid.
2. For each cell of the image, make an output label y of the same format as defined on the object detection page.
3. The final output label Y will be formed by stacking together the individual y labels of all the cells to make an $n \times n \times (c+5)$ matrix, where c is the number of classes you are trying to detect.
4. If an object is split into multiple grid cells, mark the label y as positive only in the cell in which the midpoint of the object resides.
5. Your output label is now $n \times n \times (c+5)$. You can then take the input image and convolve or pool or do whatever you want and bring it down to dimensions $n \times n \times (c+5)$ instead of $(c+5) \times 1$.

Usually the grid size $n \times n$ is around 19×19 or 20×20 .

To check if the bounding box predicted by your model fits well with the true bounding box, we give it a number from 0 to 1. The number is calculated as the ratio between the area of intersection of the 2 boxes and the area of union of the two boxes. This is known as the intersection over union, or IoU.

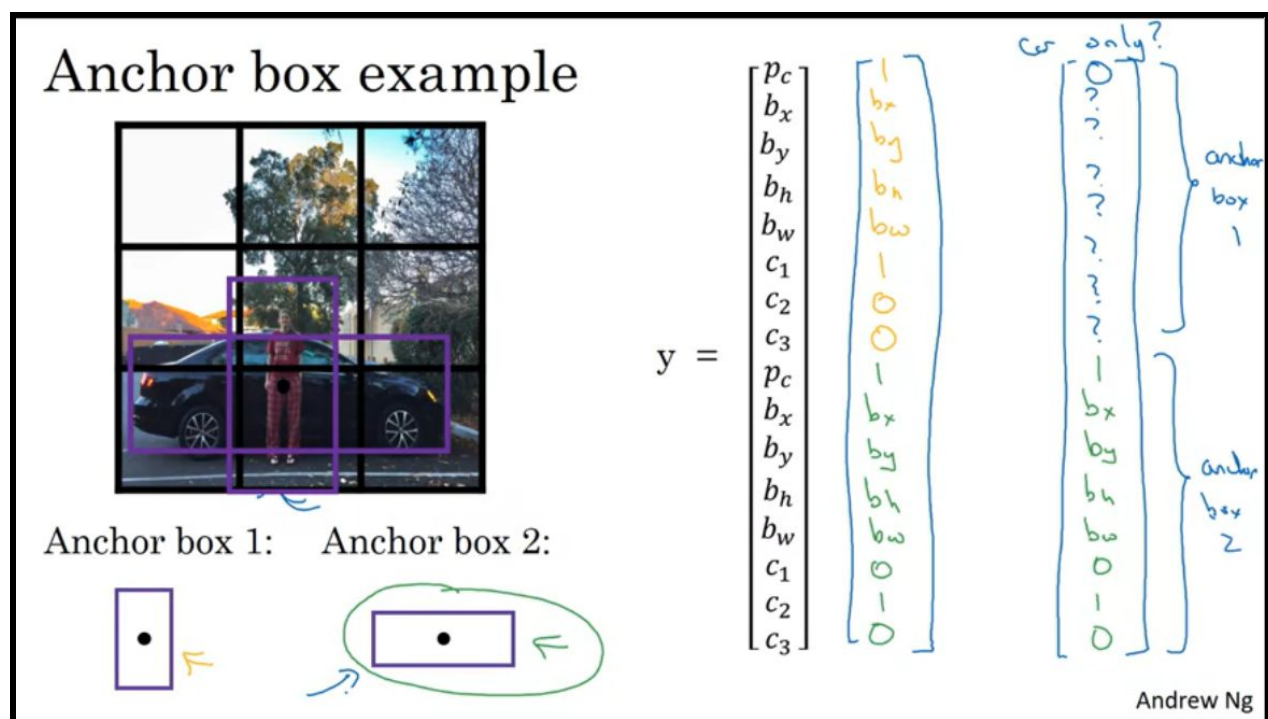
Non Max Suppression

Sometimes many close by grid cells can output a positive label and draw their own bounding box. This leads to there being many bounding boxes near a single object. To fix this, we use non-max suppression. We only consider and draw the box which has the highest confidence (denoted by the first element of the label output).

We consider the list of all bounding boxes the model has output. We consider the box with the highest confidence, and remove all the boxes around it which have a high IoU with it. We repeat till there are no boxes left.

To detect more than one object in a grid cell, we change the output y of a single grid cell from being $1 \times 1 \times (c+5)$ dimensional to $1 \times 1 \times (2c+10)$ dimensional. We simply repeat the same numbers $P_c, b_x, b_y, b_h, b_w, C_1, \dots, C_n$ once more and define the second bounding box.

However, you must decide on an anchor box pattern. Define a vertical rectangle as anchor box 1, and a horizontal rectangle as anchor box 2. Then the label y will output the parameters for anchor box 1 above the parameters for anchor box 2. If there is a single object in a grid cell which fits in anchor box 2, the first set of parameters describing anchor box 1 will be NaNs.



WEEK 4

Face Recognition

You can build a model to perform face recognition or face verification. Face verification is when the model just tries to verify if the input image is a particular person or not. Face recognition is when the model has to identify the person out of a database of K people.

One challenge with face recognition or verification is the problem of one shot learning. Often you only have one photo of a person in your database (the passport sized standard photo). If you have just one photo, you can't use the standard approach of training a CNN on a training set, because there is no training set.

Plus, if you're performing face recognition and you need to identify a new person after your model has been trained, you will need to retrain your entire model with a different dimension of the softmax output.

Instead, we don't output a softmax unit, we calculate a similarity function. Given the standard image, we take a photo of the person during runtime, and compare the two photos. We then calculate a similarity function and output the degree of difference between the two images.

We denote the function as

$$d(img1, img2) \leq \Gamma$$

Where Γ is the threshold of the degree of difference you decide.

Instead of training a complete CNN to output a logistic or softmax unit, you stop at an intermediate fully connected layer of, say, 128 units. So your complete model will take in an image and output a 128 dimensional vector. To tell if two people are similar or not, you then put both the images into the input and define the $d()$ function as the squared error of the norm.

The 128 dimensional output is called the 'encoding' of the input image and denoted as $f(x^{(i)})$ for the input image $x^{(i)}$. This encoding function $f(x)$ is the one you have to learn.

While training, we always look at 3 images at the same time - an anchor image (the photo of the person we are trying to recognize), a positive example, and a negative example. The objective of learning is then to minimize the quantity

$$\begin{aligned} d(A, P) - d(A, N) \\ d(A, P) - d(A, N) + \alpha &\leq 0 \\ |f(A) - f(P)|^2 - |f(A) - f(N)|^2 + \alpha &\leq 0 \end{aligned}$$

Alpha is called the margin and it is a hyperparameter. Make alpha higher if you want the distance between similar and dissimilar images to be bigger.

Triplet Loss

We define the loss and the cost function as follows.

$$L(A, P, N) = \max(|f(A) - f(P)|^2 - |f(A) - f(N)|^2 + \alpha, 0)$$
$$J(A, P, N) = \sum_{i=1}^m L(A^{(i)}, P^{(i)}, N^{(i)})$$

Although you only need one photo of a person to recognize them during run time, you do need multiple pictures of the same people to train your model. Once the model is trained, however, you can run with just one picture of the person in your database.

While generating your training set, don't randomly create A,N and A,P pairs. This is because it is much easier to satisfy the cost function if you have positive and negative examples that look very different.

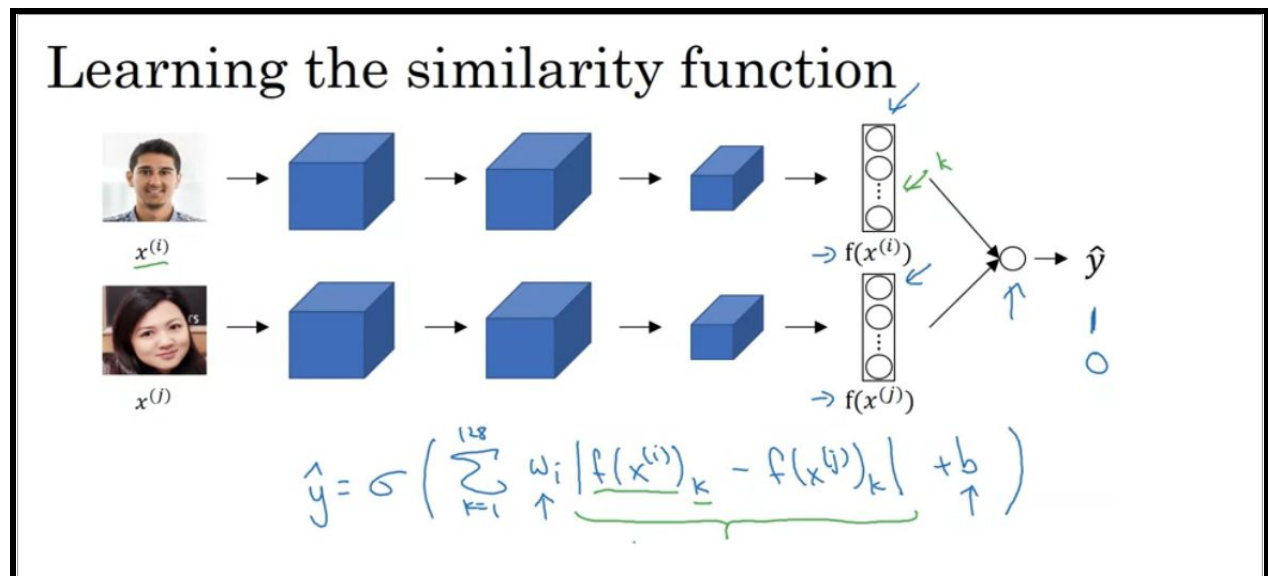
However, training such networks is very painstaking because of the amount of data you need to acquire to make an accurate enough model. Fortunately, many companies in the face recognition field have trained their models on huge data sets (often more than a million examples) and open sourced the parameters. This is one area where you might want to take their parameters and apply transfer learning.

