DEEP LEARNING SPECIALIZATION*

A PREPRINT

Kai Yi†
williamyi96@gmail.com

March 18, 2020

ABSTRACT

This paper is the notes of Deep Learning Specialization at Coursera taught by Prof. Andrew Ng. There are five courses in this specialization. They are *Neural Network and Deep Learning (NN Overview)*. *Improving Deep Neural Networks : Hyperparameter tuning, Regularization and Optimization (Optimization)*, *Structuring Machine Learning Projects (NN-Projects)*, *Convolutional Neural Networks (CNNs)* and *Sequence Model (SNNs)*, respectively.

Course *NN Overview* mainly talks about the foundations of deep learning. It includes the major technology trends driving deep learning, the building, training and applying of fully connected deep neural network, the vectorization technology improving network efficiency and the key parameters in a neural network's architecture. Course *Optimization* represents the key rules to improve neural networks' performance. It includes the industry best-practices for building deep learning applications, and a variety of tricks about hyperparameter tuning, regularization and optimization. More information about the rest three courses will conclude here later.

*This is the specialization at Coursera taught by Professor Andrew Ng.

†The author received his B.Eng with honor from Department of Software Engineering, Xi'an Jiaotong University in June 2019. His current research interests include cognition-based artificial intelligence, machine learning, computer vision and conputational psychology. His homepage is kaiyi.me. Now, he is planning to pursue PhD studies.

Contents

List of Figures

List of Tables

1 Neural Networks and Deep Learning

This is the first course of deep learning specialization at Coursera taught by Professor Andrew Ng. Here is my certificate after finishing this course (Fig. 1):

Figure 1: Certificate of Neural Networks and Deep Learning. Obtained in Feb. 2020.

1.1 Course Overview

According to the official site of this course, you will learn the foundations of deep learning. When you finish this course, you will:

- Understand the major technology trends driving deep learning.
- Be able to build, train and apply fully connected deep neural networks.
- Know how to implement efficient (vectorized) neural networks.
- Understand the key parameters in a neural network's architecture.

1.2 Introduction to Deep Learning

The section will mainly talk about the major trends driving the rise of deep learning, and understand where and how it is applied today. As I'm very familiar with this part, so I will summarize the background briefly.

Why is deep learning taking off? Data, Computation and Algorithm.

1.3 Neural Network Basics

1.3.1 Logistic Regression

Algorithm is used for classifications with 2 classes.

$y = w^T x + b$ or $y = sigmoid(w^T x + b)$ if we need y to be in between 0 and 1 (probability).

1.3.2 Logistic Regression Cost Function

We don't use square root error ($L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$) because it leads us to optimization problem which is non=convex, means it contains local optimum points. We are using cross-entropy loss like $L(\hat{y}, y) = -(y \log(\hat{y}) + (1-y) \log(1-\hat{y})$.

To explain the last function lets see:

- If $y = 1 \rightarrow L(\hat{y}, 1) = -\log(\hat{y}) \rightarrow$ we want $\hat{y}$ to be the largest $\rightarrow \hat{y}$ biggest value is 1.
- If $y = 0 \rightarrow L(\hat{y}, 0) = -\log(1 - \hat{y}) \rightarrow$ we want $1 - \hat{y}$ to be the largest $\rightarrow \hat{y}$ to be smaller as possible because it can only has 1 value.

Then the cost function will be: $J(w, b) = \frac{1}{m} \sum(\hat{y}^{(i)}, y^{(i)})$. The cost function is the average of the loss of the entire training set.

1.3.3 Gradient Descent

The actual equations we will implement:

- $w = w - \alpha dw$ (how much the function slopes in the w direction).
- $b = b - \alpha db$ (how much the function slopes in the b direction).

dw represents $\frac{dJ(w,b)}{dw}$.

1.3.4 Logistic Regression Gradient Descent

In the video, Prof. Andrew discussed the derivatives of gradient descent example for one sample with two features $x1$ and $x2$ (Figure 2).

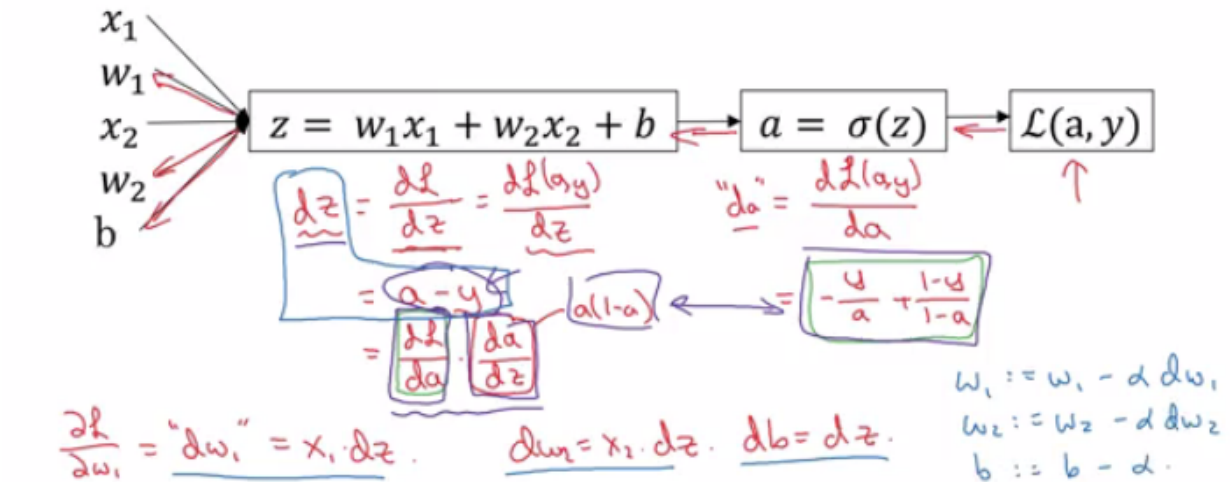


Figure 2: Logistic regression derivatives.

1.3.5 Gradient Descent on m Examples

The logistic regression pseudo code:

```
J = 0; dw1 = 0; dw2 =0; db = 0;       # Devs.
w1 = 0; w2 = 0; b=0;                                  # Weights
for i = 1 to m
          # Forward pass
          z(i) = W1*x1(i) + W2*x2(i) + b
          a(i) = Sigmoid(z(i))
          J += (Y(i)*log(a(i)) + (1-Y(i))*log(1-a(i)))

          # Backward pass
          dz(i) = a(i) - Y(i)
          dw1 += dz(i) * x1(i)
          dw2 += dz(i) * x2(i)
          db  += dz(i)
J /= m
dw1/= m
dw2/= m
db/= m

# Gradient descent
w1 = w1 - alpha * dw1
w2 = w2 - alpha * dw2
b = b - alpha * db
```

The above code should run for some iterations to minimize the error. So there will be two inner loops to implement the logistic regression.

Vectorization is so important on deep learning to reduce loops. In the last code we can make the whole loop in one step using vectorization!

1.3.6 Vectorization

Deep learning shines when the dataset are big. However for loops will make you wait a lot for a result. Thats why we need vectorization to get rid of some of our for loops.

NumPy library (dot) function is using vectorization by default. Most of the NumPy library methods are vectorized version.

The vectorization can be done on CPU or GPU thought the SIMD operation. But its faster on GPU.

Whenever possible avoid for loops.

1.3.7 Vectorizing Logistic Regression

As an input we have a matrix X and its $[n_x, m]$ and a matrix Y and its $[n_y, m]$.

Thus we have:

```
Z = np.dot(W.T,X) + b  # Vectorization, then broadcasting, Z shape is (1, m)
A = 1 / 1 + np.exp(-Z) # Vectorization, A shape is (1, m)
```

Vectorizing Logistic Regression's Gradient Output:

```
dz = A - Y                    # Vectorization, dz shape is (1, m)
dw = np.dot(X, dz.T) / m      # Vectorization, dw shape is (n_x, 1)
db = dz.sum() / m             # Vectorization, dz shape is (1, 1)
```

1.3.8 Notes on Python and Numpy

In Numpy, obj.sum(axis = 0) sums the columns while obj.sum(axis = 1) sums the rows.

In Numpy, obj.reshape(1,4) changes the shape of the matrix by broadcasting the values.

Reshape is cheap in calculations so put it everywhere you're not sure about the calculations.

Broadcasting works when you do a matrix operation with matrices that doesn't match for the operation, in this case Numpy automatically makes the shapes ready for the operation by broadcasting the values.

In general principle of broadcasting. If you have an (m,n) matrix and you add(+) or subtract(-) or multiply(*) or divide(/) with a (1,n) matrix, then this will copy it m times into an (m,n) matrix. The same with if you use those operations with a (m , 1) matrix, then this will copy it n times into (m, n) matrix. And then apply the addition, subtraction, and multiplication of division element wise.

Some tricks to eliminate all the strange bugs in the code:

- If you didn't specify the shape of a vector, it will take a shape of (m,) and the transpose operation won't work. You have to reshape it to (m, 1).
- Try to not use the rank one matrix in ANN.
- Don't hesitate to use assert(a.shape == (5,1)) to check if your matrix shape is the required one.
- If you've found a rank one matrix try to run reshape on it.

Jupyter / IPython notebooks are so useful library in python that makes it easy to integrate code and document at the same time. It runs in the browser and doesn't need an IDE to run.

To open Jupyter Notebook, open the command line and call: jupyter-notebook It should be installed to work.

To Compute the derivative of Sigmoid:

```
s = sigmoid(x)
ds = s * (1 - s)        # derivative using calculus
```

To make an image of (width,height,depth) be a vector, use this:

```
v = image.reshape(image.shape[0]*image.shape[1]*image.shape[2],1) # reshapes the image.
Gradient descent converges faster after normalization of the input matrices.
```

1.3.9 General Notes

The main steps for building a Neural Network are:

- Define the model structure (such as number of input features and outputs).
- Initialize the model's parameters.
- Loop. 1) Calculate current loss (forward propagation); 2) Calculate current gradient (backward propagation); 3) Update parameters (gradient descent).

Preprocessing the dataset is important.

Tuning the learning rate (which is an example of a "hyperparameter") can make a big difference to the algorithm.

1.4 Shallow Neural Networks

In this section, you will learn to build a neural network with one hidden layer, using forward propagation and backpropagation.

1.4.1 Neural Networks Overview

In logistic regression we have:

```
X1   \
X2    ==>   z = WX + b ==> a = Sigmoid(z) ==> L(a,Y)
X3   /
```

In neural networks with one layer we will have:

```
X1   \
X2    =>   z1 = W1X + B1 => a1 = Sigmoid(z1) => z2 = W2a1 + B2 =>
                a2 = Sigmoid(z2) => L(a2,Y)
X3   /
```

1.4.2 Neural Network Representation

We will define the neural networks that has one hidden layer.

NN contains of input layers, hidden layers, output layers.

Hidden layer means we can't see that layers in the training set.

a0 = x (the input layer)

a1 will represent the activation of the hidden neurons.

a2 will represent the output layer.

We are talking about 2 layers NN. The input layer isn't counted.

1.4.3 Computing a Neural Network's Output

Equation of hidden layers (Figure 3):

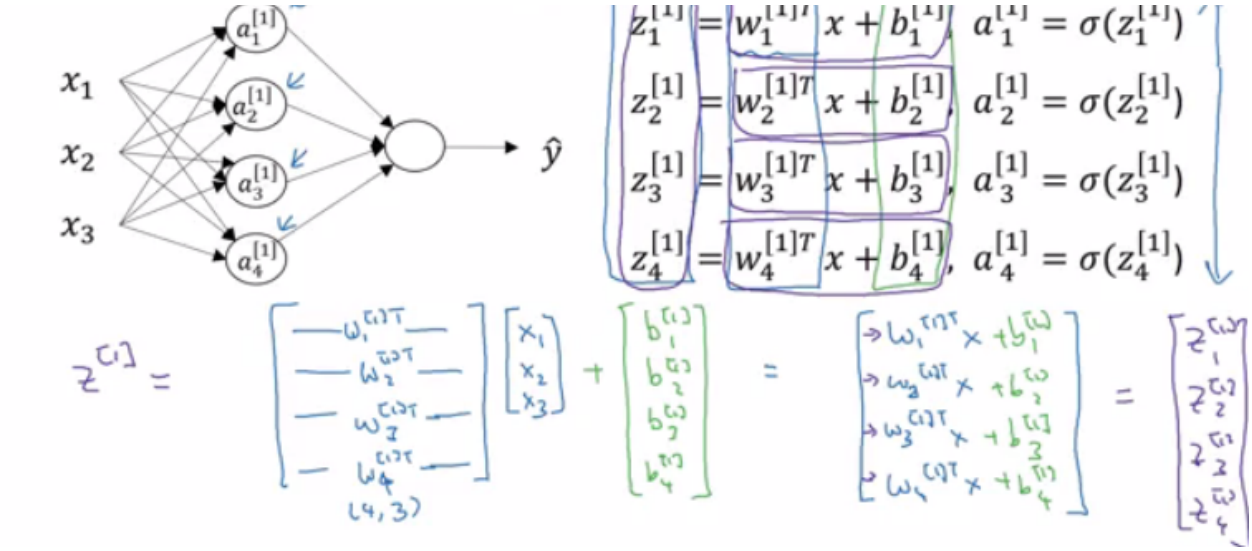


Figure 3: Neural Network Representation.

Shapes of variables:

- $w^{[1]} : (n_h, n_x), b^{[1]} : (n_h, 1), z^{[}1] \rightarrow w^{[1]}X + b : (n_h, 1), a^{[1]} \rightarrow g(z^{[1]}) : (n_h, 1).$
- $w^{[2]} : (1, n_x), b^{[2]} : (1, 1), z^{[}2] \rightarrow w^{[2]}a^{[1]} + b : (1, 1), a^{[2]} \rightarrow g(z^{[2]}) : (1, 1).$

1.4.4 Simple Neural Network

Here is the neural network with a single hidden layer (Figure 4):

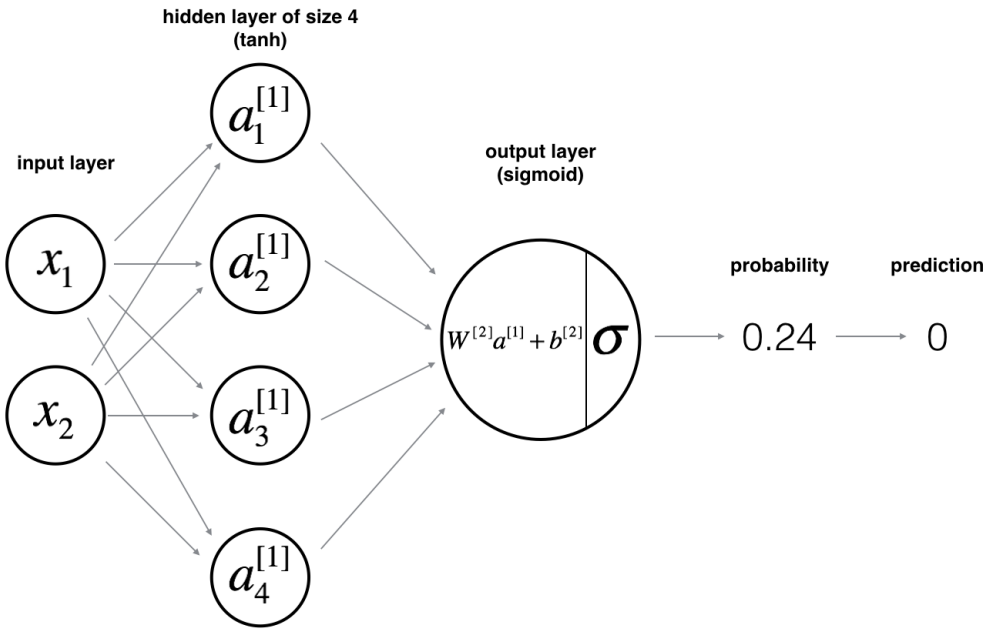


Figure 4: Network architecture with single hidden layer.

Mathematically, for one example $x^(i)$, we have:

$$\begin{aligned} z^{[1](i)} &= W^{[1]} x^{(i)} + b^{[1]} \\ a^{[1](i)} &= \tanh\left(z^{[1](i)}\right) \\ z^{[2](i)} &= W^{[2]} a^{[1](i)} + b^{[2]} \\ \hat{y}^{(i)} &= a^{[2](i)} = \sigma\left(z^{[2](i)}\right) \\ y^{(i)}_{\text{prediction}} &= \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{1}$$

Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right) + \left(1-y^{(i)}\right)\log\left(1-a^{[2](i)}\right)\right) \tag{2}$$

The general methodology to build a neural network is to:

i. Define the neural network structure (number of input units, number of hidden units, etc).
ii. Initialize the model's parameters.
iii. Loop: 1) Implement forward propagation; 2) Compute loss; 3) Implement backward propagation to get the gradients; 4) Update parameters (gradient descent).

1.4.5 Gradient of NN with One Hidden Unit

Here is the updating rules for the neural network with one hidden unit (Figure 5):

1.5 Deeper Neural Networks

As the pipeline for deeper neural network is very similar to shallow neural network, so I will summarize this part briefly.

$dz^{[2]} = a^{[2]} - y$	$dZ^{[2]} = A^{[2]} - Y$
$dW^{[2]} = dz^{[2]} a^{[1]^T}$	$dW^{[2]} = \frac{1}{m} dZ^{[2]} A^{[1]^T}$
$db^{[2]} = dz^{[2]}$	$db^{[2]} = \frac{1}{m} np.sum(dZ^{[2]}, axis = 1, keepdims = True)$
$dz^{[1]} = W^{[2]T} dz^{[2]} * g^{[1]\prime}(\mathrm{z}^{[1]})$	$dZ^{[1]} = W^{[2]T} dZ^{[2]} * g^{[1]\prime}(\mathrm{Z}^{[1]})$
$dW^{[1]} = dz^{[1]} x^T$	$dW^{[1]} = \frac{1}{m} dZ^{[1]} X^T$
$db^{[1]} = dz^{[1]}$	$db^{[1]} = \frac{1}{m} np.sum(dZ^{[1]}, axis = 1, keepdims = True)$

Andrew Ng

Figure 5: Summary of gradient descent.

1.5.1 Building Your Deeper Neural Network: Step by Step

The pipeline of parameters updating (Figure 6):

Forward Propagation The linear forward module (vectorized over all the examples) computes the following equations:

$$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]} \tag{3}$$

where $A^{[0]} = X$.

Cost Function

$$-\frac{1}{m} \sum_{i=1}^{m} \left(y^{(i)} \log \left(a^{[L](i)} \right) + \left(1 - y^{(i)} \right) \log \left(1 - a^{[L](i)} \right) \right) \tag{4}$$

Illustrations of forward and backward propagation Could refer to Figure 7.

Backward Gradients Here are the gradients for layer l:

$$\begin{aligned} dW^{[l]} &= \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \\ db^{[l]} &= \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \\ dA^{[l-1]} &= \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \end{aligned} \tag{5}$$

Update Parameters :

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]} b^{[l]} = b^{[l]} - \alpha db^{[l]} \tag{6}$$

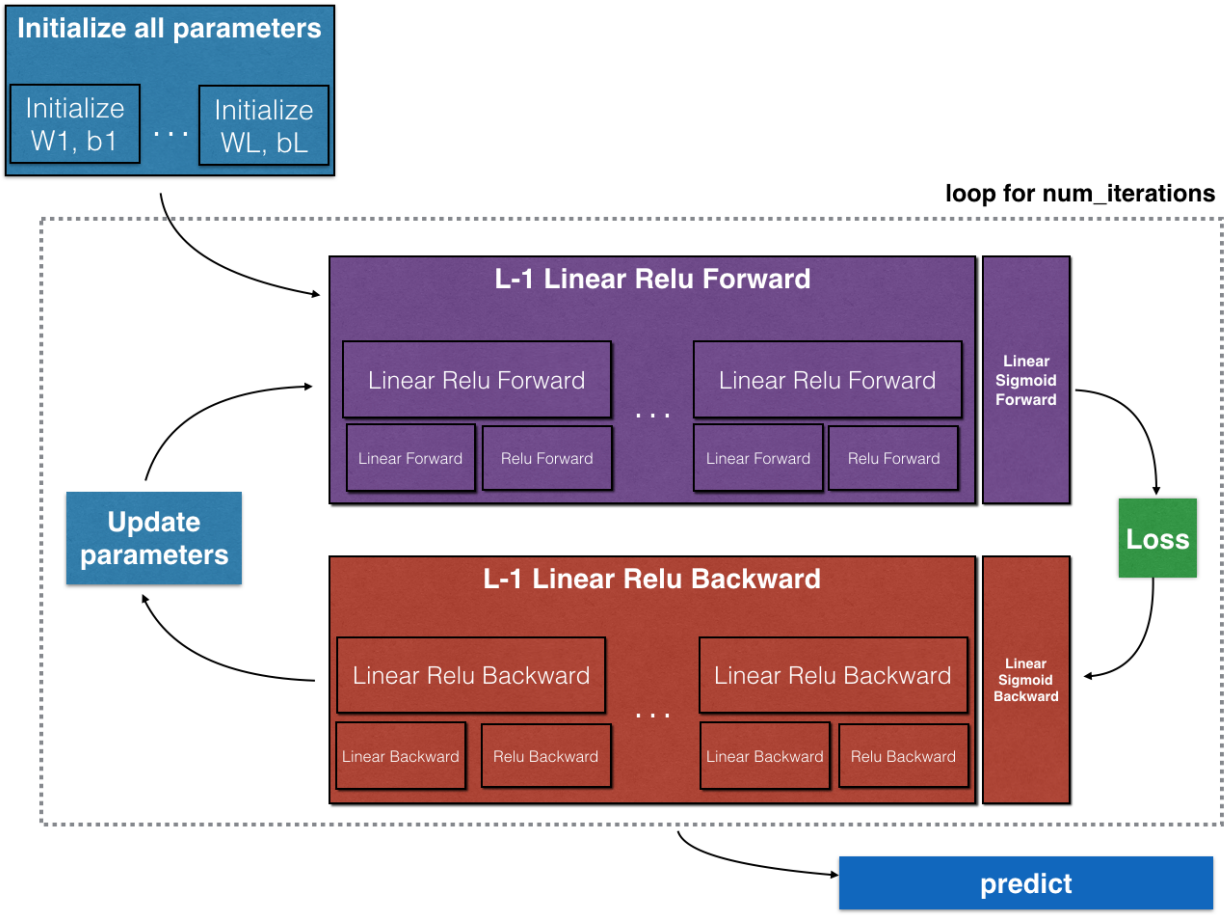


Figure 6: The pipeline of parameters updating.

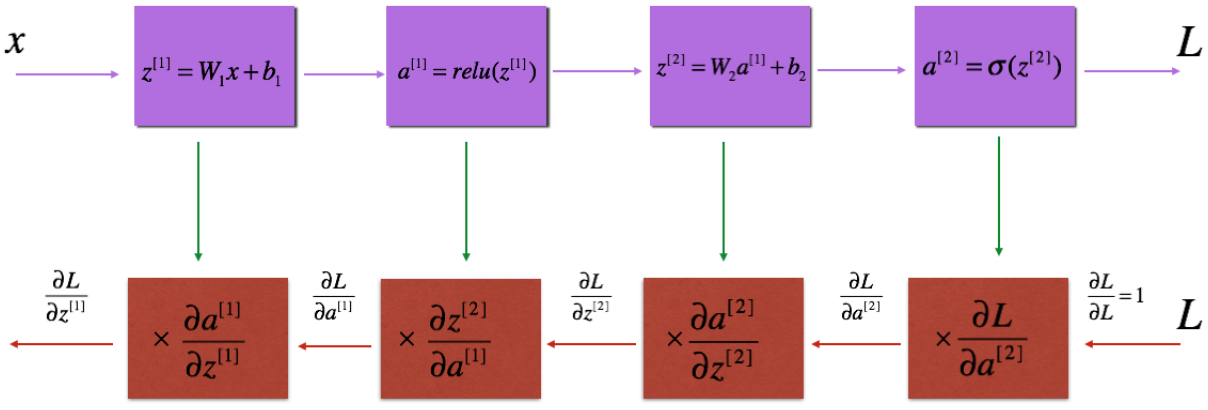


Figure 7: Forward and Backward propagation for LINEAR->RELU->LINEAR->SIGMOID. The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

2 Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization

This is the second courseof deep learning specialization at Coursera taught by Professor Andrew Ng. Here is my certificate after finishing this course (Fig. 8):

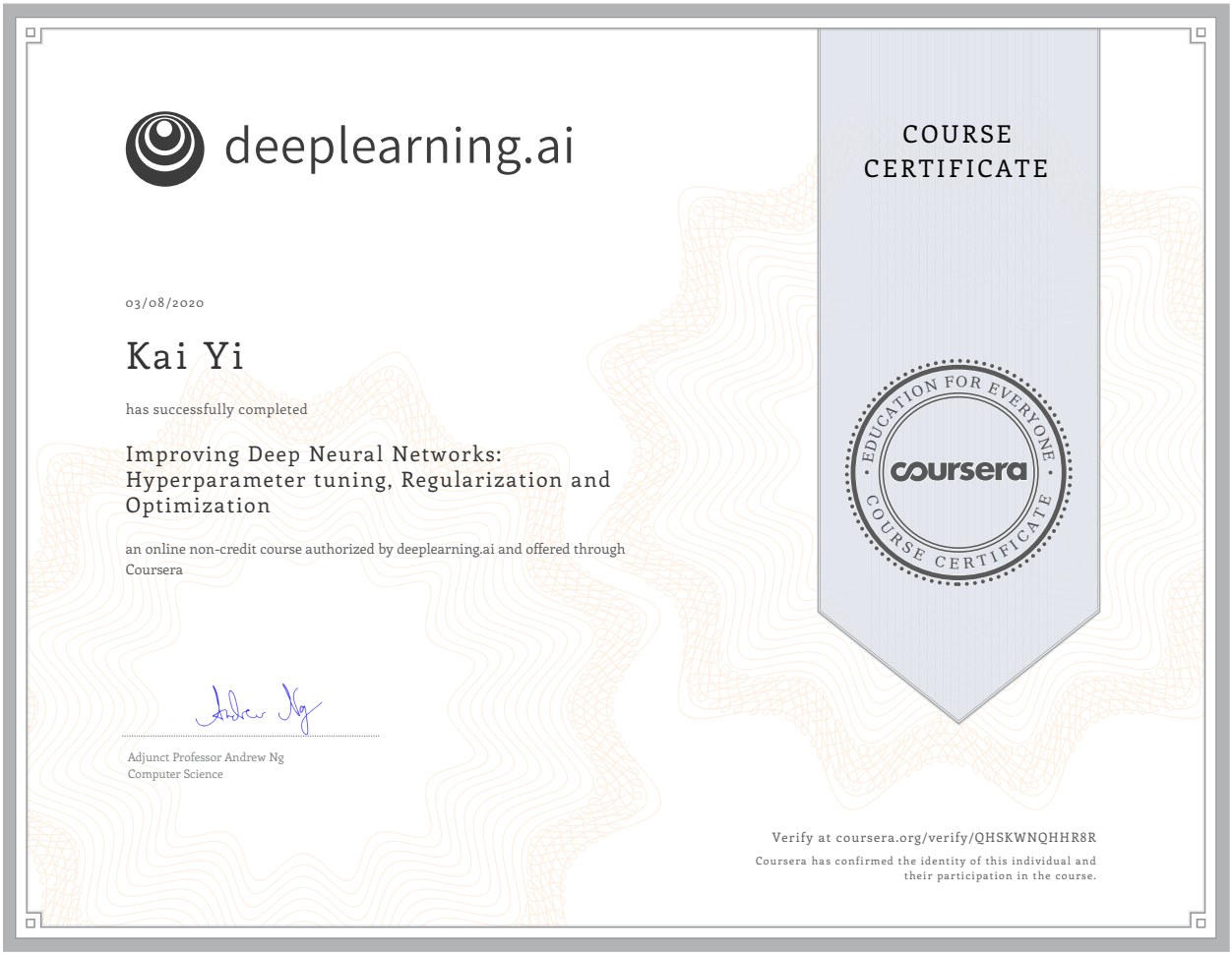


Figure 8: Certificate of Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization. Obtained in March 2020.

2.1 Course Overview

According to the official site of this course, there are following key components:

- Understand industry best-practices for building deep learning applications.
- Be able to effectively use the common neural network 'tricks', including initialization, L2 and dropout regularization, batch normalization and gradient checking.
- Be able to implement and apply a variety of optimization algorithms, such as mini-batch gradient descent, momentum, RMSprop and Adam, and check for their convergence.
- Understand new best-practices for the deep learning era of how to set up train/dev/test sets and analyze bias/variance.
- Be able to implement a neural network in TensorFlow.

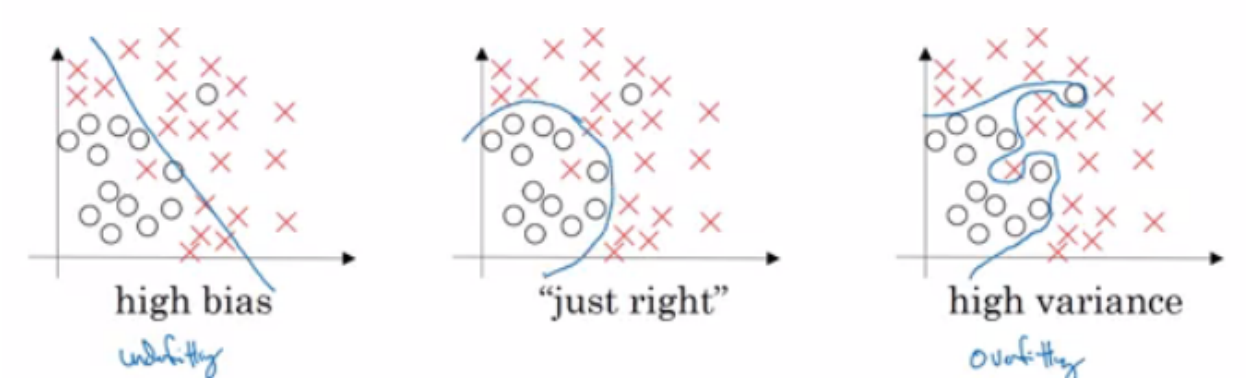


Figure 9: Bias and Variance.

2.2 Practical Aspects of Deep Learning

2.2.1 Train/Dev/Test Sets

Pipeline of deep learning practice: Idea >> Code >> Experiment >> Idea. We get initial idea first, then implement the rough idea. Next fine tuning our model based on experiment. Then we get new thoughts.

Data will be split into three parts:

- Training Set. (the largest set)
- Hold-Out Cross Validation Set / Development Set.
- Test Set.

We will try to build a model upon training set then try to optimize hyperparameters on dev set as much as possible. Then after our model is ready we try and evaluate the testing set.

Thus the trend on the ratio of splitting the models:

- If size of the dataset is 100 to 1,000,000, then => 60/20/20.
- If size of the dataset is more than 1,000,000, then => 98/1/1 or 99.5/0.25/0.25.

Make sure the dev and test set are coming from the same distribution.

- E.g. The cat training pictures from the web can be put into training set to get more data. However, the picture from users cell phone and the web have different distribution. Therefore, the web picture can not be used separately on dev set or test set. Besides, it's better not to use a little web picture on dev/test set (it will cause mismatch problem). Besides,

It's OK to only have a dev set without a testing set. But a lot of people in this case call the dev set as the test set. A better terminology is to call it a dev set as its used in the development.

2.2.2 Bias / Variance

Bias / Variance techniques are easy to learning, but difficult to master.

- Underfitting => High Bias
- Overfitting => High Variance

The following visualization (Fig. 9):

If Bayesian error (optimal error) is 0%, next is several cases:

- High variance (overfitting): Train error: 1%, Dev error: 15%
- High bias (underfitting): Train error: 10%, Dev error: 8%

- High bias & high variance: Train error: 16%, Dev error: 30%
- Quite well: Train error: 0.6%, Dev error: 0.9%

2.2.3 Basic Recipe for Machine Learning

If my algorithm has a high bias:

- Try to make my NN bigger (size of hidden units, number of layers).
- Try to train it longer.
- (Not unique) Try a different model that is suitable for my data / Try different (advanced) optimization algorithms.

If my algorithm has a high variance:

- Get more data.
- Try regularization.
- (Not unique) Try a different model that is suitable for my data / Try different (advanced) optimization algorithms.

I should try the previous two points until I have a low bias and low variance.

In the older days before deep learning, there was a 'Bias/Variance Tradeoff'. But nowadays we can reduce bias and variance separately by using more advanced tools and techniques.

2.2.4 Regularization

Adding regularization to NN will help it reduce variance (overfitting).

- L1 matrix norm: $||W|| = \sum w_{ij}$.
- L2 matrix norm: $||W||^2 = \sum w_{ij}^2$. Also $||W||^2 = W^T * W$ if W is a vector.