Logistic Regression for Face Verification

Another approach is to take the two input images, compute the encoding function $f(x)$ for both of them, and then feed the two encodings into a logistic unit and make it output a 1 or a 0.

However, instead of feeding both the 128 dimensional vectors concatenated together, we feed the difference between the corresponding elements to the logistic regression unit. (See the image for how we calculate \hat{y}).

You then formulate your training set as pairs of images, with the corresponding output label being 1 if they are of the same person and 0 if they are of the different person.



Neural Style Transfer

Neural style transfer is a specific application of deep learning which takes in two paintings or images and generates painting 1 in the 'style' of painting 2.

I'm not really interested in Neural Style Transfer, so I won't be taking very detailed notes.

The cost function $J(G)$ for NST will measure how good the generated image is.

$$J(G) = \alpha * J_{content}(C, G) + \beta * J_{style}(C, G)$$

To actually generate the final image, you initialize the output image as random pixel values of the dimension you want. You then just run gradient descent on all the pixel values of the image to reduce $J(G)$.

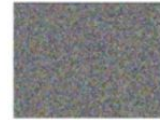
Find the generated image G

1. Initiate G randomly

G: $100 \times 100 \times 3$
 ↑
 RGB

2. Use gradient descent to minimize $J(G)$

$$G := G - \frac{\partial}{\partial G} J(G)$$



[Gatys et al., 2015. A neural algorithm of artistic style]

Andrew Ng

Content cost function

$$J(G) = \alpha J_{\text{content}}(C, G) + \beta J_{\text{style}}(S, G)$$

- Say you use hidden layer l to compute content cost.
- Use pre-trained ConvNet. (E.g., VGG network)
- Let $a^{[l](C)}$ and $a^{[l](G)}$ be the activation of layer l on the images
- If $a^{[l](C)}$ and $a^{[l](G)}$ are similar, both images have similar content

$$J_{\text{content}}(C, G) = \frac{1}{2} \| \underbrace{a^{[l](C)}}_{\text{activation of layer } l \text{ on } C} - \underbrace{a^{[l](G)}}_{\text{activation of layer } l \text{ on } G} \|^2$$

[Gatys et al., 2015. A neural algorithm of artistic style]

Andrew Ng

Here both the activations are rolled out into vectors before we take their norm.

Style matrix

Let $a_{i,j,k}^{[l]}$ = activation at (i, j, k) . $G^{[l]}$ is $n_c^{[l]} \times n_c^{[l]}$

$$G_{kk'}^{[l](S)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijk}^{[l](S)} a_{ijk'}^{[l](S)}$$

$$G_{kk'}^{[l](G)} = \sum_{i=1}^{n_H^{[l]}} \sum_{j=1}^{n_W^{[l]}} a_{ijk}^{[l](G)} a_{ijk}^{[l](G)}$$

"Gram matrix"

$$G_{kk'}^{[l]} = \sum_{k=1}^{n_c^{[l]}} \sum_{k'=1}^{n_c^{[l]}} G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)}$$

$$J_{style}^{[l]}(S, G) = \|G^{[l](S)} - G^{[l](G)}\|_F^2 = \sum_k \sum_{k'} (G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)})^2$$

[Gatys et al., 2015. A neural algorithm of artistic style]

Andrew Ng

Style cost function

$$J_{style}^{[l]}(S, G) = \frac{1}{(2n_H^{[l]}n_W^{[l]}n_C^{[l]})^2} \sum_k \sum_{k'} (G_{kk'}^{[l](S)} - G_{kk'}^{[l](G)})^2$$

$$J_{style}(S, G) = \sum_l \lambda_l J_{style}^{[l]}(S, G)$$

$$J(G) = \alpha J_{content}(G) + \beta J_{style}(S, G)$$

[Gatys et al., 2015. A neural algorithm of artistic style]

Andrew Ng

Course 5

Sequence Models

WEEK 1

Recurrent Neural Networks

Recurrent Neural Networks are mainly applied to sequence models. Sequence models are models in which either the input, or the output, or both are sequences. Sequences are data which change with time.

New notation -

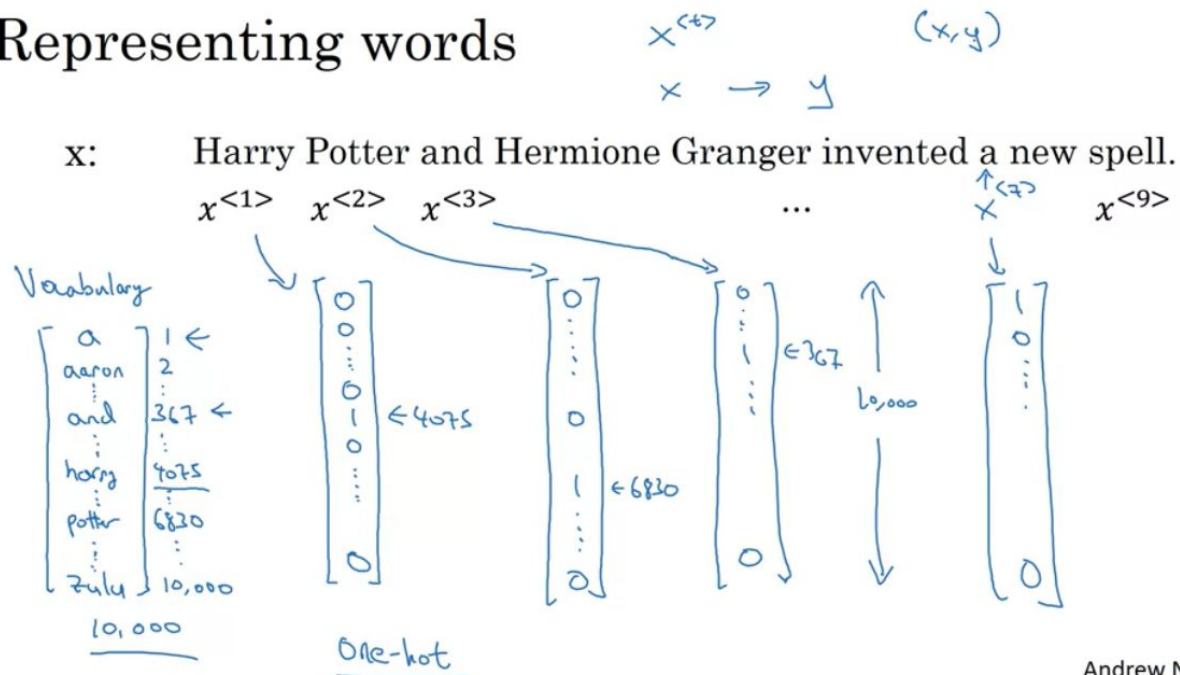
We use $x^{<t>}$ to index into the t^{th} position of the sequence input data.

We use T_x to denote the length of the input sequence, and T_y to denote the length of the output sequence.

$x^{(i)<t>}$ will be the t^{th} index of the i^{th} training example. T_x and T_y can be different for different training examples. Therefore, $T_x^{(i)}$ will be the length of the input for the i^{th} example.

If you're working with words, you represent them differently than numerical data. You create a dictionary of however many words you want to include in your model (which is most commonly around 50,000), and arrange them alphabetically in a list or dictionary. We then look at the input data and replace each word by a column vector which is 50,000 dimensional with only one 1 in the position where the word occurs in the dictionary/list and all other elements as 0.

Representing words



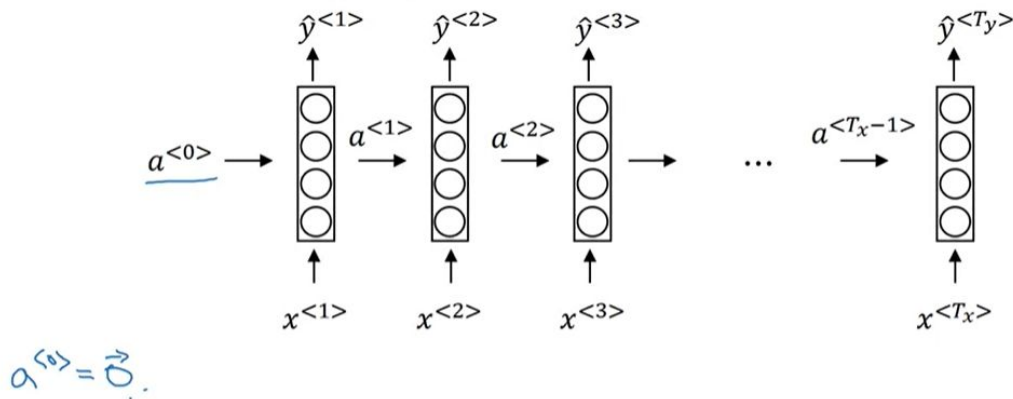
Andrew Ng

So you will have arrays as your input and your output data. The output data can also be a single number or not a sequence at all. Now we learn a mapping from the given X to the given Y using a recurrent neural network.