Regularization for logistic regression: (Find more accurate equations for regularizations)

- The normal cost function that we want to minimize is: $J(W, b) = \frac{1}{m} \sum L(y_i, \hat{y}_i)$
- The L2 regularization version: $J(W, b) = \frac{1}{m} \sum L(y_i, \hat{y}_i) + \frac{\lambda}{2m} \sum w_i^2$
- The L1 regularization version: $J(W, b) = \frac{1}{m} \sum L(y_i, \hat{y}_i + \frac{\lambda}{m} \sum w_i$
- L2 regularization is being used much more often and λ here is the regularization parameter (hyperparameter)

(Some points I can not understand very clearly here.)

2.2.5 Why regularization reduces overfitting?

Here are som intuitions:

Intuition 1:

- If λ is too large - a lot of w's will be close to zeros which will make the NN simpler (I can think of it as it would behave closer to logistic regression).
- If λ is good enough it will just reduce some weights that makes the neural network overfit.

Intuition2 (with tanh activation function):

- If λ is too large, w's will be small (close to zero) - will use the linear part of the tanh activation function, so we will go from non linear activation to roughly linear which would make the NN a roughly linear classifier.
- If λ is good enought, it will just make some of tanh activations roughly linear which will prevent overfitting.

Implementation tip: if you implement gradient descent, one of the steps to debug gradient descent is to plot the cost function J as a function of the number of iterations of gradient descent and you want to see that the cost function J decreases monotonically after every elevation of gradient descent with regularization. If you plot the old definition of J (no regularization) then you might not see it decrease monotonically.

2.2.6 Dropout Regularization

The dropout regularization eliminates some neurons/weights on each iteration based on a probability. A most common technique to implement dropout is called "inverted dropout".

Code for inverted dropout:

```
keep_prob = 0.8 # 0 <= keep_prob <= 1, this code is only for layer 3

# the generated number that are greater than keep_prob will be
# dropped. Thus 80% stay, 20% dropped.
# a3.shape = (num_neurons l3, m examples)
d3 = np.random.rand(a3.shape[0], a[1].shape[1]) < keep_prob

a3 = np.multiply(a3, d3) # only keep the values in d3 (True)

a3 = a3 / keep_prob # re-scaling a3, inverted dropout.
```

Matrix d3 is used for forward and back propagation and is the same for them, but it is different for each iteration due to the random initialization.

At test time we don't use dropout. If I implement dropout at test time - it would add noise to predictions.

2.2.7 Understanding Dropout

Intuition 1:

- The intuition was that dropout randomly knocks out units in my network. So it's as if on every iteration I'm working with a smaller NN, and so using a smaller NN seems like it should have a regularizing effect.

Intuition 2:

- NN can't rely on any one feature, so it has to spread out weights.

Dropout can have different keep_prob per layer. And the input layer dropout has to be near 1 because I don't want to eliminate a lot of features.

A lot of researchers are using dropout in the area of computer vision because they have a very big input size and almost never have enough data, so overfitting is the usual problem. And dropout is a regularization technique to prevent overfitting.

A downside of dropout is that the cost function J is not well defined and it will be hard to debug (plot J by iteration).

- To solve that I'll need to turn off dropout, set all the keep_prob s to 1, and then run the code and check that it monotonically decreases J and then turn on the dropouts again.

2.2.8 Other regularization methods

Data Augmentation For example in a computer vision task:

- Flip all my pictures horizontally which will give my more data instances
- Apply a random position and rotation to an image to get more data.

New data obtained using this technique isn't as good as the real independent data, but still can be used as a regularization technique.

Early Stopping In this technique we plot the training set and the dev set cost together for each iteration. At some iteration the dev set cost will top decreasing and will start increasing.

We will pick the point at which the training set error and dev set error are best (great trade-off between dev error and train error). Thus we can take these parameters as the best parameters (Visualized at Fig. 10).

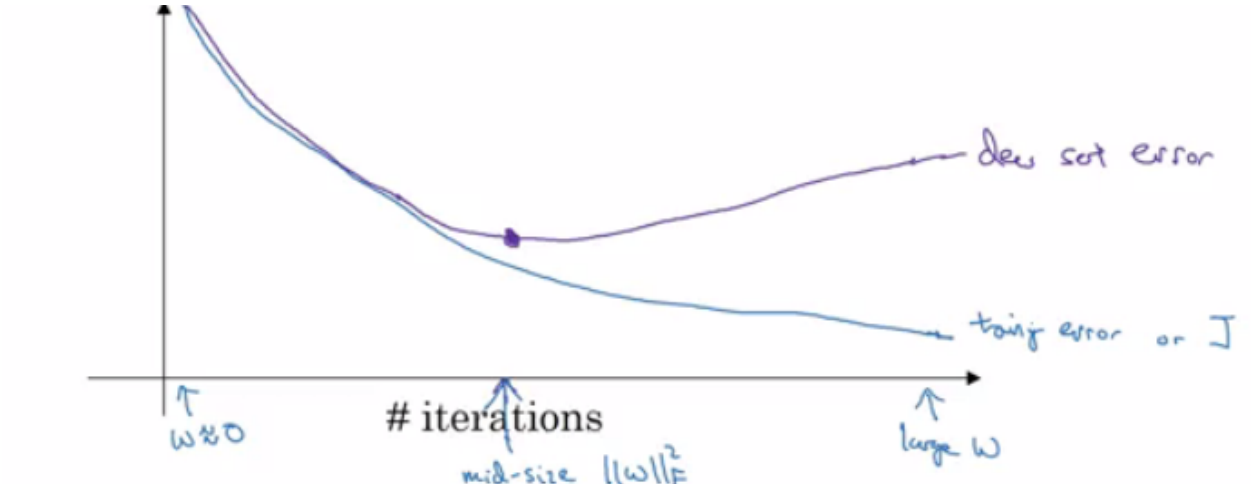


Figure 10: Early stopping method preventing overfitting.

Professor Andrew prefers to use L2 regularization instead of early stopping because this technique simultaneously tries to minimize the cost function and not to overfit which contradicts the orthogonalization approach (we should not make decision on two variables).

However, the advantage of early stopping is that we don't need to search a hyperparameter like in other regularization approaches (like λ in L2 regularization).

Model Ensembles Basic idea: Train multiple independent models and at test time average their results.

It can get 2% improved performance and can be used in competitions instead of practical deployments. It reduces the generalization error.

2.2.9 Normalizing Inputs

If I normalize my inputs this will speed up the training process a lot (convert optimizing from ellipse to circular).

Normalization are going on these steps:

- Get the mean of the training set: mean = (1/m) * sum(x(i))
- Subtract the mean from each input: X = X - mean. This makes my inputs centered around 0.
- Get the variance of the training set: variance = (1/m) * sum((x(i)-mean)**2)
- Normalize the variance: X /= variance

These steps should be applied to training, dev, and testing sets (but using mean and variance of the train set).

Why normalize?

- If we don't normalize the inputs our cost function will be deep and its shape will be inconsistent (elongated) then optimizing it will take a long time.
- But if we normalize it the opposite will occur. The shape of the cost function will be consistent (look more symmetric like circle in 2D example) and we can use a larger learning rate α - the optimization will be faster.

2.2.10 Vanishing / Exploding gradients

The vanishing / exploding gradients occurs when our derivatives become very small or very big.

To understand the problem, suppose that we have a deep neural network with number of layer L, and all the activation functions are linear and each b=0.

Then $\hat{Y} = W[L]W[L-1] \cdot W[1]X$. If we have 2 hidden units per layer and x1=x2=1, if W[i,j] is greater than 1, then $\hat{Y}$ will be exponentially large. Meanwhile, if W[i,j] is smaller than 1, then $\hat{Y}$ will be exponentially small.

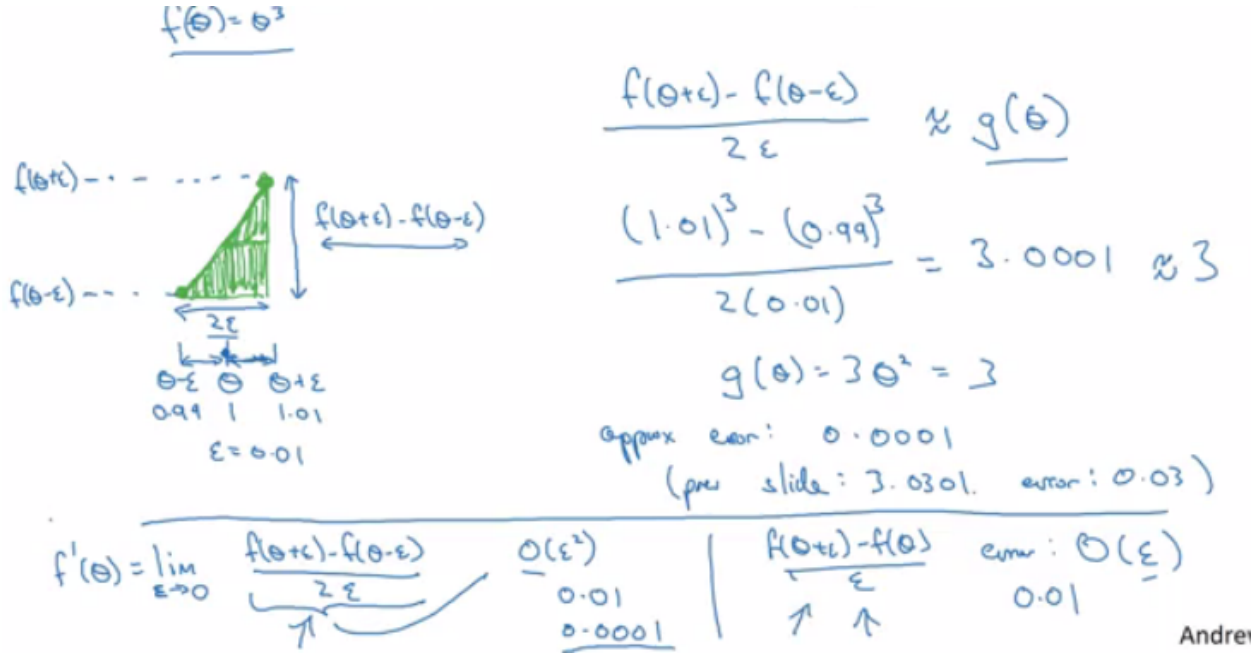


Figure 11: Checking the derivative computation.

2.2.11 Weight Initialization for Deep Networks

A partial solutions to the vanishing / exploding gradients in NN is better or more careful choice of the random initialization of weights.

A well chosen initialization can:

- Speed up the convergence of gradient descent
- Increase the odds of gradient descent converging to a lower training (and generalization) error.

- Zero Initialization: *np.zeros(shape)*
- Random Initialization: *np.random.randn(shape* * 0.01
- Bengio et al. Initialization: *np.random.randn(shape) * np.sqrt(2/(n[l-1] + n[l])*
- He et al. Initialization (Xavier Initialization): *np.random.randn(shape) + np.sqrt(2/n[l-1])* (Set ReLu as activation is better)

2.2.12 Numerical approximation of gradients

There is a technique called gradient checking which tells you if your implementation of backpropagation is correct.

There is a numerical way to calculate the derivative (Fig. 11)

Gradient checking is used only for debugging.

Main goal is to check whether the following satisfied:

$$\frac{\partial J}{\partial \theta} = lim_{\epsilon \to 0} \frac{J(\theta + \epsilon) - J(\theta - \epsilon}{2\epsilon} \tag{7}$$

2.2.13 Gradient checking implementation notes

Don't use the gradient checking algorithm at training time because it's very slow. Use gradient checking only for debugging. If algorithm fails grad check, look at components to try to identify the bug.

Don't forget to add $\frac{\lambda}{2m} \sum w_i$ to J if we're using L1 or L2 regularization.

Gradient checking doesn't work with dropout because J is not consistent. We must first turn off dropout, run gradient checking and then turn on dropout again.

Run gradient checking at random initialization and train the network for a while maybe there's a bug which can be seen when w's and b's become larger (further from 0) and can't be seen on the first iteration (when w's and b's are very small).

2.2.14 Initialization Summary

The weights should be initialized randomly to break symmetry.

It's OK to initialize the biases to zeros. Symmetry still can be broken so long as weights are initialized randomly.

Different initializations lead to different results.

Random initialization is used to break symmetry and make sure different hidden units can learn different things.

Don't initialize to values that are to large.

He initialization (Xavier Initialization) works well for networks with ReLU activations.

2.2.15 Regularization Summary

L2 Regularization The value of λ is a hyperparameter that you can tune using a dev set. L2 regularization makes the decision boundary smoother. If λ is too large, it's also possible to 'oversmooth', resulting in a model with high bias.

What is L2-regularization actually doing? L2 regularization relies on the assumption that a model with large weights is simpler than one with small weights. Thus, by penalizing the square value of the weights in the cost function, you drive all the weights to smaller values. It becomes too costly for the cost to have large weights. This leads to a smoother model in which the output changes more slowly as the input changes.

Dropout Dropout is a regularization technique. You only use dropout during training. Don't use it (randomly eliminates nodes) during test time.

Apply dropout both during forward and backward propagation.

During training time, divide each dropout layer by keep_prob to keep the same expected value for the activations. For example, if keep_prob is 0.5, then we will on average shut down half the nodes, so the output will be scaled by 0.5 since only the remaining half are contributing to the solution. Dividing by 0.5 is equivalent to multiplying by 2. Hence, the output now has the same expected value. You can check that this works even when keep_prob is other values than 0.5.

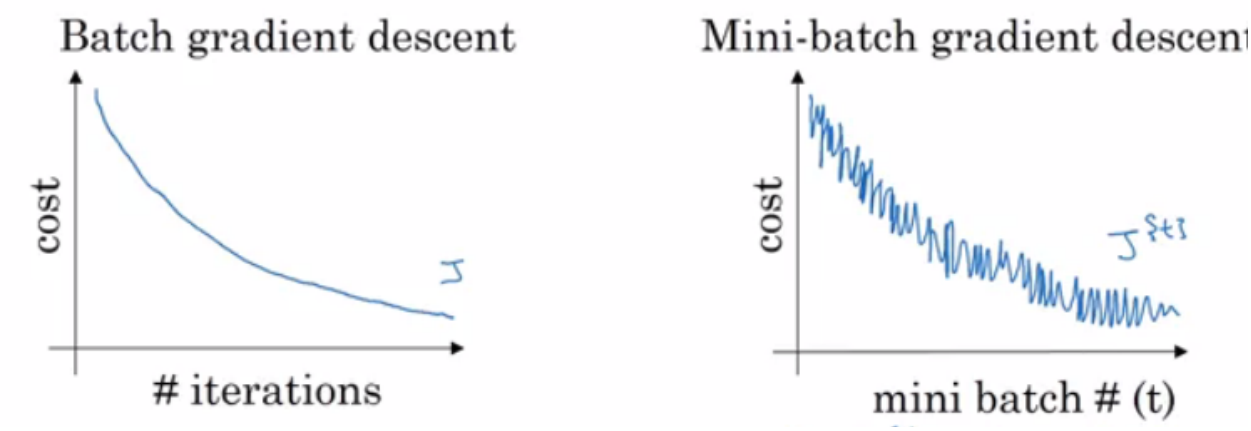


Figure 12: Training with mini batch gradient descent

2.3 Optimization Algorithms

2.3.1 Mini-Batch Gradient Descent

Training NN with large data is slow. So to find an optimization algorithm that runs faster is a good idea.

Suppose we have m = 50 million. To train this data it will take a huge processing time for one step. Because 50 million won't fit in the memory at once we need other processing to make such a thing.

In Batch Gradient Descent we run the gradient descent on the whole dataset while in Mini-Batch Gradient Descent we run the gradient descent on the mini datasets.

```
for t = 1:No_of_batches  # for every epoch
        AL, caches = forward_prop(X{t}, Y{t})
        cost = compute_cost(AL, Y{t})
        grads = backward_prop(AL, caches)
        update_parameters(grads)
```

The code inside an epoch should be vectorized. Mini-batch gradient descent works much faster in the large datasets.

2.3.2 Understanding mini-batch gradient descent

In mini-batch algorithm, the cost won't go down with each step as it does in batch algorithm. It could contain some ups and downs but generally it has to go down (unlike the batch gradient descent where cost function decreases on each iterations (Fig. 12).