We don't use a standard neural network for such an application because the input lengths and output lengths can be (and almost always are) different for different training examples. A standard network also doesn't generalize across different positions of the input.

A recurrent neural network uses not only the input $x^{<t>}$ as its input but also takes in the activation generated by the previous input to predict $y^{<t>}$. This is known as a unidirectional RNN. The fact that it only takes in the activations from the previous inputs and not the activations from the later inputs is kind of a disadvantage, but we fix that by using a bidirectional RNN.

Forward Propagation



Andrew Ng

Forward Propagation

For calculating $y^{<t>}$, we need only $a^{<t>}$. For calculating $a^{<t>}$, we need $a^{<t-1>}$ and $x^{<t>}$. Therefore, $a^{<t-1>}$ and $x^{<t>}$ are both taken into consideration while calculating $y^{<t>}$.

So the formulae become

$$a^{<t>} = g(W_{aa}a^{<t-1>} + W_{ax}x^{<t>} + b_a)$$

$$y^{<t>} = g_1(W_{ya}a^{<t>} + b_y)$$

To make the calculation of $a^{<t>}$ a little more efficient, we join W_{aa} and W_{ax} together horizontally and we stack $a^{<t-1>}$ and $x^{<t>}$ together vertically so that we can perform the two operations at the same time. We then write the formula as

$$a^{<t>} = g([W_{aa}, W_{ax}][a^{<t-1>}; x^{<t>}] + b_a)$$

As we would stack matrices and join matrices in MATLAB.

Backward propagation

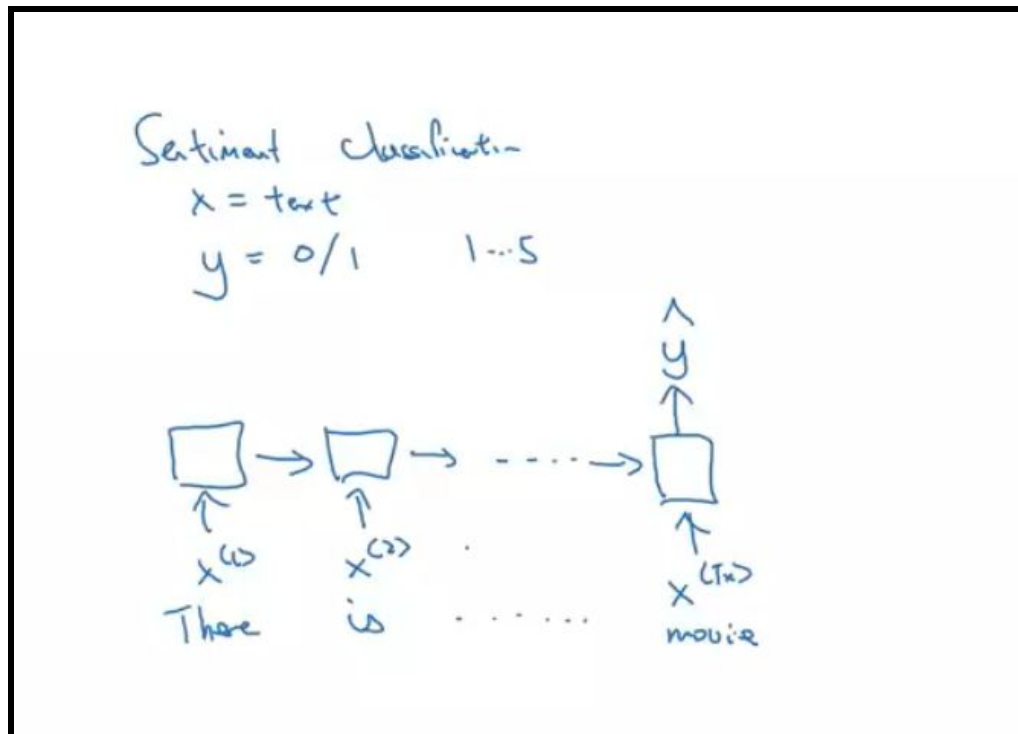
Once we perform forward propagation, we need to define a cost function and a loss function to implement backward propagation. We define the loss function to be the same as logistic regression -

$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log(\hat{y}^{<t>}) - (1 - y^{<t>}) * \log(1 - \hat{y}^{<t>})$$

$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>})$$

When you code in a programming framework, the framework takes care of the backward propagation equations.

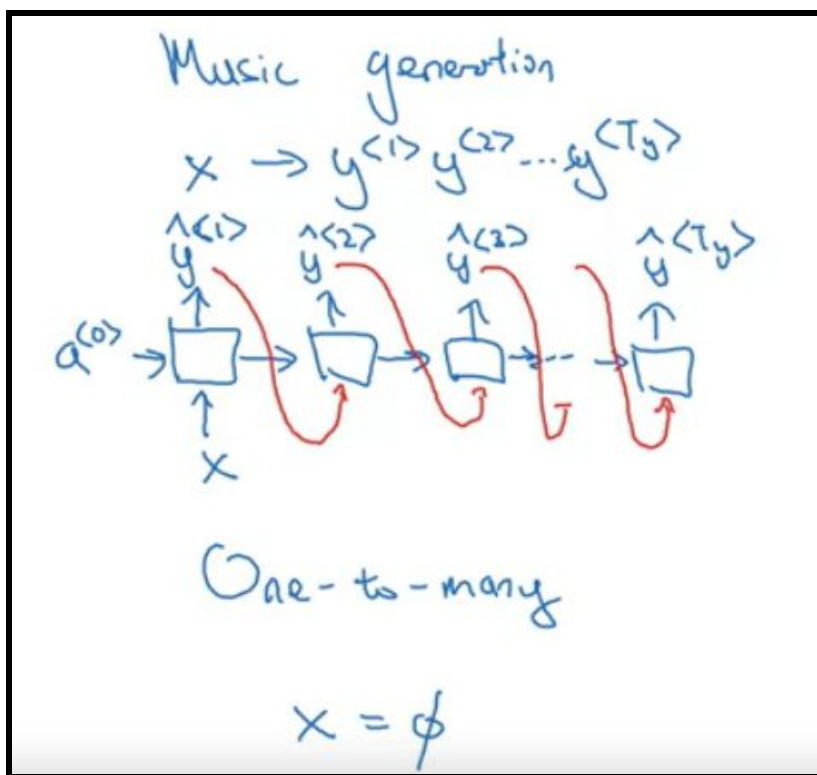
In case T_x and T_y are not equal, for example in sentiment analysis where you take in some text and output the emotion behind the text, the input is a sequence and the output is a single integer. In cases like these, you can take the output just at the last input and not take any output for all the other words.



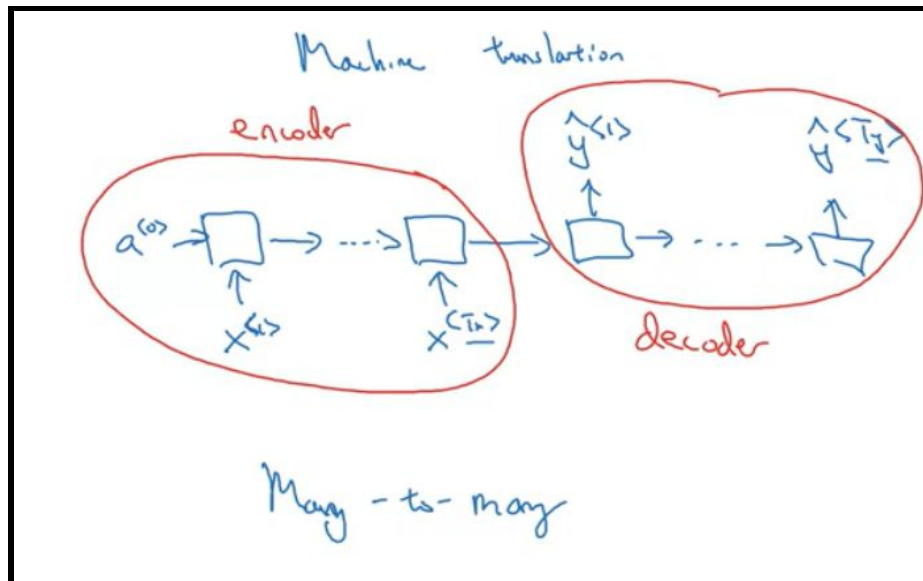
This is a many to one architecture.

There can also be a one to many architecture, music generation, for example, where the input can be just a zero vector or a number telling your network the genre of music you want to generate.

For one to many type architecture, you feed the input to the neural net and take the first output and the first activation. You then feed the activation and the first output to the next input. You keep feeding the previous activation and the previous output as the next input to keep generating notes.