Mini-batch size:

- (mini_batch_size = m) => batch gradient descent
- (mini_batch_size = 1) => stochastic gradient descent (SGD)
- (mini_batch_size = between 1 and m) => ordinary mini-batch gradient descent

Batch gradient descent:

- too long per iteration (epoch)

Stochastic gradient descent:

- too noisy regarding cost minimization (can be reduced by using smaller learning rate)
- won't ever converge (reach the minimum cost)
- lose speedup from vectorization

Mini-batch gradient descent:

- faster learning (you have the vectorization advantage and make progress without waiting to process the entire training set)
- doesn't always exactly converge (oscelates in a very small region, but you can reduce learning rate)

Guidelines for choosing mini-batch size:

- If small training set (<2000 examples) - use batch gradient descent.
- It has to be a power of 2 (because of the way computer memory is layed out and accessed, somtimes your code runs faster if your mini-batch size is a power of 2): 64, 128, 256, 512, 1024, ...
- Make sure that mini-batch fits in CPU/GPU memory.

Mini-batch size is a hyperparameter.

2.3.3 Exponentially weighted averages

There are optimization algorithms that are better than gradient descent, but you should first learn about Exponentially Weighted Averages.

If we have data like the temperature of day through the year it could be like this:

$$\begin{aligned} \theta(1) &= 40 \ \ (t(1) = 40) \\ \theta(2) &= 49 \\ \theta(3) &= 45 \\ &... \\ \theta(180) &= 60 \\ &... \end{aligned} \tag{8}$$

This data is small in winter and big in summer. If we plot this data we will find it some noisy.

Now lets computer the exponentially weighted averages:

$$\begin{aligned} v0 &= 0 \\ v1 &= 0.9 * v0 + 0.1 * \theta(1) = 4 \\ v2 &= 0.9 * v1 + 0.1 * \theta(2) = 8.5 \\ v3 &= 0.9 * v2 + 0.1 * \theta(3) = 12.15 \\ &... \end{aligned} \tag{9}$$

General equation:

$$v(t) = \beta v(t-1) + (1-\beta)\theta(t) \tag{10}$$

If we plot Equation 10 it will represent averages over $\frac{1}{1-\beta}$ entries:

- $\beta = 0.9$ will average last 10 entries
- $\beta = 0.98$ will average last 50 entries
- $\beta = 0.5$ will average last 2 entries

Best β average for our case is between 0.9 and 0.98.

Intuition: The reason why exponentially weighted averages are useful for further optimizing gradient descent algorithms is that it can give different weights to recent data points (θ) based on value of β. If β is high (around 0.9), it smoothens out the averages of skewed data points (oscillations w.r.t gradient descent terminology). So this reduces oscillations in gradient descent and hence makes faster and smoother path towards minimal (Fig 13).

Figure 13: Visualization of exponentially weighted averages.

2.3.4 Bias correction in exponentially weighted averages

The bias correction helps make the exponentially weighted averages more accurate. Because $v(0) = 0$, the bias of the weighted averages is shifted and the accuracy suffers at the start.

To solve the bias issue we have to use this equation:

$$v(t) = \frac{\beta v(t-1) + (1-\beta)\theta(t)}{(1-\beta^t)} \tag{11}$$

As t becomes larger, $1 - \beta^t$ becomes close to 1.

2.3.5 Gradient Descent With Momentum

The momentum algorithm almost always works faster than standard gradient descent.

The simple idea is to calculate the exponentially weighted average for your gradients and then update your weights with the new values.

Here is the pseudo code:

```
vdW = 0, vdb = 0
on iteration t:
    # can be mini-batch or batch gradient descent
    # to compute dW, db on current batch

    vdW = beta * vdW + (1 - beta) * dW
    vdb = beta * vdb + (1 - beta) * db
    W = W - learning_rate * vdW
    b = b - learning_rate * vdb
```

Momentum helps the cost function to go to the minimum point in a more fast and consistent way.

β is a hyperparameter . $\beta = 0.9$ is very common and works very well in most cases. In practice, people don't bother implementing bias correction.

2.3.6 RMSprop (Root Mean Square prop)

Here is the pseudo code:

```
sdW = 0, sdb = 0
sdW = 0, sdb = 0
on iteration t:
    # can be mini-batch or batch gradient descent
    # compute dw, db on current mini-batch

    sdW = (beta * sdW) + (1 - beta) * dW^2  # squaring is element-wise
    sdb = (beta * sdb) + (1 - beta) * db^2  # squaring is element-wise
    W = W - learning_rate * dW / sqrt(sdW)
    b = B - learning_rate * db / sqrt(sdb)
```

RMSprop will make the cost function move slower on the vertical direction and faster on the horizontal direction in the following example (Fig. 14):

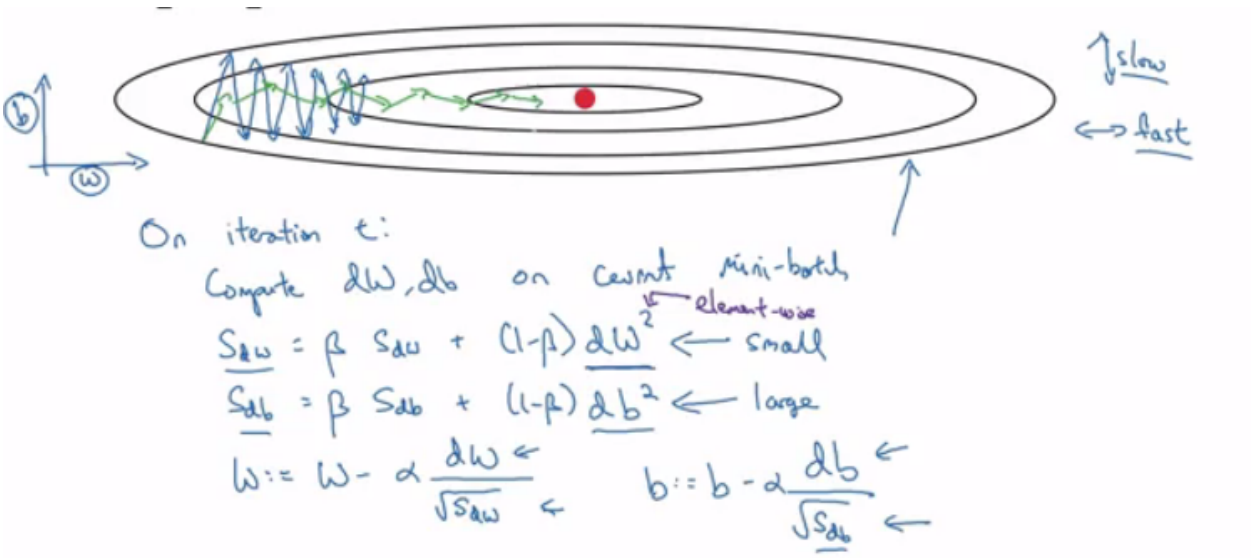


Figure 14: Intuition of RMSprop.

Ensure that sdW is not zero by adding a small value ϵ (e.g. $\epsilon = 10^{-8}$) to it (α represents the learning rate):

$$W = W - \alpha * \frac{dW}{\sqrt{sdW + \epsilon}} \tag{12}$$

With RMSprop you can increase your learning rate. Its developed by Geoffrey Hinton and firstly introduced on Coursera.org course (Neural Networks and Machine Learning).

2.3.7 Adam Optimization Algorithm

Stands for Adaptive Moment Estimation.

Adam optimization and RMSprop are among the optimization algorithms that worked very well with a lot of NN architectures.

Adam optimization simply puts RMSprop and momentum together.

Here is the pseudo code:

```
vdW = 0, vdW = 0
sdW = 0, sdb = 0
on iteration t:
        # can be mini-batch or batch gradient descent
        # to compute dw, db on current mini-batch

        vdW = (beta1 * vdW) + (1 - beta1) * dW     # momentum
        vdb = (beta1 * vdb) + (1 - beta1) * db     # momentum

        sdW = (beta2 * sdW) + (1 - beta2) * dW^2   # RMSprop
        sdb = (beta2 * sdb) + (1 - beta2) * db^2   # RMSprop

        vdW = vdW / (1 - beta1^t)      # fixing bias
        vdb = vdb / (1 - beta1^t)      # fixing bias

        sdW = sdW / (1 - beta2^t)      # fixing bias
        sdb = sdb / (1 - beta2^t)      # fixing bias

        W = W - learning_rate * vdW / (sqrt(sdW) + epsilon)
        b = b - learning_rate * vdb / (sqrt(sdb) + epsilon)
```

Here is the equation form:

$$
\begin{aligned}
vdW &= (\beta_1 * vdW) + (1 - \beta_1) * dW \\
vdb &= (\beta_1 * vdb) + (1 - \beta_1) * db \\
sdW &= (\beta_2 * sdW) + (1 - \beta_2) * dW^2 \\
sdb &= (\beta_2 * sdb) + (1 - \beta_2) * db^2 \\
vdW &= vdW/(1 - \beta_1^t) \\
vdb &= vdb/(1 - \beta_1^t) \\
sdW &= sdW/(1 - \beta_2^t) \\
sdb &= sdb/(1 - \beta_2^t) \\
W &= W - \alpha * \frac{vdW}{\sqrt{(sdW)} + \epsilon} \\
b &= b - \alpha * \frac{vdb}{\sqrt{(sdW)} + \epsilon}
\end{aligned}
\tag{13}
$$

Hyperparameters for Adam:

- Learning rate: needed to be tuned.
- β_1: parameter of the momentum - 0.9 is recommended by default.
- β_2: parameter of the RMSprop - 0.999 is recommended by default.
- ϵ: 10^-8 is recommended by default.

2.3.8 Learning Rate Decay

Slowly reduce learning rate.

As mentioned before mini-batch gradient descent won't reach the optimum point (converge). But by making the learning rate decay with iterations it will be much closer to it because the steps (and possible oscillations) near the optimum are smaller.

Several popular decay rate decay (r represents the decay rate and i represents the current number of epoch):

$$\begin{aligned} \alpha &= \frac{1}{1 + r * i} * \alpha_0 \\ \alpha &= 0.95^i * \alpha_0 \\ \alpha &= \frac{k}{\sqrt{i}} * \alpha_0 \end{aligned} \tag{14}$$

Besides, some people perform learning rate decay discretely - repeatedly decrease after some number of epochs. While also some people are making changes to the learning rate manually.

decay_rate r is another hyperparameter which has less priority.

2.3.9 The Problem of Local Optima

The normal local optima is not likely to appear in a deep neural network because data is usually high dimensional. For point to be a local optima it has to be a local optima for each of the dimensions which is highly unlikely.

It's unlikely to get stuck in a bad local optima in high dimensions, it is much more likely to get to the saddle point rather to the local optima, which is not a problem.

Plateaus can make learning slow:

- Plateau is a region where the derivative is close to zero for a long time.
- This is where algorithms like momentum, RMSprop or Adam can help.

2.4 Hyperparameter Tuning, Batch Normalization and Programming Frameworks

2.4.1 Tuning Process

We need to tune our hyperparameters to get the best out of them.

Hyperparameters importance are (as for Andrew Ng, importance doesn't decrease strictly):

i. Learning rate
ii. Momentum beta
iii. Mini-batch size
iv. No. of hidden units
v. No. of layers
vi. Learning rate decay
vii. Regularization lambda
viii. Activation functions
ix. Adam β_1, β_2 and ϵ

It's hard to decide which hyperparameter is the most important in a problem. It depends a lot on your problem.

One of the ways to tune is to sample a grid with N hyperparameters settings and then try all settings combinations on your problem.

However, try random values instead of using a grid.

You can use Coarse to fine sampling scheme:

- When you find some hyperparameters values that give you a better performance - zoom into a smaller region around these values and sample more densely within this space.

2.4.2 Using an Appropriate Scale to Pick Hyperparameters

Let's say you have a specific range for a hyperparameter from a to b. It's better to search for the right ones using the logarithmic scale rather than in linear scale:

- Calculate: a_log = log(a), e.g. a = 0.0001 then a_log = -4
- Calculate: b_log = log(b), e.g. b = 1 then b_log = 0
- Then: r = (a_log - b_log), result = 10r̂

It uniformly samples values in log scale from [a, b].

If you want to use the last method on exploring on the "momentum beta β":

i. β best range is from 0.9 to 0.999.

ii. You should search for $1 - \beta$ in range 0.001 to 0.1 and then replace a with 0.001 and replace b with 0.1.

2.4.3 Hyperparameters Tuning in Practice: Pandas vs. Caviar

Intuitions about hyperparameter settings from one application area may or may not transfer to a different one.

If you don't have much computational resources you can use "babysitting model":

- Day 0 you might initialize your parameter as random and then start training.
- Then you watch your learning curve gradually decrease over the day.
- And each day you nudge your parameters a little during training.
- Called panda approach.

If you have enough computational resources, you can run some models in parallel and at the end of the day(s) you check the results.

- Called Caviar approach.

2.4.4 Normalizing Activations in a Network

Batch Normalization can speed up learning.

Before we normalized input by subtracting the mean and dividing by variance. This helped a lot for the shape of the cost function and for reaching the minimum point faster.

The question is: for any hidden layer can we normalize A[l] to train W[l+1], b[l+1] faster? This is what batch normalization is about.

There are some debates in the deep learning literature about whether you should normalize values before the activation function Z[l] or after applying the activation function A[l]. In practice, normalizing Z[l] is done much more often and that is what Andrew Ng presents.

Algorithm:

```
1. Given Z[l] = [z(1), ..., z(m)], i = 1 to m (for each input)
2. Compute mean = 1/m * sum(z(i))
3. Compute variance = 1/m * sum(z(i) - mean)^2
4. Then Z_norm[l] = (z[l] - mean) / np.sqrt(variance + epsilon)
     Forcing the inputs to a distribution with zero mean and 1 variance
5. Then Z_tilde[l] = gamma * Z_norm[l] + beta
     To make inputs belong to other distribution;
     Gamma and beta are learnable parameters of the model;
     Making the NN learn the distribution of the outputs.
```

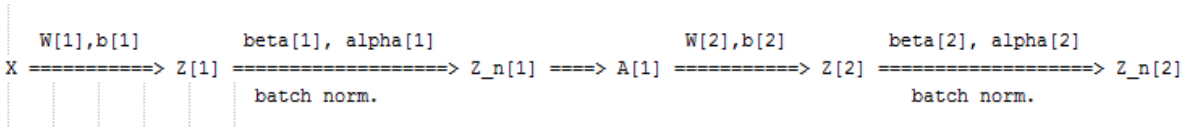


Figure 15: Workflow of 2 layers NN with batch normalization.

2.4.5 Fitting Batch Normalization into a Neural Network

Using batch norm in 2 layers NN:

$\beta[i]$ and $\gamma[i]$ are updated using any optimization algorithms (like GD, Momentum, RMSprop, Adam).

Batch normalization is usually applied with mini-batches.

Shapes: $Z[l] - (n[l], m), \beta[l] - (n[l], m), \gamma[l] - (n[l], m)$.

2.4.6 Why Does Batch Normalization Work?

The first reason is the same reason as why we normalize X (Shape the inputs and makes loss reach minimum point faster).

The second reason is that batch normalization reduces the problem of input values changing (shifting).

2.4.7 Batch Normalization at Test Time

When we train a NN with batch normalization, we compute the mean and the variance of the mini-batch.

In testing we might need to process examples one at a time. The mean and the variance of one example won't make sense. We have to compute an estimated value of mean and variance to use it in testing time.

We can use the weighted average cross the mini-batches. We will use the estimated values of the mean and variance to test. This method is also sometimes called "running average".

In practice most often you will use a deep learning framework and it will contain some default implementation of doing such a thing.

2.4.8 Softmax Regression

In every example we have used so far we were talking about binary classification. There are a generalization of logistic regression called softmax regression that is used for multiclass classification/regression.

Each of C values in the output layer will contain a probability of the example to belong to each of the classes. In the last layer we will have to activate the softmax activation function instead of the sigmoid activation.

$$A[L] = \frac{e^{z[L]}}{\sum e^{z[L]}} \tag{15}$$

2.4.9 Training a Softmax Classifier

There is an activation which is called hard max, which gets 1 for the maximum value and zeros for the others. If you are using NumPy, it's np.max over the vertical axis.

The softmax name came from softening the values and not harding them like hard max.

Softmax is a generalization of logistic activation function to C classes. If C = 2 softmax reduces to logistic regression.

The loss function used with softmax:

$$L(y, \hat{y}) = -\sum y_j * \log \hat{y}_j, \;\; j : 0 \rightarrow C - 1 \tag{16}$$

With m examples:

$$L(Y, \hat{Y}) = -\frac{1}{m} \sum y_j^{(i)} * \log \hat{y}_j^{(i)}, \;\; j : 0 \rightarrow C - 1, i : 0 \rightarrow m \tag{17}$$

Back propagation with softmax:

$$dZ[L] = \hat{Y} - Y \tag{18}$$

That's all for Course II: Improving Deep Neural Networks: Hyperparameter tuning, Regularization and Optimization.

3 Structuring Machine Learning Projects

This is the third course of deep learning specialization at Coursera taught by Professor Andrew Ng. Here is my certificate after finishing this course (Fig. 16):

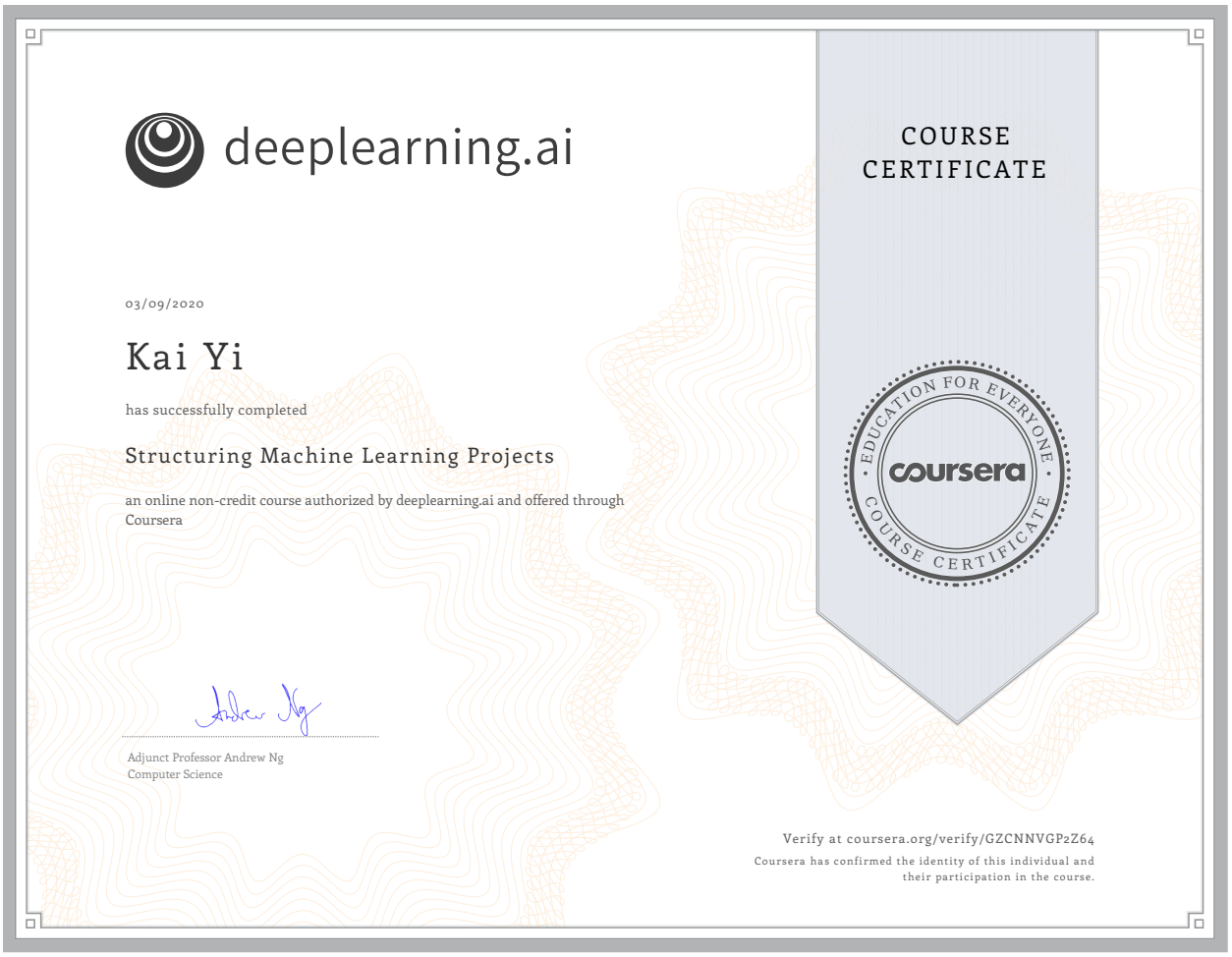


Figure 16: Certificate of Structuring Machine Learning Projects. Obtained in Mar. 2020.

3.1 Course Overview

According to the official site of this course, there are following key components:

- Understand how to diagnose errors in a machine learning system.
- Be able to prioritize the most promising directions for reducing error.
- Understand complex ML settings, such as mismatched training sets, and comparing to and/or surpassing human-level performance.
- Know how to apply end-to-end learning, transfer learning, and multi-task learning.

3.2 Machine Learning (ML) Strategy I

3.2.1 Why ML Strategy

You have a lot of ideas for how to improve the accuracy of your deep learning system:

- Collect more data.

- Collect more diverse training set.
- Train algorithm longer with gradient descent.
- Try different optimization algorithm (e.g. Adam)
- Try bigger network.
- Try smaller network.
- Try dropout.
- Add L2 regularization.
- Change network architecture (activation functions, nums of hidden units, etc)

This course will give you some strategies to help analyze your problem to go in a direction that will help you get better results.

3.2.2 Orthogonalization

Some deep learning developers know exactly what hyperparameter to tune in order to try to achieve one effect. This is a process we call orthogonalization.

In orthogonalization, you have some controls, but each control does a specific task and doesn't affect other controls.

For a supervised learning system to do well, you usually need to tune the knobs of your system to make sure that four things hold true -chain of assumptions in machine learning:

- You'll have to fit training set well on cost function (near human level performance if possible). (If it's not achieved you could try bigger network, another optimization algorithm (like Adam)...)
- Fit dev set well on cost function. (If it's not achieved you could try regularization, bigger training set...)
- Fit test set well on cost function. (If it's not achieved you could try bigger dev set...)
- Performs well in real world. (If it's not achieved you could try to change dev set, change cost function...)

3.2.3 Single Number Evaluation Metric

It's better and faster to set a single number evaluation metric for your project before you start it.

Difference between precision and recall (in cat classification example):

- Suppose we run the classifier on 10 images which are 5 cats and 5 non-cats. The classifier identifies that there are 4 cats, but it identified 1 wrong cat.
- Confusion matrix (Table 1).

-	Predicted Cat	Predicted Non-Cat
Actual Cat	3	2
Actual Non-Cat	1	4

Table 1: Confusion matrix.

- Precision: percentage of true cats in the recognized result: P = 3/(3+1)
- Recall: percentage of true recognition cat of the all cat predictions: R = 3/(3+2)
- Accuracy: (3+4)/10

Using a precision/recall for evaluation is good in a lot of cases, but separately they don't tell you which algorithm is better. E.g. (Table 2):

A better thing is to combine precision and recall in one single (real) number evaluation metric. There is a metric called F1 score, which combines them:

- You can think of F1 score as an average of precision and recall (Equation 19).

$$F1 = \frac{2}{\frac{1}{P} + \frac{1}{R}} \tag{19}$$

Classifier	Precision	Recall
A	95%	90%
B	98%	85%

Table 2: Precision and recall of two classifiers.

3.2.4 Satisfying and Optimizing Metric

Its hard sometimes to get a single number evaluation metric. E.g. (Table 3):

Classifier	F1	Running Time
A	90%	80 ms
B	92%	95 ms
C	92%	1,500 ms

Table 3: F1 and Running Time of 3 classifiers.

So we can solve that by choosing a single optimizing metric and decide that other metrics are satisfying. E.g.:

- Maximize 1 (F1 here) # optimizing metric (one optimizing metric)
- subject to N-1 satisfying metrics (running time < 1, 000 here) # (N-1 satisfying metrics)

3.2.5 Train/dev/test Distributions

Dev and test sets have to come from the same distribution.

Choose dev set and test set to reflect data that you expect to get in the future and consider important to do well on.

Setting up the dev set, as well as the validation metric is really defining what target you want to aim at.

3.2.6 Size of the Dev and Test Sets

An old way of splitting the data was 70% training, 30% test or 60% training, 20% dev and 20% test.

The old way was valid for a number of examples <100000.

In the modern deep learning if you have a million or more examples a reasonable split wound be 98% training, 1% dev and 1% test.

3.2.7 When to Change Dev/Test Sets and Metrics

Let's take an example. In a cat classification example we have these matric results (Table 4):

Algorithm	Classification Error
A	3% error (but a lot of porn images are treated as cat images here)
B	5% error

Table 4: Example of "When to Change Dev/Test Sets and Metrics".

In the above example if we choose the best algorithm by metric it would be "A", but if the users decide if will be "B".

Thus in this case, we want and need to change our metric. The solution is to add weights to select from different criterion.

Conclusion: If doing well on your metric + dev/test set doesn't correspond to doing well in your application, change your metric and/or dev/test set.

3.2.8 Why human-level performance?

We compare to human-level performance because of two main reasons:

- Because of advances in deep learning, machine learning algorithms are suddenly working much better and so it has become much more feasible in a lot of application areas for machine learning algorithms to actually become competitive with human-level performance.

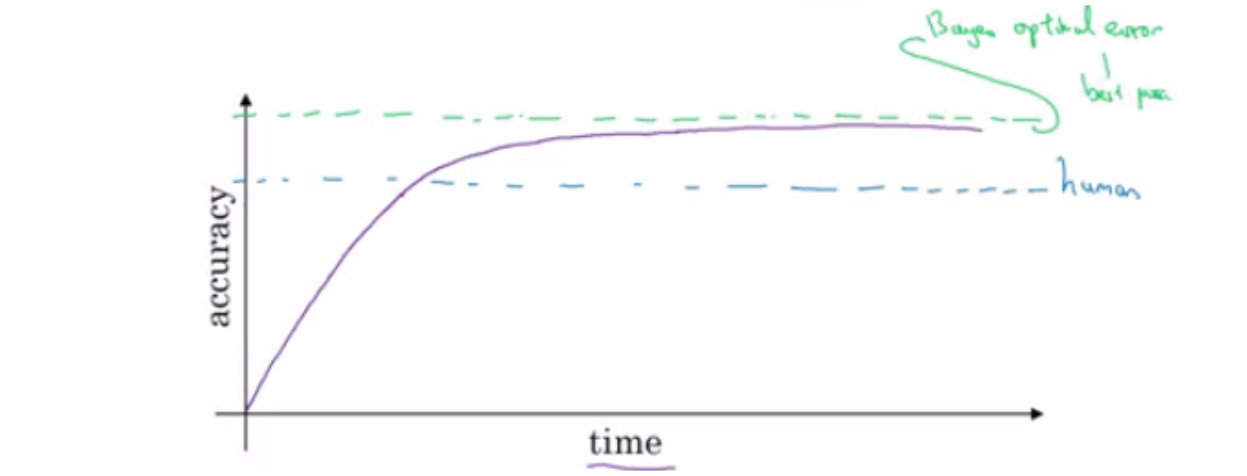


Figure 17: Comparing to human level performance.

- It turns out that the workflow of designing and building a machine learning system is much mor efficient when you're trying to don something that humans can also do.

After an algorithm reaches the human level performance the progress of accuracy slow don 17.

You won't surpass an error that's called "Bayes optimal error".

There isn't much error range between human-level error and Bayes optimal error in computer vision tasks.

Humans are quite good at a lot of tasks. So as long as the machine learning method is worse than humans, you can:

- Get labeled data from humans.
- Gain insight from manual error analysis: why did a person get it right?
- Better analysis of bias/variance.

3.2.9 Avoidable Bias

Suppose that the cat classification algorithm gives these results (Table 5):

Humans	1%	7.5%
Training Error	8%	8%
Dev Error	10%	10%

Table 5: Example of avoidable bias.

In the left example, because the human level error is 1% then we have to focus on the bias.

In the right example, because the human level error is 7.5% then we have to focus on the variance.

The human-level error as a proxy (estimate) for Bayes optimal error. Bayes optimal error is always less (better), but human-level in most cases is not far from it.

Avoidable bias = Training error - Human (Bayes) error

Variance = Dev error - Training error

3.2.10 Understanding Human-Level Performance

When choosing human-level performance, it has to be chosen in the terms of what you want to achieve with the system.

You might have multiple human-level performances based on the human experience. Then you choose the human-level performance (proxy for Bayes error) that is more suitable for the system you're trying to build.

Improving deep learning algorithms is harder once you reach a human-level performance.

Summary of bias/variance with human-level performance:

- Human-level error (proxy for Bayes error). Calculate avoidable bias = training error - human-level error. If avoidable bias difference is the bigger, then it's bias problem and you should use a strategy for bias resolving.
- Training error. Calculate variance = dev error - training error. If variance difference is bigger, then you should use a strategy for variance resolving.
- Dev error.

So having an estimate of human-level performance gives you an estimate of Bayes error. And this allows you to more quickly make decisions as to whether you should focus on trying to reduce a bias or trying to reduce the variance of your algorithm.

These techniques will tend to work well until you surpass human-level performance, whereupon you might no longer have a good estimate of Bayes error that still helps you make this decision really clearly.

3.2.11 Surpassing Human-Level Performance

In some problems, deep learning has surpassed human-level performance. Like:

- Online advertising
- Product recommendation
- Load approval

The listed examples are not natural perception task. Humans are far better in natural perception tasks like computer vision and speech recognition. It's harder for machines to surpass human-level performance in natural perception task. But there are already some systems that achieved it.

3.2.12 Improving Your Model Performance

To improve your deep learning supervised system follow these guidelines:

i. Look at the difference between human level error and the training error - avoidable bias.
ii. Look at the difference between the dev/test set and training set error - variance.
iii. If avoidable bias is large you have these options: 1) Train bigger model; 2) Train longer/better optimization algorithm (like Momentum, RMSprop, Adam); 3) Find better NN architecture/hyperparameters search.
iv. If variance is large you have these options: 1) Get more training data; 2) Regularization (L2, dropout, data augmentation); 3) Find better NN architecture/hyperparameters search.

3.3 Machine Learning Strategy II

3.3.1 Carrying Out Error Analysis

Error analysis - process of manually examining mistakes that your algorithm is making. It can give you insights into what to do next. E.g.:

- In the cat classification example, if you have 10% error on your dev set and you want to decrease the error.
- You discovered that some of the mislabeled data are dog pictures that look like cats. Should you try to make your cat classifier do better on dogs (this could take some weeks)?
- Error analysis approach:
  - Get 100 mislabeled dev set examples at random.
  - Count up how many are dogs.
  - If 5 of 100 are dogs then training your classifier to do better on dogs will decrease your error up to 9.5% (called ceiling), which can be too little.
  - If 50 of 100 are dogs then you could decrease your error up to 5%, which is reasonable and you should work on that.

Based on the above example, error analysis helps you to analyze the error before taking an action that could take a lot of time with no need.

Sometimes, you can evaluate multiple error analysis ideas in parallel and choose the best idea. Create a spreadsheet to do that and decide, e.g. (Table 6):

Image	Dog	Great Cats	Blurry	Instagram Filters	Comments
1	√			√	Pitbull
2	√		√	√	
3					Rainy day at zoo
4		√			
...					
% totals	8%	43%	61%	12%	

Table 6: Spreadsheet of 100 selected images for error analysis.

In the above example you will decide to work on great cats or blurry images to improve your performance.

This quick counting procedure, which you can often do in, at most, small numbers of hours can really help you make much better prioritization decisions, and understand how promising different approaches are to work on.

3.3.2 Cleaning Up Incorrectly Labeled Data

Deep learning algorithms are quite robust to random errors in the training set but less robust to systematic errors. But it's OK to go and fix these labels if you can.

If you want to check for mislabeled data in dev/test set, you should also try error analysis with the mislabeled column. E.g. (Table 7):

Image	Dog	Great Cats	Blurry	Mislabeled	Comments
1	√			√	Pitbull
2	√		√	√	
3					Rainy day at zoo
4		√			
...					
% totals	8%	43%	61%	12%	

Table 7: Spreadsheet of 100 selected images for error analysis.

Then: If overall dev set error is 10%, then errors due to incorrect data is 0.6% while the errors due to other causes is 9.0%. Thus you should focus on the 9.4% error rather than the incorrect data.

Consider these guidelines while correcting the dev/test mislabeled examples:

- Apply the same process to your dev and test sets to make sure they continue to come from the same distribution.
- Consider examining examples your algorithm got right as well as ones it got wrong. (Not always done if you reached a good accuracy)
- Train and (dev/test) data may now come from a slightly different distributions.
- It's very important to have dev and test sets to come from the same distribution. But it could be OK for a train set to come from slightly different distribution.