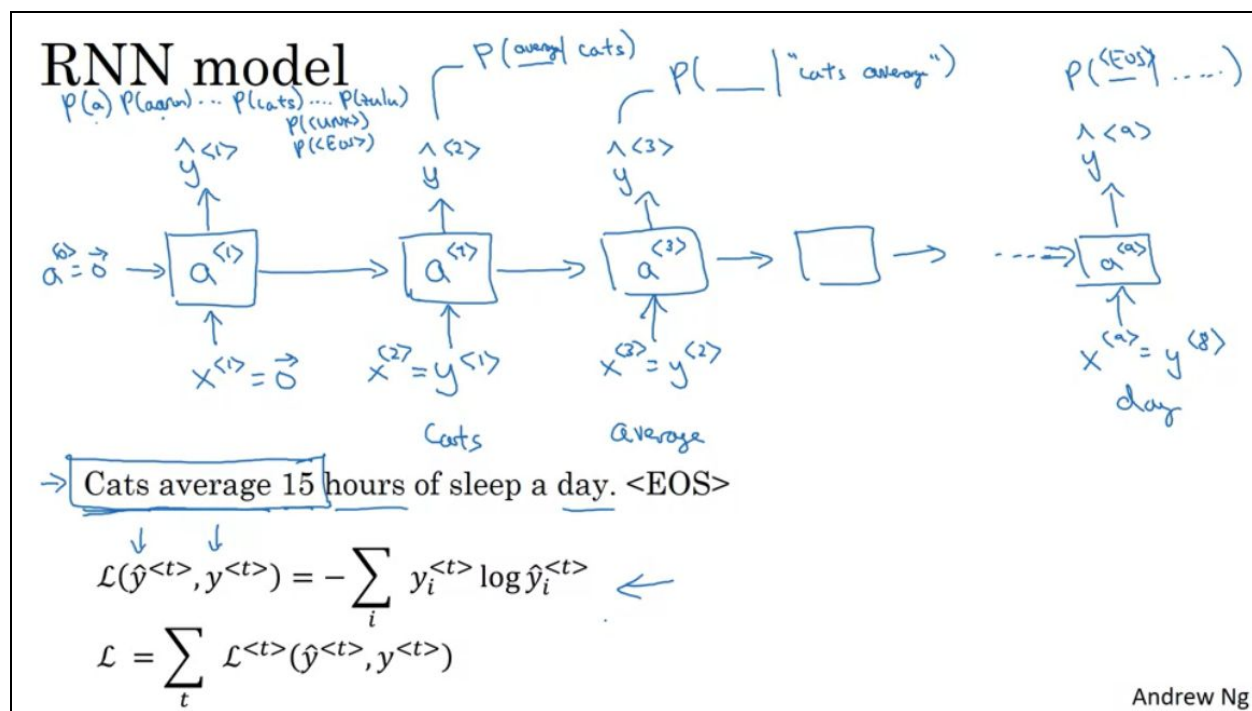


For many to many type architectures where the input and output lengths are different, for example in machine translation, we read in the input first and then generate as many output words as needed.



For sentence prediction or next-word prediction using RNNs, we need a large body of english text to train the model on. Once we train the model on as many sentences as we can find, we can give it a word or a part of a sentence and have it output the next word in the sentence.

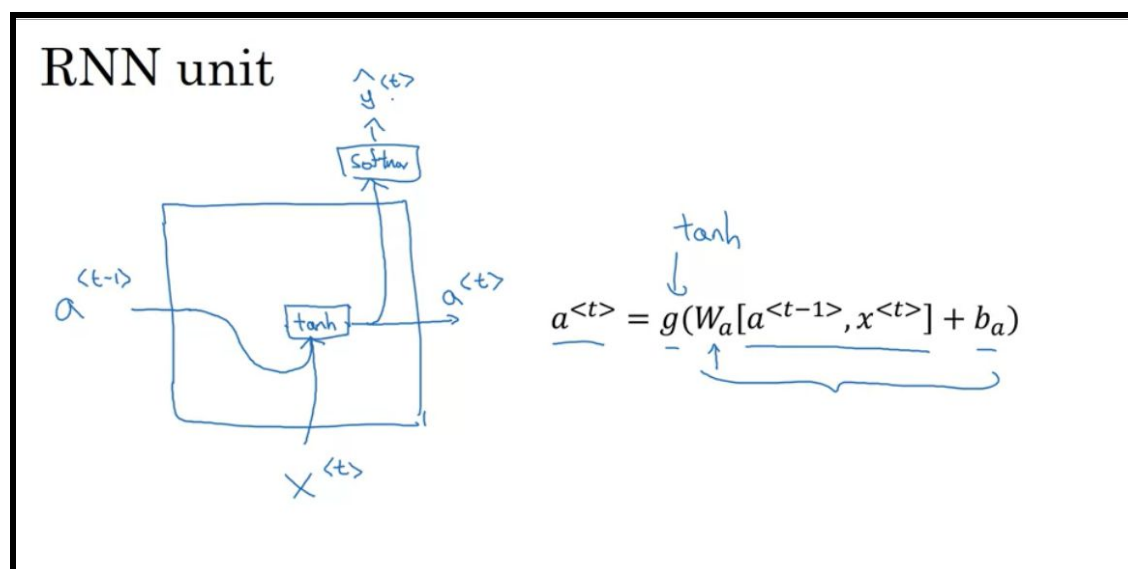
The way we train the RNN is that we first input a zero vector as both the activation and the input and ask it to make a blind prediction of what it thinks the first word is going to be. It makes a blind prediction by outputting a softmax vector containing as many elements as the number of words in the dictionary we decided on. We then give it the actual first word of the sentence we are training on and the activation it calculated in the first step to predict the second word of the sentence, and so on. After it has finished making predictions on all the words in the sentence, we calculate the cost of its predictions. This results in the model being trained on the sentences we have in the training set.



Gated Recurrent Unit (GRU)

A GRU helps a lot with the vanishing gradients problem and helps the RNN form much deeper connections.

A single unit of an RNN is



At any time step t , the GRU has a memory unit called $C^{<t>}$. $C^{<t>}$ is also called the 'gate'. In effect, $C^{<t>}$ memorizes if the subject of the sentence is a singular or plural till that knowledge needs to be used or updated. We define the activation at that time step $a^{<t>}$ to be equal to $C^{<t>}$. At every time step, we define a candidate

$$\bar{C}^{<t>} = \tanh(W_C[\Gamma_r * C^{<t-1>}; X^{<t>}] + b_C).$$

This $\bar{C}^{<t>}$ is a candidate to replace $C^{<t>}$. To decide if we should replace $C^{<t>}$ by $\bar{C}^{<t>}$, we define two more quantities

$$\Gamma_r = \sigma(W_r[C^{<t-1>}; X^{<t>}] + b_r).$$

$$\Gamma_u = \sigma(W_u[C^{<t-1>}; X^{<t>}] + b_u).$$

Since Γ_u is a sigmoid function, it can either be 0 or 1. If it is 0, we keep the old value of $C^{<t>}$, if it is 1, we replace $C^{<t>}$ by $\bar{C}^{<t>}$. We explicitly define $C^{<t>}$ as

$$C^{<t>} = \Gamma_u * \bar{C}^{<t>} + (1 - \Gamma_u) * C^{<t-1>}$$

GRU (simplified)

$C^{<t-1>} = a^{<t-1>}$
 $\hat{C}^{<t>} = \tanh(W_C[C^{<t-1>}, x^{<t>}] + b_C)$
 $\Gamma_u = \sigma(W_u[C^{<t-1>}, x^{<t>}] + b_u)$
 $C^{<t>} = \Gamma_u * \hat{C}^{<t>} + (1 - \Gamma_u) * C^{<t-1>}$

$\Gamma_u = 1$
 $C^{<t>} = 1$
 The cat, which already ate ..., was full.

$C = \text{memory cell}$
 $\rightarrow C^{<t>} = a^{<t>}$

$\hat{C}^{<t>} = \tanh(W_C[C^{<t-1>}, x^{<t>}] + b_C)$
 $\Gamma_u = \sigma(W_u[C^{<t-1>}, x^{<t>}] + b_u)$

$C^{<t>} = \Gamma_u * \hat{C}^{<t>} + (1 - \Gamma_u) * C^{<t-1>}$

$\Gamma_u = 1$ (update gate)

[Cho et al., 2014. On the properties of neural machine translation: Encoder-decoder approaches] ←

[Chung et al., 2014. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling] ←

Andrew Ng

LSTM (Long Short Term Memory)

LSTM is another kind of architecture that helps with vanishing gradients and with forming long range connections.