3.3.3 Build You First System Quickly, Then Iterate

The steps you take to make your deep learning project:

- Setup dev/test set and metric.
- Build initial system quickly.
- Use bias/variance analysis & error analysis to prioritize next steps.

3.3.4 Training and Testing on Different Distributions

A lot of teams are working with deep learning applications that have training sets that are different from the dev/test sets due to the hunger of deep learning to data.

There are some strategies to follow up when training set distribution differs from dev/test sets distribution:

- Option I (not recommended): shuffle all the data together and extract randomly training and dev/test sets.
  - Advantages: all the sets now come from the same distribution.
  - Disadvantages: the other (real world) distribution that was in the dev/test sets will occur less in the new dev/test sets and that might be not what you want to achieve.
- Option II: take some of the dev/test set examples and add them to the training set.
  - Advantages: the distribution you care about is the target now.
  - Disadvantages: the distributions in training and dev/test sets are now different. But you will get a better performance over a long time.

3.3.5 Bias and Variance With Mismatched Data Distributions

Bias and variance analysis changes when training and dev/test set is from the different distribution.

Example: the cat classification example. Suppose you've worked in the example and reached this:

- Human error: 0%
- Train error: 1%
- Dev error: 10%

In this example, you may think this is a variance problem, but because the distributions aren't the same you can't tell for sure. Because it could be that train set was easy to train on, but the dev set was more difficult.

To solve this issue, we create a new set called train-dev set as a random subset of the training set (so it has the same distribution) and we get:

- Human error: 0%
- Train error: 1%
- Train-dev error: 9%
- Dev error: 10%

Now we are sure that this is a high variance problem.

Suppose we have a different situation:

- Human error: 0%
- Train error: 1%
- Train-dev error: 1.5%
- Dev error: 10%

In this case, we have something called *data mismatch* problem.

Conclusions:

- Human-level error (proxy for Bayes error)
- Train error
  - Calculate *avoidable bias = training error - human level error*.
  - If the difference is big then its Avoidable bias problem then you should use a strategy for high bias.
- Train-dev error
  - Calculate *variance = training-dev error - training error*.
  - If the difference is big then its high variance problem then you should use a strategy for solving it.

- Dev error
  - Calculate *data mismatch = dev error - train-dev error*.
  - If difference is much bigger then train-dev error its Data mismatch problem.
- Test error
  - Calculate *degree of overfitting to dev set = test error - dev error*.
  - Is the difference is big (positive) then maybe you need to find a bigger dev set (dev set and test set come from the same distribution, so the only way for there to be a huge gap here, for it to do much better on the dev set than the test set, is if you somehow managed to overfit the dev set).

Unfortunately, there aren't many systematic ways to deal with data mismatch. There are some things to try about this in the next section.

3.3.6 Addressing Data Mismatch

There aren't completely systematic solutions to this, but there are some things you could try.

- Carry out manual error analysis to try to understand the difference between training and dev/test sets.
- Make training data more similar, or collect more data similar to dev/test sets.

If your goal is to make the training data more similar to your dev set one of the techniques you can use is *artificial data synthesis* that can help you make more training data.

- Combine some of your training data with something that can convert it to the dev/test set distribution. E.g.:
  - Combine normal audio with car noise to get audio with car noise example.
  - Generate cars using 3D graphics in a car classification example.
- Be cautious and bear in mind whether or not you might be accidentally simulating data only from a tiny subset of the space of all possible examples because your NN might overfit these generated data (like particular car noise or a particular design of 3D graphics cars).

3.3.7 Transfer Learning

Apply the knowledge you took in a task A and apply it in another task B.

For example, you have trained a cat classifier with a lot of data, you can use the part of the trained NN to solve medical image classification problem.

To do transfer learning, delete the last few layers of NN and theis weights and:

- Option I: If you have a small data set - keep all the other weights as a fixed weights. Add a new last layer(-s) and initialize the new layer weights and feed the new data to the NN and learn the new weights.
- Option II: If you have enough data you can retrain all the weights.

Option I and II are called **fine-tuning** and training on task A called **pretraining**.

When transfer learning make sense:

- Task A and B have the same input X (e.g. image, audio).
- You have a lot of data for the task A you are transferring from and relatively less data for the task B your transferring to.
- Low level features from task A could be helpful for learning task B.

3.3.8 Multi-Task Learning

Whereas in transfer learning, you have a sequential process where you learn from task A and then transfer that to task B. In multi-task learning, you start off simultaneously, trying to have one neural network do several things at the same time. And then each of these tasks helps hopefully all of the other tasks.

Example: You want to build an object recognition system that detects pedestrians, cars, stop signs and traffic lights (image has multiple labels).

Then Y shape will be (4,m) because we have 4 classes and each one is a binary one.

Then.

$$Cost = \frac{1}{m}\sum_{i=1}^{m}\sum_{j=1}^{4} L(\hat{y}_j^i, y_j^i), \;\; i = 1 \to m, j = 1 \to 4$$
$$L = -y_j^i * \log(\hat{y}_j^i) - (1 - y_j^i) * \log(1 - \hat{y}_j^i) \tag{20}$$

In the above example, you could have trained 4 neural networks separately but if some of the earlier features in NN can be shared among these different types of objects, then you find that training one NN to do four things results in better performance than training 4 completely separate NNs to do the four tasks separately.

Multi-task learning will also work if y isn't complete for some labels. E.g.:

```
Y = [1  ?  1  ...]
    [0  0  1  ...]
    [?  1  ?  ...]
```

In the above case, it will do good with the missing data, just the loss function will be different:

$$L = -y_j^i * \log(\hat{y}_j^i) - (1 - y_j^i) * \log(1 - \hat{y}_j^i) \;\; for\ all\ j\ which\ y_j^i != ? \tag{21}$$

Multi-task learning makes sense:

- Training on a set of tasks that could benefit from having shared lower-level features.
- Usually, amount of data you have for each task is quite similar.
- Can train a big enough network to do well on all the tasks.

If you can train a big enough NN, the performance of the multi-task learning compared to splitting the tasks is better.

Today transfer learning is used more often than multi-task learning. But in the area of computer vision, multi-task learning is widely adapted.

3.3.9 What Is End-to-end Deep Learning

Some systems have multiple stages to implement. An end-to-end deep learning system implements all these stages with a single NN. E.g.:

- Speech recognition system:

```
# non-end-to-end system
Audio ---> Features --> Phonemes --> Words --> Transcript
# end-to-end deep learning system
Audio ------------------------------------------------> Transcript
```

- End-to-end deep learning gives data more freedom, it might not use phonemes when traning.

To build the end-to-end deep learning system that works well, we need a big dataset (more data then in non end-to-end system). If we have a small dataset the ordinary implementation could work just fine.

Another example of end-to-end learning:

- Face recognition system:

```
# end-to-end deep learning system
Image ------------------------> Face recognition
# deep learning system - best approach for now
Image --> Face detection --> Face recognition
```

- In practice, the best approach is the second one for now.
- In the second implementation, it's a two steps approach where both parts are implemented using deep learning.
- Its working well because it's harder to get a lot of pictures with people in front of the camera than getting faces of people and compare them.
- In the second implementation at the last step, the NN takes two faces as an input and outputs if the two faces are the same person or not (Siamese Neural Network[1]).

Example 3:

- Machine translation system:

```
# non-end-to-end system
English --> Text analysis --> ... --> French
# end-to-end deep learning system - best approach
English ------------------------------> French
```

- Here end-to-end deep leaning system works better because we have enough data to build it.

Example 4:

- Estimating child's age from the x-ray picture of a hand:

```
# non-end-to-end system - best approach for now
Image --> Bones --> Age
# end-to-end deep learning system
Image ------------> Age
```

- In this example non-end-to-end system works better because we don't have enough data to train end-to-end system.

3.3.10 Whether to Use End-to-end Deep Learning

Pros of end-to-end deep learning:

- Let the data speak. By having a pure machine learning approach, your NN learning input from X to Y may be more able to capture whatever statistics are in the data, rather than being forced to reflect human preconceptions.
- Less hand-designing of components needed.

Cons of end-to-end deep learning:

- May need a large amount of data.
- Excludes potentially useful hand-design components (it helps more on the smaller dataset).

Applying end-to-end deep learning:

- Key question: Do you have sufficient data to learn a function of the **complexity** needed to map x to y?
- When applying supervised learning you should carefully choose what types of X to Y mappings you want to learn depending on what task you can get data for.

4 Convolutional Neural Networks

This is the fourth course of deep learning specialization at Coursera taught by Professor Andrew Ng. Here is my certificate after finishing this course (Fig. 18):

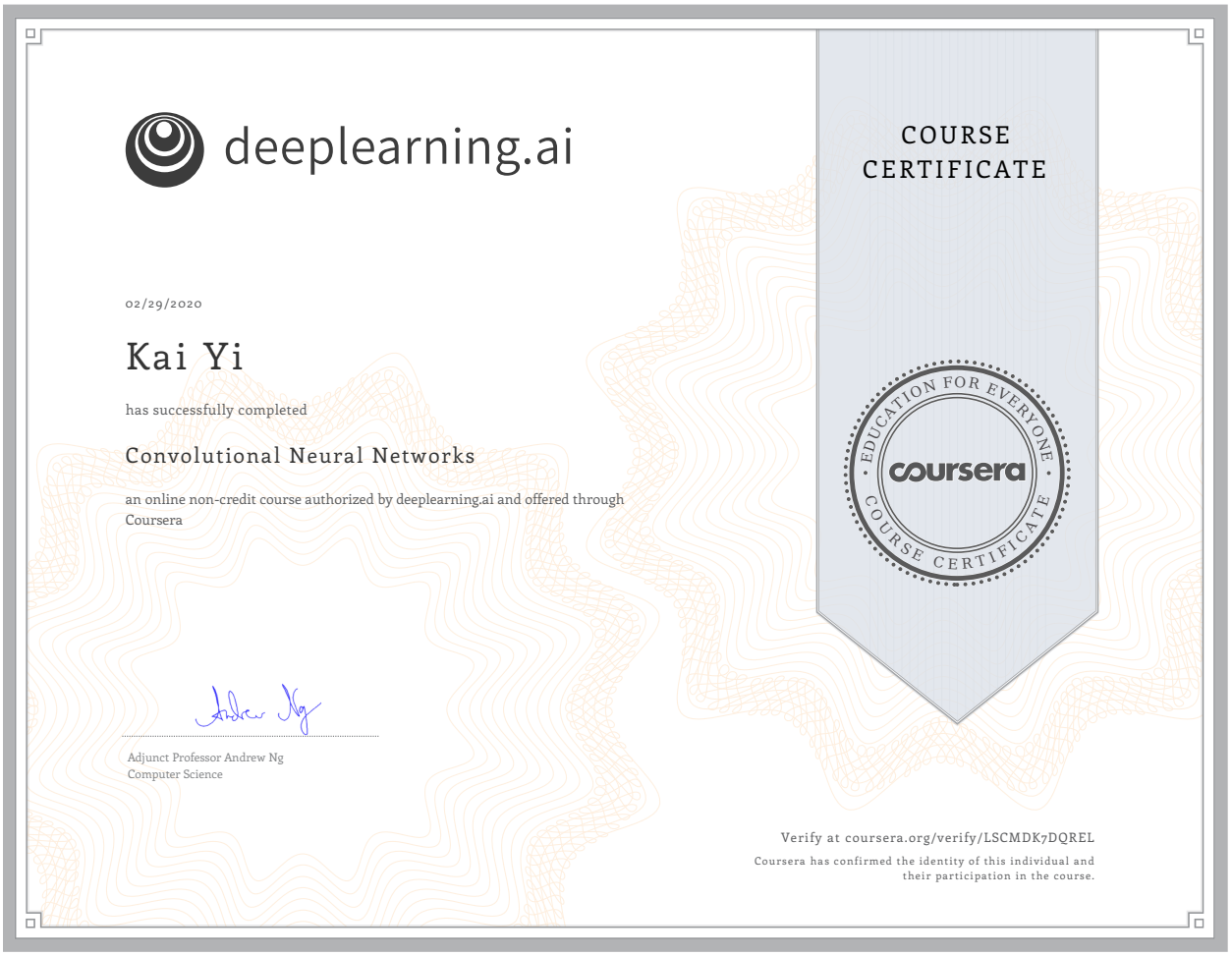


Figure 18: Certificate of Convolutional Neural Networks. Obtained in Feb. 2020.

4.1 Course Overview

According to the official site of this course, there are following key components:

- Understand how to build a convolutional neural network, including recent variations such as residual networks.
- Know how to apply convolutional networks to visual detection and recognition tasks.
- Know to use neural style transfer to generate art.
- Be able to apply these algorithms to a variety of image, video, and other 2D or 3D data.

4.2 Foundations of CNNs

The aims of this section is to implement the foundational layers of CNNs (pooling, convolutions) and to stack them properly in a deep network to solve multi-class image classification problems.

4.2.1 Edge Detection Example

Early layers of CNN might detect edges then the middle layers will detect parts of objects and the later layers will put these parts together to produce an output.

4.2.2 Padding, Stride

$$C_{out} = (C_{in} - k + 2 * p)/s + 1 \tag{22}$$

k represents the kernel size, p represents padding and s represents stride.

4.2.3 Pooling layers

Other than the conv layers, CNNs often uses pooling layers to reduce the size of the inputs, speed up computation, and to make some of the features it detects more robust.

The max pooling is saying, if the feature is detected anywhere in this filter then keep a high number. But the main reason why people are using pooling because its works well in practice and reduce computations.

Max pooling has no parameters to learn.

Average pooling is taking the averages of the values instead of taking the max values.

Max pooling is used more often than average pooling in practice.

If stride of pooling equals the size, it will then apply the effect of shrinking.

4.2.4 Convolutional Neural Network Example

Here are some statistics about a mentioned example (Fig. 19):

	Activation shape	Activation Size	# parameters
Input:	(32,32,3)	3,072	0
CONV1 (f=5, s=1)	(28,28,8)	6,272	208
POOL1	(14,14,8)	1,568	0
CONV2 (f=5, s=1)	(10,10,16)	1,600	416
POOL2	(5,5,16)	400	0
FC3	(120,1)	120	48,001
FC4	(84,1)	84	10,081
Softmax	(10,1)	10	841

Figure 19: Some statistics about a LeNet-like network. Every parameter is calculated by $(k^2 + 1) * c$, k represents kernel size and c represents output channel.

Usually the input size decreases over layers which the number of filters increases.

A CNN usually consists of one or more convolution (not just one as the shown example) followed by a pooling.

Fully connected layers has the most parameters in the network.

4.2.5 Why Convolutions?

Two main advantages of Convs are:

- Parameter sharing. A feature detector (such as a vertical edge detector) that's useful in one part of the image is probably useful in another part of the image.
- Sparsity of Connections. In each layer, each output value depends only on a small number of inputs which makes it translation invariance (Translation invariance means that the system produces exactly the same response, regardless of how its input is shifted.).

4.3 Deep Convolutional Models: Case Studies

The goal of this section is to learn about the practical tricks and methods used in deep CNNs straight from the research papers.

4.3.1 Classic Networks

In this subsection we will talk about classic networks which are LeNet-5, AlexNet and VGG.

LeNet-5 The goal for this model was to identify handwritten digits in a $32 \times 32 \times 1$ gray image. Here is the drawing of it (Fig. 21):

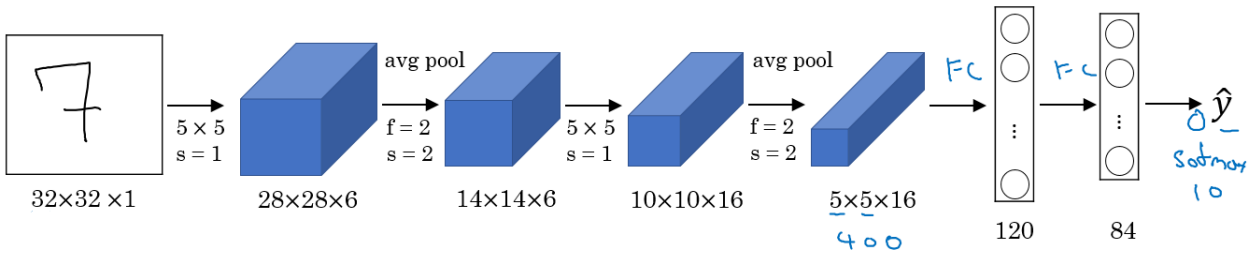


Figure 20: LeNet architecture.

This model was published in 1998 [2]. The last layer wasn't using softmax back then.