LSTM uses three gates in total. The equations of an LSTM unit are

$$\overline{C}^{<t>} = \tanh(W_c[C^{<t-1>}; a^{<t>}] + b_c).$$

$$\Gamma_u = \sigma(W_u[a^{<t-1>}; X^{<t>}] + b_u)$$

$$\Gamma_f = \sigma(W_f[C^{<t-1>}; X^{<t>}] + b_f)$$

$$\Gamma_o = \sigma(W_o[C^{<t-1>}; X^{<t>}] + b_o)$$

$$C^{<t>} = \Gamma_u * \overline{C}^{<t>} + \Gamma_f * C^{<t-1>}$$

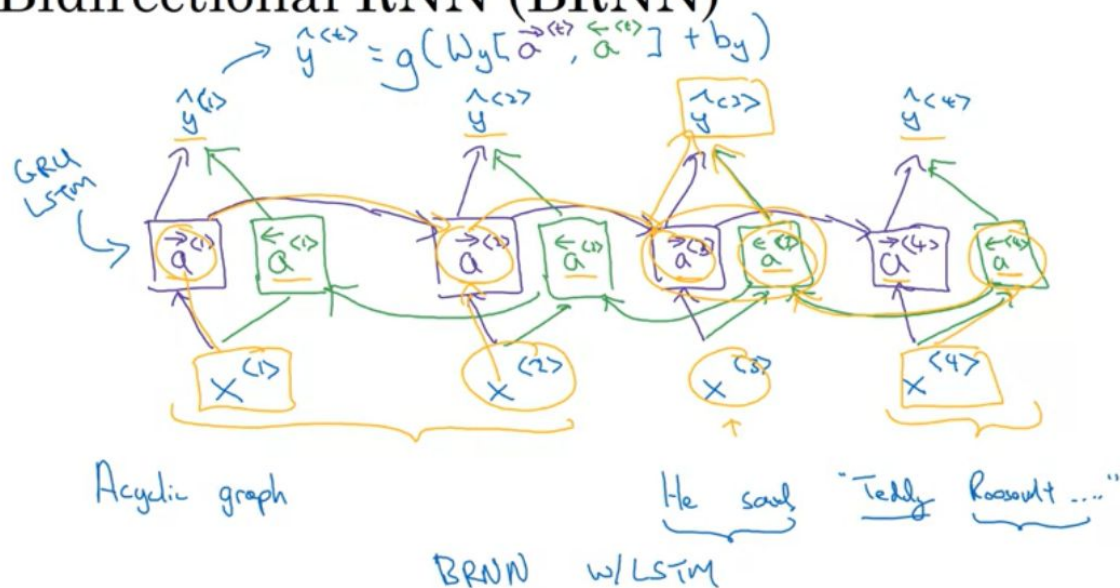
$$a^{<t>} = \Gamma_o * \tanh(C^{<t>})$$

Bidirectional RNNs (BRNNs)

For some applications such as name entity recognition, a unidirectional RNN is not as accurate. This is because to make sense of a sentence, you need information from both the words before and the words after the word being considered.

To achieve this, we first calculate the forward activations as usual, but then we start from the last word of the sequence and calculate activations from the last word to the first word. We then combine both of these activations while calculating our output y.

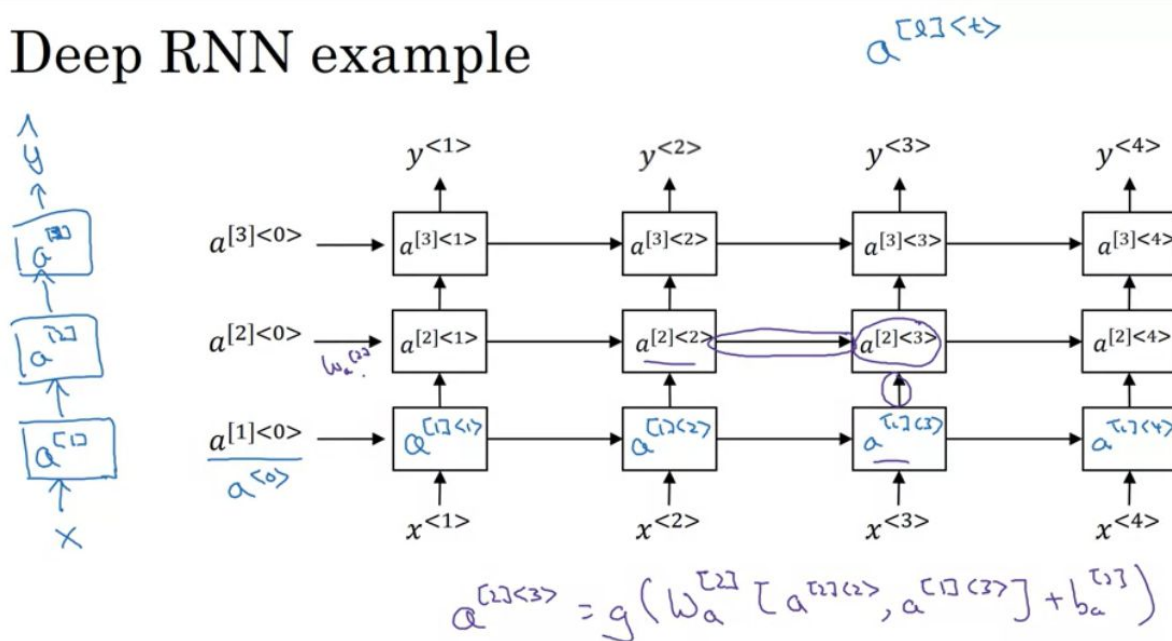
Bidirectional RNN (BRNN)



Andrew Ng

All the RNN architectures you saw till now had only one layer, but you can extend them upwards to make deep RNNs. The process is pretty self-explanatory.

Deep RNN example



Andrew Ng

Three layers is already pretty deep for an RNN, but if you want to make it deeper, you can just extend the network upwards and not pass its activation to the future time steps.

WEEK 2

Word Embedding

Representing words in a sentence just as one-hot vectors doesn't allow your model to make connections and form relationships among different words which may be closely related in real life. To add an element of connection between different words, we use a different method of representing words in a sentence. We still use the one hot vector for each word. We represent the one hot vector for a particular word as O_n for a word in which the n th position in the one hot vector is 1.

But, for each one hot vector, we also construct a list of features such as age, weight, gender, noun, verb, cost, etc. We then train each of these values, and words which are closely related to each other end up getting the same or similar values. We represent these relationship vectors as e_n for the embedding for the n th word in the dictionary.

Featurized representation: word embedding						
	Man (5391)	Woman (9853)	King (4914)	Queen (7157)	Apple (456)	Orange (6257)
Gender	-1	1	-0.95	0.97	0.00	0.01
Royal	0.01	0.02	0.93	0.95	-0.01	0.00
Age	0.03	0.02	0.7	0.69	0.03	-0.02
Food	0.04	0.01	0.02	0.01	0.95	0.97
size				
cost				
verb				

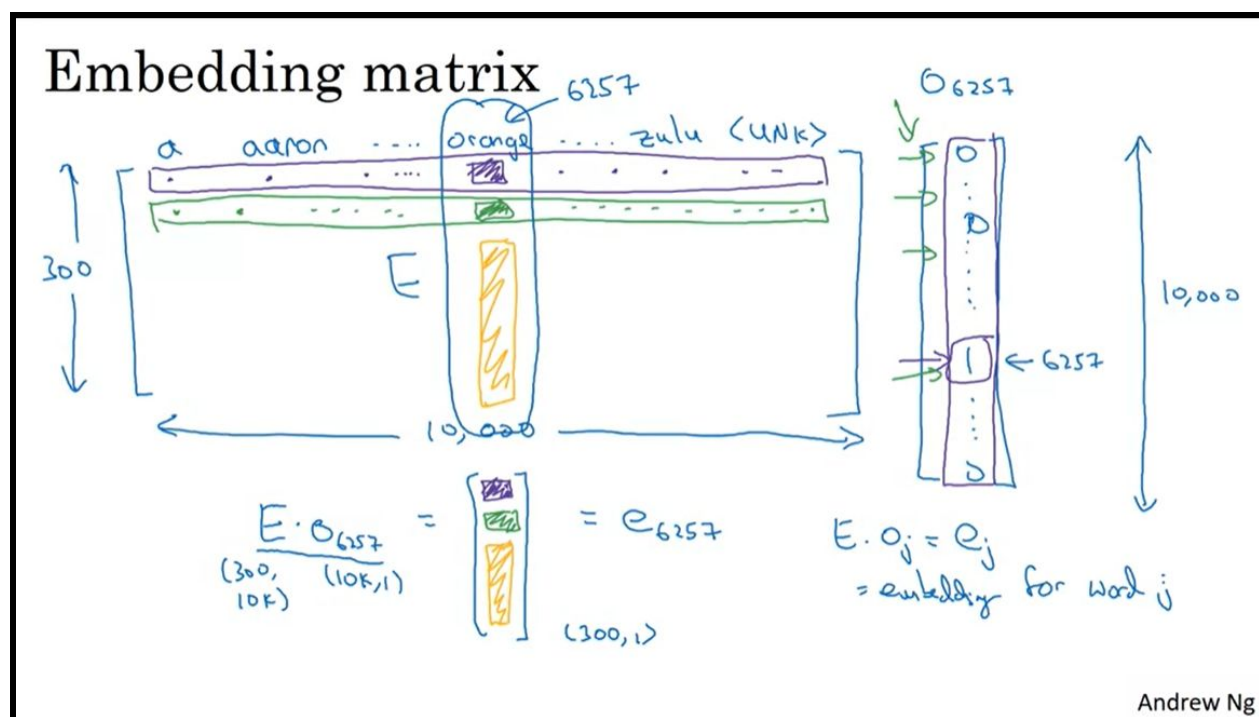
e_{5391} e_{9853}

I want a glass of orange juice.
 I want a glass of apple juice.