It has 60k parameters.

The dimensions of the image decreases as the number of channels increases.

$Conv \rightarrow Pool \rightarrow Conv \rightarrow Pool \rightarrow FC \rightarrow FC \rightarrow Softmax$ this type of arrangement is quite common.

The activation function used in the paper was Sigmoid and Tanh. Modern implementation uses ReLU in most of the cases.

AlexNet Named after Alex Krizhevsky who was the first author of this paper [3]. The other authors includes Geoffrey Hinton.

The goal for this model was the ImageNet challenge which classifies images into 1000 classes. Here are the drawing of this model:

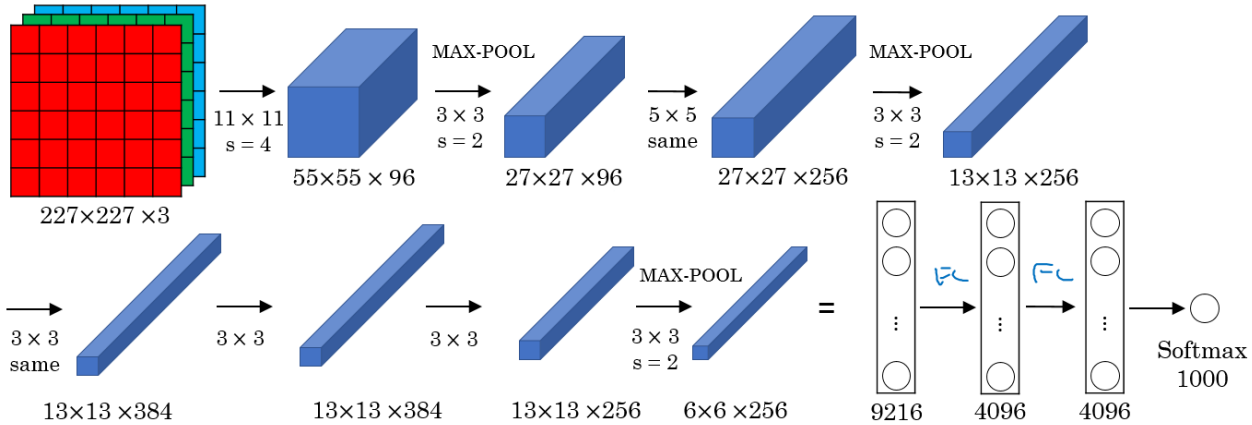


Figure 21: AlexNet architecture.

Architecture:

$Conv \rightarrow Max - pool \rightarrow Conv \rightarrow Max - pool \rightarrow Conv \rightarrow Conv \rightarrow Conv \rightarrow Max - pool \rightarrow Flatten \rightarrow FC \rightarrow FC \rightarrow Softmax$

Similar to LeNet-5 but bigger.

Has 60 million parameters compared to 60k parameterof LeNet-5.

It used the ReLU activation function.

The original paper contains multiple GPUs and Local Response Normalization (LRN). Multiply GPUs were used because the GPUs were not so fast back then. Researchers proved that LRN doesn't help much so for now don't botter yourself for understanding or implementing it.

This paper convinced the computer vision researchers that deep learning is so important.

VGG-16 VGG-16 [4] is a modification fro AlexNet.

Instead of having a lot of hyperparameters let's have some simpler network.

Focus on having only these blocks:

- Conv = 3×3, filter, s = 1, 'SAME'
- Max-Pool = 2×2, s = 2

Here is the architecture (Fig. 22):

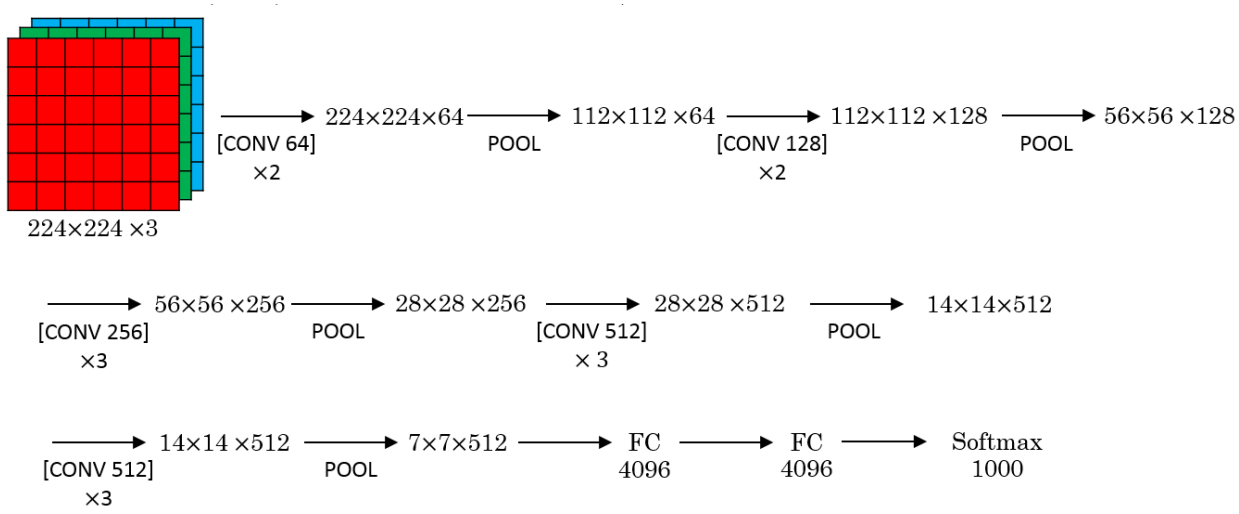


Figure 22: VGG-16 architecture.

This network is large even by modern standards. It has around 138 million parameters (Most of the parameters are in the fully connected layers).

In has a total memory of 96MB per image for only forward propagation (Most memory are in the earlier layers).

Number of filers increases from 64 to 128 to 256 to 512. 512 was made twice.

Pooling was the only one who is responsible for shrinking the dimensions.

There are another version called VGG-19 which is a bigger version. But most people uses the VGG-16 instead of the VGG-19 because it does the same.

VGG paper is attractive it tries to make some rules regarding using CNNs.

4.3.2 Residual Networks (ResNets)

Very, very deep NNs are difficult to train becaus of vanishing and exploding gradiednts problems.

In this section we will learn about skip connection which makes you take the activation from one layer and subsequently feed it to another layer even much deeper in NN which allows you to train large NNs even with layers greater than 100.

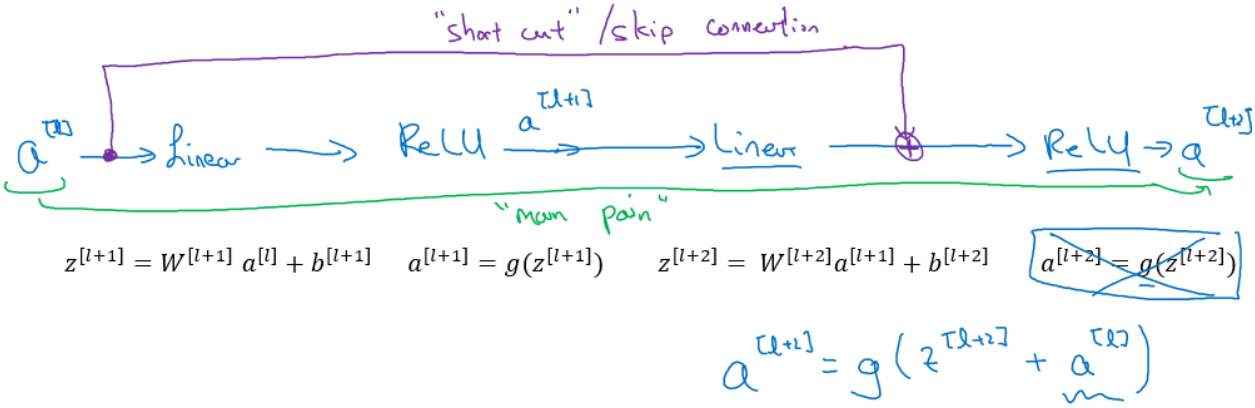


Figure 23: Residual block architecture.

Residual Block ResNets are built out of some Residual blocks (Fig. 23).

They add a shortcut/skip connection before the second activation.

The authors of this block find that you can train a deeper NNs by stacking this block.

Residual Network Are a NN that consists of some Residual blocks (Fig. 24).

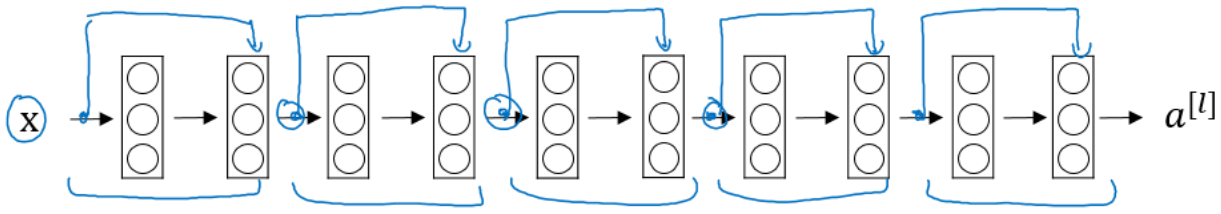


Figure 24: Residual network architecture.

These networks can go deeper without hurting the performance. In the normal NN - Plain networks - the theory tell us that if we go deeper we will get a better solution to out problem, but because of the vanishing and exploding gradients problems the performances of the network suffers as it goes deeper. Thanks to Residual Network we can go deeper as we want now (Fig. 25).

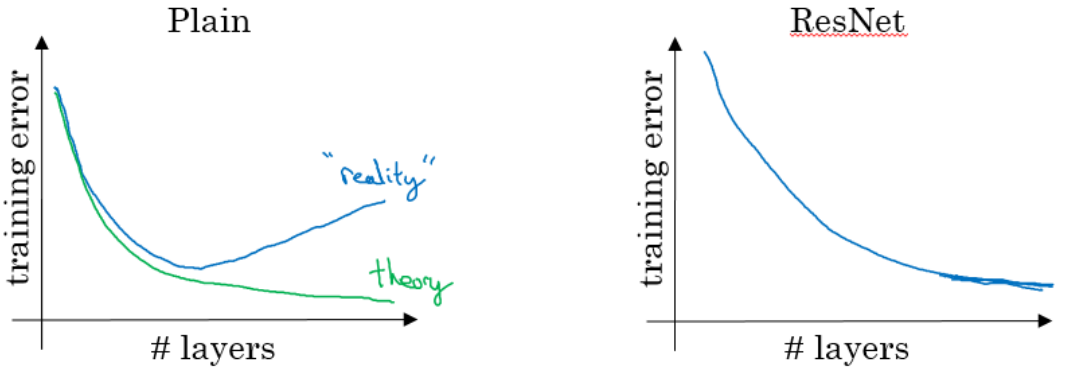


Figure 25: Plain networks vs. ResNet

On the left is the normal NN and on the right are the ResNet. As you can see the performance of ResNet increases as the network goes deeper.

In some cases going deeper won't effect the performance and that depends on the problem on your hand.

Some people are trying to train 1000 layer now which isn't used in practice.

4.3.3 Why ResNets Work

Lets see some example that illustrates why ResNets work.

We have a big NN as the following: $X \rightarrow Big\ NN \rightarrow a[l]$.

Lets add two layers to this network as a residual block: $X \rightarrow Big\ NN \rightarrow a[l] \rightarrow layer1 \rightarrow layer2 \rightarrow a[i+2]$. $a[i]$ has a direct connection to $a[i+2]$.

Suppose we are using ReLU activations.

Then:

```
a[l+2] = g(z[l+2] + a[l])
       = g(W[l+2] a[l+1] + b[l+2] + a[l])
```

Then if we are using L2 regularization as an example, Let's say $W[l+2]$ and $b[l+2]$ will be zero. Then $a[l+2] = g(a[l]) = a[l]$ with no negative values.

This show that identity function function is easy for a residual block to learn. And that's why it can train deeper NNs.

Also that the two layers we added doesn't hurt the performance of big NN we made.

Hint: dimensions of z[l+2] and a[l] have to be the same in resNets. In case they have different dimensions what we put a matrix parameters (Which can be learned or fixed)

- $a[l+2] = g(z[l+2] + ws * a[l])$ # The added Ws should make the dimensions equal.
- ws also can be a zero padding.

Using a skip-connection helps the gradient to backpropagate and thus helps you to train deeper networks.

Lets take a look at ResNet on images. Here is the architecture of ResNet-34 (Fig. 26):

In ResNet-34:

- All the 3×3 Conv are same Convs.
- Keep it simple in design of the network.
- Spatial Size / 2 $\rightarrow$ Number of filters * 2.
- No FC layers, no dropout is used.
- Two main types of blocks are used in a ResNet, depending mainly on whether the input/output dimensions are same or different. You are going to implement both of them.
- The dotted lines is the case when the dimensions are different. To solve then they down-sample the input by 2 and then pad zeros to match the two dimensions. There's another trick which is called bottleneck which we will explore later.

4.3.4 Residual Block Types

Identity Block Here is the identity block (Fig. 27):

The Conv is followed by a batch norm before ReLU. Dimensions here are same.

This skip is over 2 layers. The skip connection can jump n connections where n>2.

Convolutional Block Here is the convolutional block (Fig. 28):

The Conv can be bottleneck 1×1 Conv.

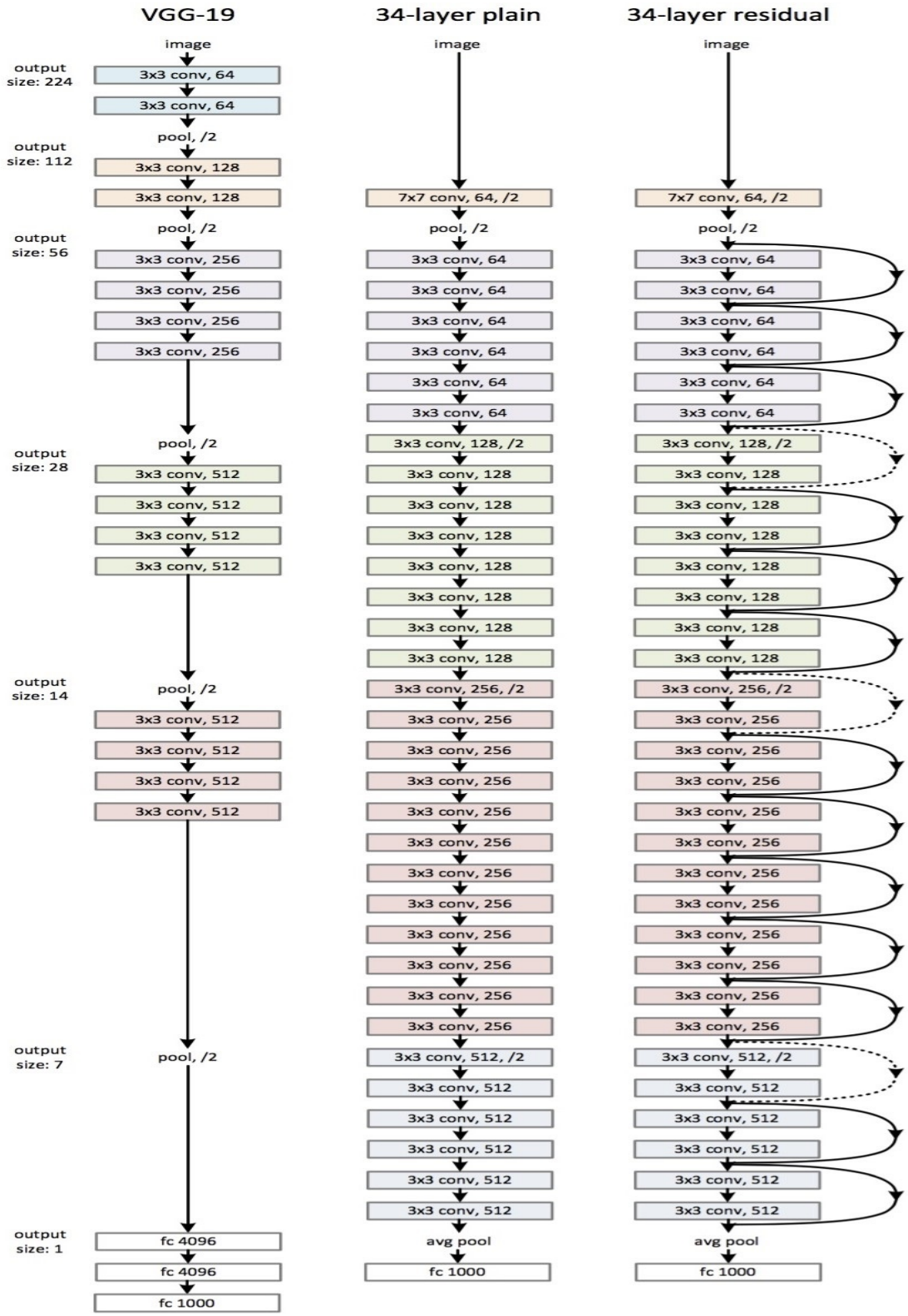


Figure 26: ResNet-34 compared with VGG-19 and 34-layer plain network.