Andrew Ng

What you actually learn is an embedding matrix E . An embedding matrix is a $n_f \times n_v$ dimensional vector, where n_f is the number of features you want to give to a particular word, and n_v is the size of the vocabulary. To select the embedding vector of a particular word v , you can multiply the embedding matrix E with a one hot vector n_v to get the features of that word v .

The embedded features for a word w are represented as a column vector. We denote this column vector by e_w .



We initialize E randomly and then use gradient descent to learn E over time.

Learning Word Embeddings

#I'm gonna stop taking notes now. I need to understand the basics better. I'm just gonna watch the videos and complete the course for now.

Course 6

Browser Based Models With TensorFlow.js

WEEK 1

The overall structure of the program is very similar to python, only the syntax is adapted to JS.

We include our code in a script tag or a separate JS file. We define a function in which we write all of our code and just call the function before closing the script tag or ending the JS file.

The function we define is an async function. Inside the function we follow this general structure -

1. Load the training data (and also the testing data if you want) from a csv file.
We define a url to get the data from. Then use the `tf.data.csv` function to load the data and turn it into a tensor. We pass the function the column name that we want to turn into the label. We get the total number of features from the data by counting the number of columns in the csv file and subtracting 1. We get the total number of training examples.
2. We then convert the training data (and testing data) into arrays by using the `.map` method on the `trainingData` and `testingData` variables.
3. We then define the model architecture and compile it.
4. We then train the model by calling the `.fitDataset` on the model object.
5. We can then ask for predictions by calling the `.predict` method on the model object and passing it some input.

```
<html>
<head></head>
<script src="https://cdn.jsdelivr.net/npm/@tensorflow/tfjs@latest"></script>
<script lang="js">
  async function run(){
    const csvUrl = 'iris.csv';
    const trainingData = tf.data.csv(csvUrl, {
      columnConfigs: {
        species: {
          isLabel: true
        }
      }
    });

    const numOfFeatures = (await trainingData.columnNames()).length - 1;
    const numOfSamples = 150;
    const convertedData =
      trainingData.map(({xs, ys}) => {
        const labels = [
          ys.species == "setosa" ? 1 : 0,
          ys.species == "virginica" ? 1 : 0,
          ys.species == "versicolor" ? 1 : 0
        ]
        return{ xs: Object.values(xs), ys: Object.values(labels)};
      }).batch(10);

    const model = tf.sequential();
    model.add(tf.layers.dense({inputShape: [numOfFeatures], activation: "sigmoid", units: 5}))
    model.add(tf.layers.dense({activation: "softmax", units: 3}));

    model.compile({loss: "categoricalCrossentropy", optimizer: tf.train.adam(0.06)});
```

```

model.compile({loss: "categoricalCrossentropy", optimizer: tf.train.adam(0.06)});

await model.fitDataset(convertedData,
    {epochs:100,
    callbacks:{
        onEpochEnd: async(epoch, logs) =>{
            console.log("Epoch: " + epoch + " Loss: " + logs.loss);
        }
    }});

// Test Cases:

// Setosa
const testVal = tf.tensor2d([4.4, 2.9, 1.4, 0.2], [1, 4]);

// Versicolor
// const testVal = tf.tensor2d([6.4, 3.2, 4.5, 1.5], [1, 4]);

// Virginica
// const testVal = tf.tensor2d([5.8,2.7,5.1,1.9], [1, 4]);

const prediction = model.predict(testVal);
const pIndex = tf.argmax(prediction, axis=1).dataSync();

const classNames = ["Setosa", "Virginica", "Versicolor"];

// alert(prediction)
alert(classNames[pIndex])

}
run();

```

WEEK 2

JUST CHECK THE COURSE FILES AT THEIR GITHUB

[dlaicourse/TensorFlow Deployment/Course 1 - TensorFlow-JS/Week 2](https://github.com/dlaicourse/TensorFlow-Deployment/Course-1-TensorFlow-JS/Week-2)

You shouldn't really train your neural net in the browser at run time. You should train your model using python and then export the model as a json and upload it to your web app. So I'm not gonna take a lot of notes this week. Plus there are a lot of extra steps you have to take to implement training on big datasets in JS.

WEEK 3

get the pre trained tfjs models - <https://github.com/tensorflow/tfjs-models>

Essentially, there are only 2 one liners you have to run to convert a model trained in python into JSON.

Train your model as much as you want and then run these -

```
saved_model_path = #whatever you want w/ a .h5 extension
model.save(saved_model_path)
!tensorflowjs_converter --input_format = keras
{saved_model_path} ./
```

Once you get your model into JSON format, write some JavaScript to import the model into your web app.

```
<script>
  async function run() {
    const MODEL_URL = '//path to model';
    const model = await tf.loadLayersModel(MODEL_URL);
    //That's it, the model has been loaded.
    //Now you can predict.
    var input = tf.tensor2d([10],[1,1]);
    const result = model.predict(input);
    alert(result);
  }
  run();
</script>
```

That's it. Done!

WEEK 4

You can do browser based transfer learning much like in python. With some scripts open sourced by Google, you can even make webcam models. For transfer learning, first load the model, then select a layer to train from. Then create a new DNN to train from new examples.

Instead of treating the new model as being appended to the end of the pre-trained model like in python, tensorflow.js treats them both as two distinct models. We pass the input into the pre-trained model and get an embedding/activation from the model. We then pass that embedding as the input to our own model and get the final classification from it. The code to do that is -

//Loading the pre-trained model and defining the input and output layers.

```
async function loadMobilenet() {  
  const mobilenet = await  
    tf.loadLayersModel('https://storage.googleapis.com/tfjs-models/  
                        /tfjs/mobilenet_v1_0.25_224/model.json');  
  const layer = mobilenet.getLayer('conv_pw_13_relu');  
  return tf.model({inputs: mobilenet.inputs, outputs: layer.output});  
}
```

//Call the function we defined above and make a random prediction to warm the model up. The first time we make a prediction we experience a slight lag, so we make a random prediction right now so we don't experience that lag when we actually want to use the model.

```
async function init(){  
  await webcam.setup();  
  mobilenet = await loadMobilenet();  
  tf.tidy(() => mobilenet.predict(webcam.capture()));  
}
```

//We then define our own model

```
async function train() {  
  model = tf.sequential({  
    layers: [  
      tf.layers.flatten({inputShape: mobilenet.outputs[0].shape.slice(1)}),  
      tf.layers.dense({ units: 100, activation: 'relu'}),  
      tf.layers.dense({ units: 3, activation: 'softmax'})  
    ]  
  });  
}
```

//We give the input to the pre-trained model, get an embedding, and pass that embedding to our model.

```
const embeddings = mobilenet.predict(img);  
const predictions = model.predict(embeddings);
```

That's it for transfer learning! If you want to use the webcam for an application, you have all the files required in the TensorFlow-js folder. It's not too complicated. Rewatch the lectures if you want.

Course 7

NLP With Classification & Vector Spaces

WEEK 1

Week 1 was a simple method in which we just assigned each word a number corresponding to the number of times it occurs in the text and used a simple deep neural network to classify.

WEEK 2

Naive Bayes

Naive bayes assumes that all the features for your model are independent. This is rarely the case, but it works nonetheless. To apply naive bayes for a simple classification task, we take the following steps -

We first make a data frame with one row for each word. For each word, we calculate the number of times it appears in the positive class and the number of times it appears in the negative class. We then divide them both by the total number of words in each class (which is just the sum of all the rows in the data frame).

$P(w_i \text{class})$					
word	Pos	Neg	word	Pos	Neg
I	3	3	I	0.24	0.25
am	3	3	am	0.24	0.25
happy	2	1	happy	0.15	0.08
because	1	0	because	0.08	0.00
learning	1	1	learning	0.08	0.08
NLP	1	1	NLP	0.08	0.08
sad	1	2	sad	0.08	0.17
not	1	2	not	0.08	0.17
Nclass	13	12	Sum	1	1

deeplearning.ai

The POS and NEG column now represent the probability that a particular word is in the text/input, given that the input is positive and negative respectively.

However, instead of just taking this probability, we normalize it a little to avoid getting a probability of 0, as with 'because' in the negative class.

Laplacian Smoothing

$$P(w_i | \text{class}) = \frac{\text{freq}(w_i, \text{class})}{N_{\text{class}}} \quad \text{class} \in \{\text{Positive}, \text{Negative}\}$$

$$P(w_i | \text{class}) = \frac{\text{freq}(w_i, \text{class}) + 1}{N_{\text{class}} + V}$$

N_{class} = frequency of all words in class

V = number of unique words in vocabulary

Introducing $P(w_i | \text{class})$ with smoothing

word	Pos	Neg
I	3	3
am	3	3
happy	2	1
because	1	0
learning	1	1
NLP	1	1
sad	1	2
not	1	2
Nclass	13	12

word	Pos	Neg
I	0.19	

$P(I | \text{Pos}) = \frac{3 + 1}{13 + 8}$
 $V = 8$

We then use this formula to get an output -

$$\left(\frac{n(\text{POS})}{n(\text{NEG})} \right) \prod_{i=0}^m \frac{P(w_i, \text{POS})}{P(w_i, \text{NEG})}$$

Where i goes from 0 to m , where m is the number of unique words in the sentence. $n(\text{POS})$ and $n(\text{NEG})$ is the total number of positive and negative training examples we have. This formula gives us a single number. If this number is greater than 1, the words in the sentence are more positive than negative, and we label the sentence as belonging to the positive class.

Just to avoid the answer becoming too small due to all the probabilities being less than one, we take the log of the above formula.

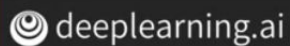
$$\bullet \log\left(\frac{P(pos)}{P(neg)} \prod_{i=1}^n \frac{P(w_i|pos)}{P(w_i|neg)}\right) \Rightarrow \log \frac{P(pos)}{P(neg)} + \sum_{i=1}^n \log \frac{P(w_i|pos)}{P(w_i|neg)}$$

log prior + log likelihood

This is the list of steps you have to take to implement naive bayes. There is no gradient descent step.

Summary

0. Get or annotate a dataset with positive and negative tweets
1. Preprocess the tweets: `process_tweet(tweet) → [w1, w2, w3, ...]`
2. Compute `freq(w, class)`
3. Get `P(w | pos)`, `P(w | neg)`
4. Get `λ(w)`
5. Compute `logprior = log(P(pos) / P(neg))`



The next step is to take a sentence and classify it. That's it!
This method is obviously used mostly for binary classification.

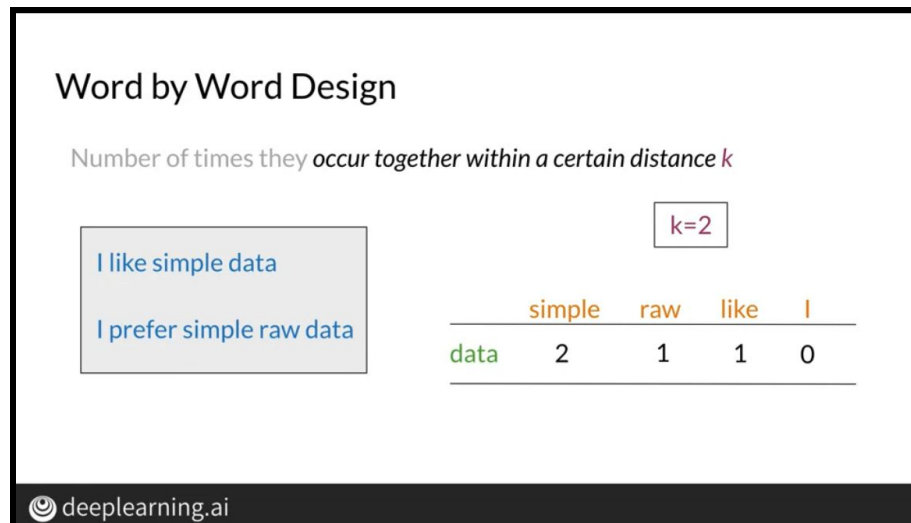
WEEK 3

Vector Space Models

Vector space models represent words or sentences as vectors (embeddings). They are used for applications like chatbot programming, machine translation, and information extraction.

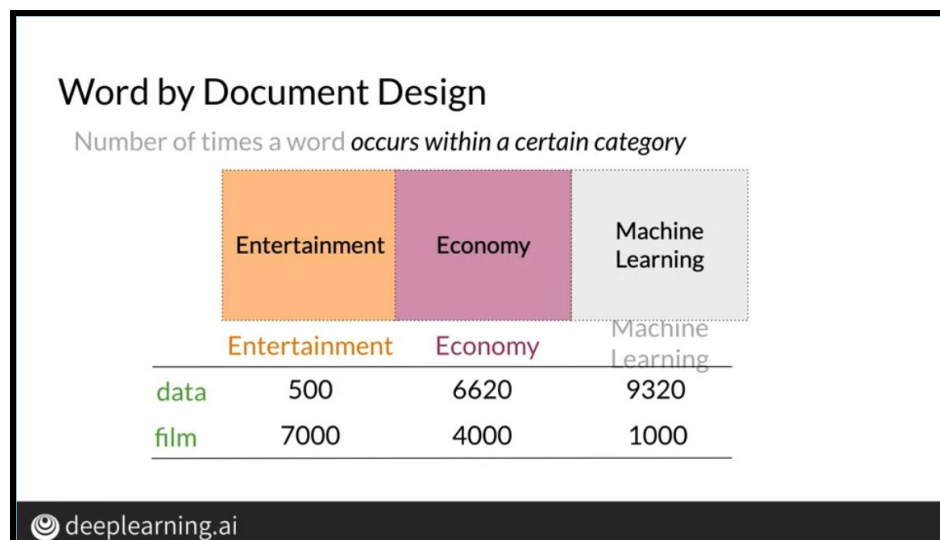
To represent these words as matrices, we either treat words individually or as belonging to a certain sentence (document). For word by word vectors, we construct a co-occurrence matrix.

We construct the co-occurrence matrix by considering a word in the corpus as a row of the matrix. Each column of the matrix corresponds to a word too. Each entry in the row is the number of times the particular word occurs within a distance of k words in the corpus.



The co-occurrence matrix would then be $m \times m$ dimensional where m is the number of unique words in the corpus.

For a word by document approach, instead of counting the number of times the word appears next to every other word in the corpus, we count how many times the word appears in examples with a certain label, kind of like naive bayes.



Once we represent a word as a vector, we can then calculate the similarity in meaning between certain words by using euclidean distance or cosine similarity.

When using the word by document design, however, we choose the individual words to be our axes and the labels or the categories to be the vectors. So in the above example, there will be a vector for the entertainment category, a vector for the economy category, and a vector for the machine learning category. The coordinates of the machine learning category will be (9320, 1000).

Euclidean distance is not a good measure for our purpose because the total number of words in each category can be different. We therefore use the cosine distance (the cos of the angle between them) instead.

Once we have these vectors, we can perform addition and subtraction between two or more vectors to derive relationships from them.

Principal Component Analysis

PCA is an algorithm to reduce the dimensions of a vector. You can transform a word vector from its n dimensions into 3 or 2 or any number of dimensions you want.

WEEK 4

Machine Translation & Document Search

Machine translation works by calculating word embeddings for both the languages in consideration. We then take a word from the first language and use a transformation matrix to transform the word vector into another vector. This new vector will be the embedding of the word in the second language. We then search for words in the second language with an embedding similar to our transformed embedding, and that gives us our translated word.

Our training process then consists of finding this transformation matrix. We do so by randomly initializing it, calculating the loss and the gradient, and updating the values in the matrix, just like a DNN. We define the loss as -

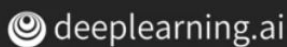
$$L = \|XR - Y\|_F^2$$

Where X is the matrix containing your training word vectors as rows of the matrix, R is the matrix to be found, and Y is the matrix containing the corresponding translated words of the X matrix. The gradient is calculated from the formula -

Gradient

$$Loss = \|\mathbf{XR} - \mathbf{Y}\|_F^2$$

$$g = \frac{d}{dR} Loss = \frac{2}{m} (\mathbf{X}^T (\mathbf{XR} - \mathbf{Y}))$$



The update to the matrix is then $R = R - \alpha * g$.

Once we learn the matrix, we can transform a vector from one language to another, after which we need to find matches for that word in the second language. To find matches quickly and efficiently, we use K nearest neighbours search.

To implement k nearest neighbours, we make a hash table and categorize each vector with its own locality sensitive hash. We then only search for neighbours in that hash.

Course 8

Natural Language Processing with Probabilistic Models

WEEK 1

Autocorrect

A basic autocorrect works in the following 4 steps -

How it works

1. Identify a misspelled word
2. Find strings n edit distance away
3. Filter candidates
4. Calculate word probabilities

deah → dear ✓
yeah
|dear|
dean
... etc

Identifying a misspelled word is easy, just check if it appears in your dictionary/vocabulary.

Then find strings n edit distances away. An edit is defined as -

1. Inserting a character.
2. Deleting a character.
3. Swapping two adjacent characters.
4. Replacing a character.

Get the list of candidate corrections to the misspelled words using this n edit distance and then remove the candidate words that are not in your dictionary.

Out of the remaining list of words, calculate the probability of that word appearing in the sentence and order the candidate replacements according to those probabilities.

You calculate the probabilities very simply - just by dividing the number of occurrences of the word in your corpus divided by the total number of words in your corpus. You can precalculate these probabilities and store them in a dictionary. Once you have the candidate replacements, you can sort the candidate list by using a key of that dictionary.

Implementing N-edit Distance

To calculate the minimum edit distance, we associate each operation with a cost. The cost of deleting or inserting a character is 1, and the cost of replacing a character is 2. You then calculate the cost of the edit using dynamic programming with the general formula being

$$D[i, j] = \min \begin{cases} D[i - 1, j] + del_cost \\ D[i, j - 1] + ins_cost \\ D[i - 1, j - 1] + \begin{cases} rep_cost; & \text{if } src[i] \neq tar[j] \\ 0; & \text{if } src[i] = tar[j] \end{cases} \end{cases}$$

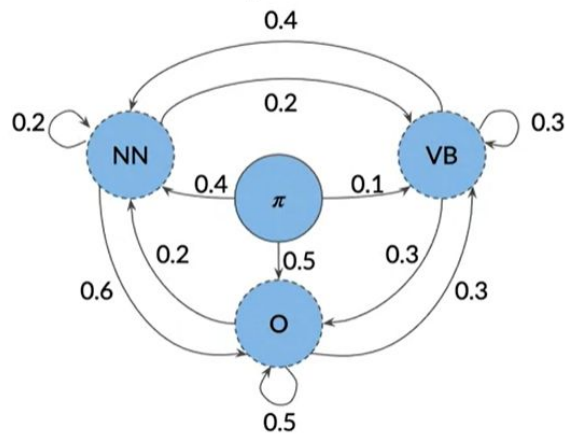
WEEK 2

POS Tagging

Part of speech tagging refers to giving each word in a sentence a tag of noun, verb, adverb, etc. It is used for name entity recognition, speech recognition, and more.

To do POS tagging, we use Markov models. Markov models are models which can assign a state to the next example based only on the state of the previous example. To do this, we construct a transition matrix. A transition matrix has rows which correspond to the different tags and each entry in the row is the probability that the next element has a particular tag, given that the previous element had the row tag.

Transition probabilities



$A =$

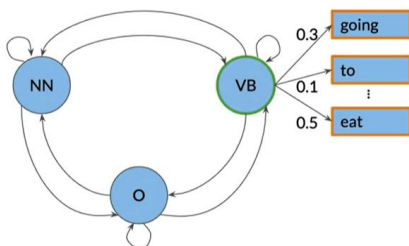
	NN	VB	O
π (initial)	0.4	0.1	0.5
NN (noun)	0.2	0.2	0.6
VB (verb)	0.4	0.3	0.3
O (other)	0.2	0.3	0.5

The first row is the initial column and it represents the probabilities for the first word in the sentence, since the first word won't have any previous tag to reference.

For POS tagging, we call the actual states (NN, VB, O, etc.) the hidden states, since they are not immediately apparent to the machine. This model is then called the hidden markov model. This matrix is represented by A .

We also calculate another matrix B , called the emission matrix, which has the probabilities that a given tag corresponds to a particular word in the corpus.

Emission probabilities



$B =$

	going	to	eat	...
NN (noun)	0.5	0.1	0.02	
VB (verb)	0.3	0.1	0.5	
O (other)	0.3	0.5	0.68	


Populating the transition matrix is easy, we just count the total number of times a tag i follows a tag j and store it in the (i, j) th cell of the transition matrix A . To get the probabilities, we divide each entry of a row by the sum of that row.

To account for division by 0, however, we use smoothing (don't use smoothing for the first row).

Smoothing

	NN	VB	O	
π	$1+\epsilon$	$0+\epsilon$	$2+\epsilon$	$3+3*\epsilon$
NN	$0+\epsilon$	$0+\epsilon$	$6+\epsilon$	$6+3*\epsilon$
VB	$0+\epsilon$	$0+\epsilon$	$0+\epsilon$	$0+3*\epsilon$
O	$6+\epsilon$	$0+\epsilon$	$8+\epsilon$	$14+3*\epsilon$

$$P(t_i|t_{i-1}) = \frac{C(t_{i-1}, t_i) + \epsilon}{\sum_{j=1}^N C(t_{i-1}, t_j) + N * \epsilon}$$

 deeplearning.ai

Viterbi Algorithm

Not gonna take notes for this week

WEEK 3

Autocomplete

To make autocomplete models, we use N-gram sequences. We take a text corpus and extract from it all n-grams. A unigram is just the list of unique words. A bigram is a list

of the unique pairs of consecutive words. A trigram is a list of the unique triads of consecutive words that appear in the corpus.

We then calculate the probability of an n gram sequence as the probability that the last word of the sequence occurs, given the n-1 previous words.

N-gram probability

$$\text{Probability of N-gram: } P(w_N | w_1^{N-1}) = \frac{C(w_1^{N-1} w_N)}{C(w_1^{N-1})}$$

$$C(w_1^{N-1} w_N) = C(w_1^N)$$

So,

$$P(\text{happy} | \text{I am}) = \frac{C(\text{I am happy})}{C(\text{I am})}$$

The probability of the trigram 'I am happy' is the number of times 'I am happy' appears in the corpus divided by the number of times 'I am' appears in the corpus.

The next step is to calculate the probabilities of entire sentences. To do that, we could just apply the bayes rule - i.e.

$$P(w_1 w_2 w_3 w_4 \dots) = P(w_1) * P(w_2 | w_1) * P(w_3 | w_1 w_2) * P(w_4 | w_1 w_2 w_3) * \dots$$

$$P(\text{the teacher drinks tea}) = P(\text{the}) * P(\text{teacher} | \text{the}) * P(\text{drinks} | \text{the teacher}) * P(\text{tea} | \text{the teacher drinks})$$

But the problem with this approach is that further down the road, it is unlikely that the sentence 'the teacher drinks' appears in our text corpus. Therefore, we use a Markov approximation, and only consider the previous n-1 words instead of all the previous words in the sentence (where n is the n gram model we are using). So the probability of 'the teacher drinks tea' becomes -

$$P(\text{the teacher drinks tea}) = P(\text{the}) * P(\text{teacher}|\text{the}) * P(\text{drinks}|\text{teacher}) * P(\text{tea}|\text{drinks})$$

Approximation of sequence probability

the teacher drinks tea

$$P(\text{tea}|\text{the teacher drinks}) \approx P(\text{tea}|\text{drinks})$$

$$\begin{aligned} &P(\text{teacher}|\text{the}) \\ &P(\text{drinks}|\text{teacher}) \\ &P(\text{tea}|\text{drinks}) \end{aligned}$$

$$P(\text{the teacher drinks tea}) =$$

$$P(\text{the})P(\text{teacher}|\text{the})P(\text{drinks}|\text{the teacher})P(\text{tea}|\text{the teacher drinks})$$



$$P(\text{the})P(\text{teacher}|\text{the})P(\text{drinks}|\text{teacher})P(\text{tea}|\text{drinks})$$

Also remember that

$$P(\text{teacher}|\text{the}) = C(\text{the teacher})/C(\text{the})$$

However, to deal with a few problems related to the addition of probabilities, we add n-1 start tokens and 1 end token to the sentence. If we are considering trigrams, our sentence becomes

<s> <s> The teacher drinks tea. </s>

The probability of this sentence is now (since we are considering trigrams)-

$$P(\text{the}|\text{<s> <s>}) * P(\text{teacher}|\text{<s> <s> the}) * P(\text{drinks}|\text{the teacher}) * P(\text{tea}|\text{teacher drinks}) * P(\text{</s>}|\text{drinks tea})$$

The steps for making a language model for an autocomplete task are -

1. Make a count matrix having the counts of each n-gram and word.
2. Transform the count matrix into a probability matrix.
3. Apply this matrix to predict/generate sentences.

The count matrix has rows made up of all unique $n-1$ grams from the corpus. The columns of the count matrix have all the unique words of the corpus. Once you populate the count matrix, transform it into a probability matrix by dividing each row by its sum.

Once you have the probability matrix, you can use it to create your language model by using the probability matrix to estimate the probability of a particular sentence, or by sampling n -grams from the probability matrix to generate new text.

To test our model, we split our corpus into train, validation, and test data. For huge corpora, the test and validation set are around 1% to 5% of the total data. We then construct our probability matrix from the training data and test it on the validation set. To test it on the validation set, we first concatenate all the sentences in the validation set, and then calculate the probability of that sentence using our probability matrix. We then use $P(\text{test set})^{-1/m}$ where m is the total number of unique words in our corpus as the metric.

This metric is known as the perplexity score. The perplexity score refers to how naturally written the text sounds. Perplexity is closely related to entropy.

To deal with unknown words which may appear in the test set or in a query, we use `<UNK>` tokens. To include `<UNK>` tokens in your model, define a vocabulary first, for example, words in your corpus which appear at least 3 times in the corpus will be considered in the vocabulary. Then, replace all the words in the corpus that are not in the vocabulary with the `<UNK>` token. Then train your model as before. Be careful while choosing your vocabulary. Too many `<UNK>`s can result in a meaningless model.

We may also encounter n grams in the test set that didn't exist in the training set. Since our language model can only consider probabilities of n grams it has seen before in the training set, we need to apply smoothing while calculating the probability matrix. We change the formula a little like this -

Smoothing

- Add-one smoothing (Laplacian smoothing)

$$P(w_n | w_{n-1}) = \frac{C(w_{n-1}, w_n) + 1}{\sum_{w \in V} (C(w_{n-1}, w) + 1)} = \frac{C(w_{n-1}, w_n) + 1}{C(w_{n-1}) + V}$$

- Add-k smoothing

$$P(w_n | w_{n-1}) = \frac{C(w_{n-1}, w_n) + k}{\sum_{w \in V} (C(w_{n-1}, w) + k)} = \frac{C(w_{n-1}, w_n) + k}{C(w_{n-1}) + k * V}$$

$P(happy | I am) = \frac{C(I am happy)}{C(I am)}$ becomes $P(happy | I am) = \frac{C(I am happy) + 1}{C(I am) + V}$ where V is the total number of words in the vocabulary.

WEEK 4

Word Embeddings Using Neural Networks

Course 9

Natural Language Processing with Attention Models

Week 1

Neural Machine Translation