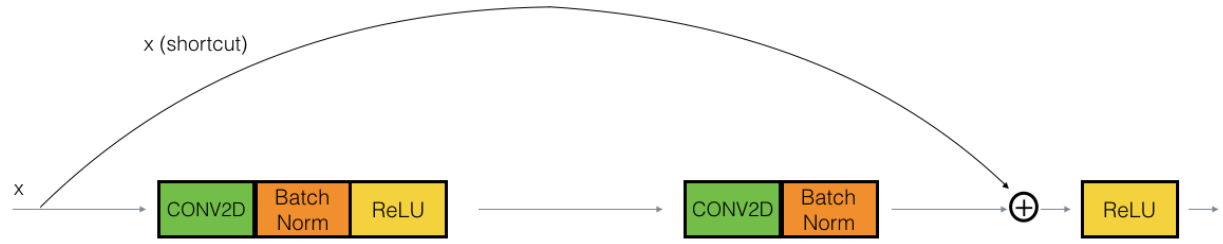


Figure 27: Identity block.

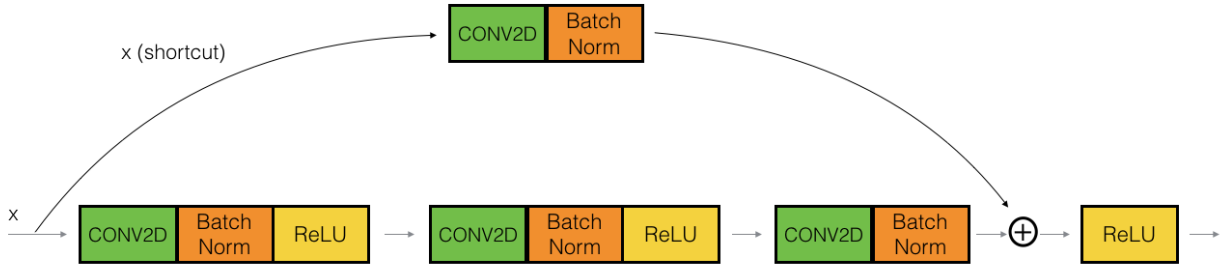


Figure 28: Convolutional block.

4.3.5 Network in Network and 1 Times 1 Convolutions

A 1×1 convolution - a very important operation in Network in Network [5]- is very useful in many CNN modes.

A 1×1 convlution is useful when:

- We want to shrink the number of channels. We also call this feature transformation.
- Save computations.
- If we have specified the number of 1 x 1 Conv filters to be the same as the input number of channels then the output will contain the same number of channels. Then the 1 x 1 Conv will act like a non linearity and will learn non linearity operator.

4.3.6 Inception Network Motivation

When you design a CNN you have to decide all the layers yourself. You may pick a 3×3 Conv or 5×5 Conv or maybe a max pooling layer. You have so many choices.

What **Inception** tell us is, why not use all of them at one?

Inception module [6], naive version (Fig. 29):

Max-pool is 'SAME' here.

Input to the inception module is $28 \times 28 \times 192$ and the output is $28 \times 28 \times 256$.

We have done all the Convs and pools we might want and will let the NN learn and decide which it want to use most.

There is high computational cost in Inception model especially for convolution operations with big kernel size. However, we can use 1×1 convolution to reduce the total number of multiplications.

A 1×1 Conv here is called Bottleneck. It turns out that 1×1 Conv won't hurt the performance.

Inception module, dimensions reduction version (Fig. 30):

Example of inception model in Keras (Figure 31):

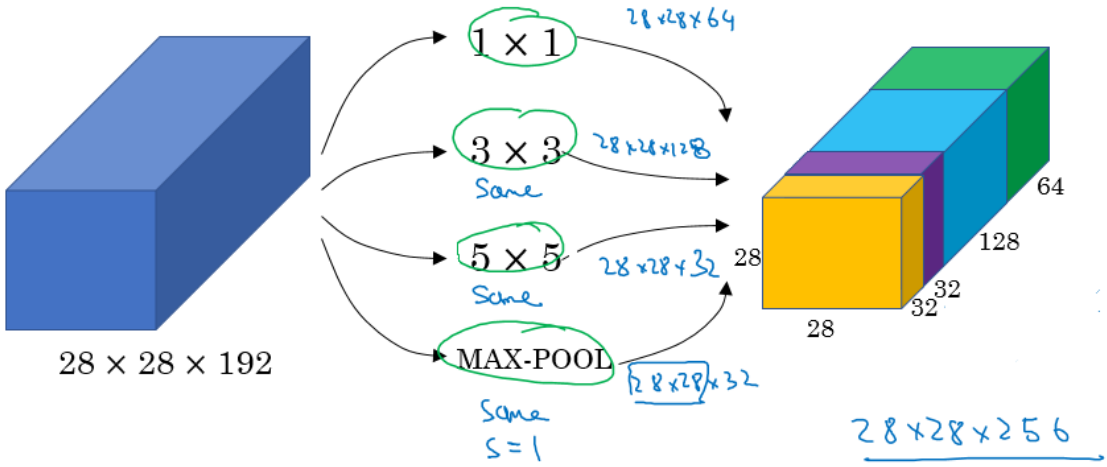


Figure 29: Inception module, naive version.

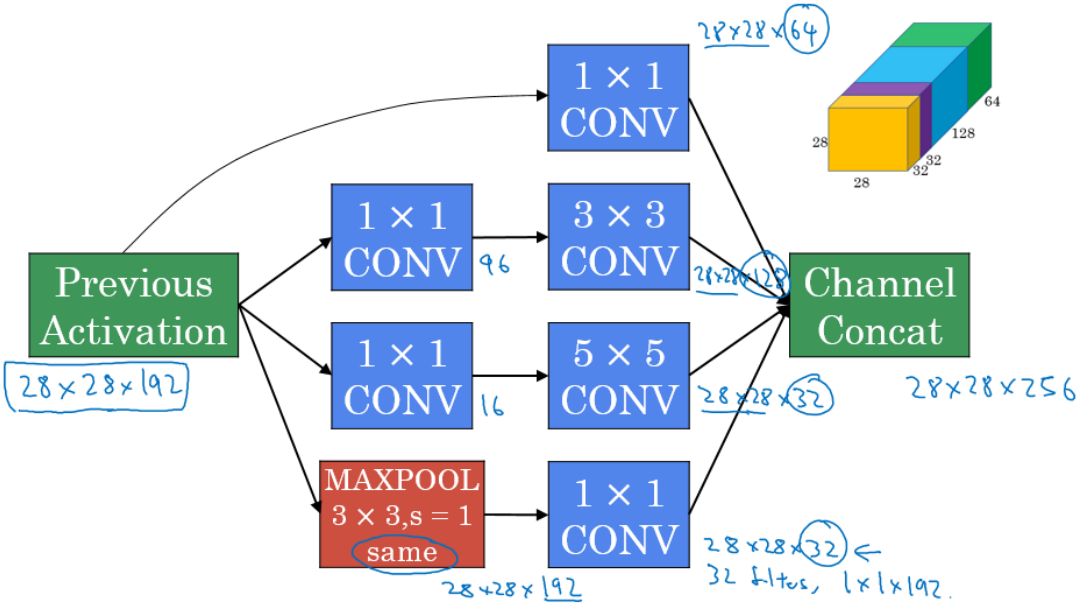


Figure 30: Inception module, dimensions reduction version.

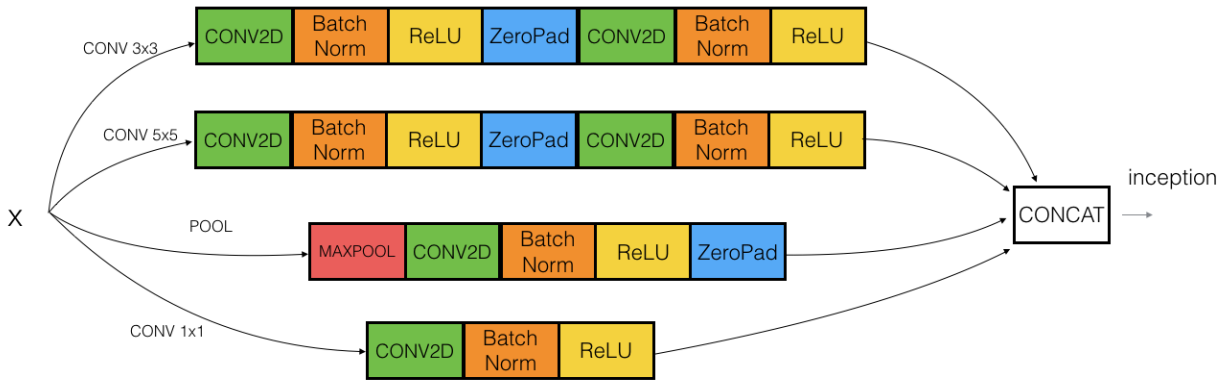


Figure 31: Inception module in Keras

4.3.7 Inception Network (GoogLeNet)

The inception network consist of concatenated blocks of the Inception module.

The full model of GoogLeNet [6] can be found at Here (Also Figure 32):

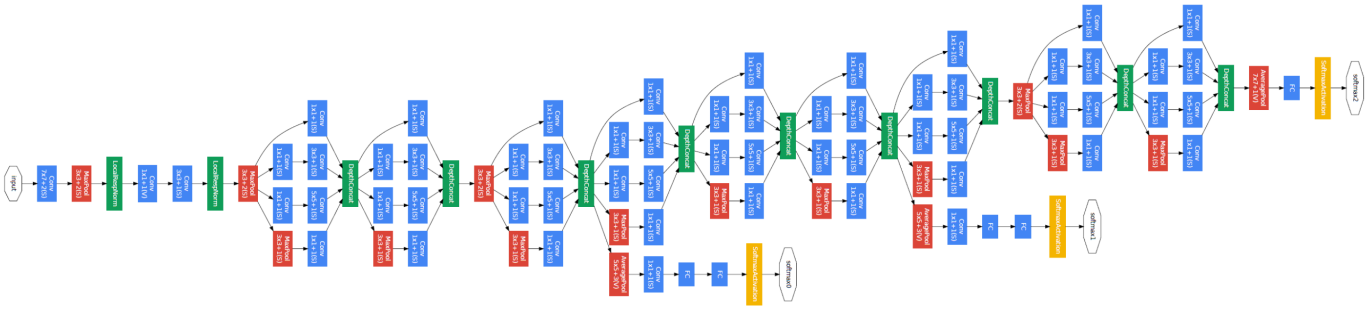


Figure 32: GoogLeNet architecture.

Some times a max-pool block is used before the inception module to reduce the dimensions of the inputs.

There are 3 softmax branches at different positions to push the network toward its goal, and helps to ensure that the intermediate features are good enough to the network and it turns out that softmax0 and softmax1 gives regularization effect.

Since the development of the Inception module, the authors and the others have built another versions of this network. Like inception v2, v3, and v4. Also there is a network that has used the inception module and the ResNet together.

4.3.8 Transfer Learning

If you're using a specific NN architecture that has been trained before, you can use this pretained parameters/weights instead of random initialization to solve your problem.

It can help you boost the performance of the NN.

The pretrained models might have trained on a large dataset like ImageNet or COCO and took a lot of time to learn those parameters/weights with optimized hyperparameters. This can save you a lot of time (parameters freezing technique).

4.3.9 Data Augmentation

If data is increased, your deep NN will perform better. Data augmentation is one of the technique that deep learning uses to increase the performance of deep NN.

The majority of computer vision applications needs more data right now.

Some data augmentation methods that are used for computer vision tasks includes:

- Mirroring.
- Random cropping.
  - The issue with this technique is that you might take a wrong crop.
  - The solution is to make your crops big enough.
- Rotation.
- Shearing.
- Local warping.
- Color shifting.
  - For example, we add to R, G, and B some distortions that will make the image identified as the same for the human but is different for the computer.
  - In practice the added value are pulled from some probability distribution and these shifts are some small.
  - Makes your algorithm more robust in changing colors in images.
  - There are an algorithm which is called **PCA color augmentation** that decides the shifts needed automatically.

Implementing distortions during training:

You can use a different CPU thread to make you a distorted mini batches while you are training your NN.

Data Augmentation has also some hyperparameters. A good place to start is to find an open source data augmentation implementation and then use it or fine tune these hyperparameters.

Tips for doing well on benchmarks/winning competitions:

I). Ensembling.

- Train several networks independently and average their outputs. Merging down some classifiers.
- After you decide the best architecture for your problem, initialize some of that randomly and train them independently.
- This can give you a push by 2%.
- But this will slow down your production by the number of the ensembles. Also it takes more memory as it saves all the models in the memory.
- People use this in competitions but few uses this in a real production.

II). Multi-crop at test time.

- Run classifier on multiple versions of test versions and average results.
- There is a technique called 10 crops that uses this.
- This can give you a better result in the production.

4.4 Object Detection

The aim of this section is to learn how to apply your knowledge of CNNs to one of the toughest but hottest field of computer vision: object detection.

4.4.1 Object Localization

What are localization and detection?

Image Classification Classify an image to a specific class. The whole image represents one class. We don't want to know exactly where is the object. usually one object is presented.

Classfication With Localization Given an image we want to learn the class of the image and where is the class location in the image. We need to detect a class and a rectangle of where the object is. Usually only one object is presented.

Object Detection Given an image we want to detect all the object in the image that belong to a specific classes and give their location. An image can contain more than one object with different classes.

Semantic Segmentation We want to label each pixel in the image with a category label. Semantic segmentation don't differentiate instances, only care about pixels. It detects no objects just pixels.

If there are two objects of the same class is intersected, we won't be able to separate them.

Instance Segmentation This is like the full problem. Rather than we want to predict the bounding box, we want to know which pixel label but also distinguish them.

To make image classification we use a ConvNet with a softmax attached to the end of it.

The make classification with localization we use a ConvNet with a softmax attached to the end of it and a four numbers bx, by, bh and bw to tell you the location of the class in the image. The dataset should contain this four numbers with the class too.

4.4.2 Landmark Detection

In some of the computer vision problems, you will need to output some points. That is called **landmark detection**.

For example, if you're working on a face recognition problem you might want some points on the face like corners of the eyes, corners of the mouth, and corners of the nose and so on. This can help in a lot of application like detecting the pose of the face.

Y shape for the face recognition problem that needs to output 64 landmarks:

```
Y = [
        ThereIsAface    # Probability of face is presented 0 or 1
        l1x ,
        l1y ,
        .... ,
        l64x ,
        l64y
]
```

Another application is when you need to get the skeleton of the person using different landmarks/points in the person which helps in some applications.

In your labeled data, if l1x,l1y is the left corner of left eye, all other l1x,l1y of the other examples has to be the same.

4.4.3 Object Detection

We will use a ConvNet first to solve the object detection problem using a technique caleed the sliding windows detection algorithm.

For example, let's say we're working on Car object detection.

The first thing, we will train a ConvNet on cropped car images and nor car images. After we finish feeding this ConvNet with data, we will then use it with the sliding windows technique.

Sliding windows detection algorithm:

i. Decide a rectangle size.
ii. Split your image rectangles of the size you picked. Each region should be covered. You can use some strides.
iii. For each rectangle feed the image into the ConvNet and decide if its a car or not.
iv. Pick larger/smaller rectangles and repeat the process from ii. to iii.
v. Store the rectangles that contains the cars.
vi. If two or more rectangles intersects choose the rectangle with the best accuracy.

Disadvantage of sliding window is the computation time.

In the era of machine learning before deep learning, people used a hand crafted linear classifiers that classifies the object and then use the sliding window technique. The linear classifier makes it a cheap computation. But in the deep learning era that is so computational expensive due to the complexity of the deep learning model.

To solve this problem, we can implement the sliding windows with a **Convolutional approach**.

One other idea is to compress your deep learning model.

4.4.4 Convolutional Implementation of Sliding Windows

Turning FC layer into convolutional layers (predict image class from four classes) (Figure 33):

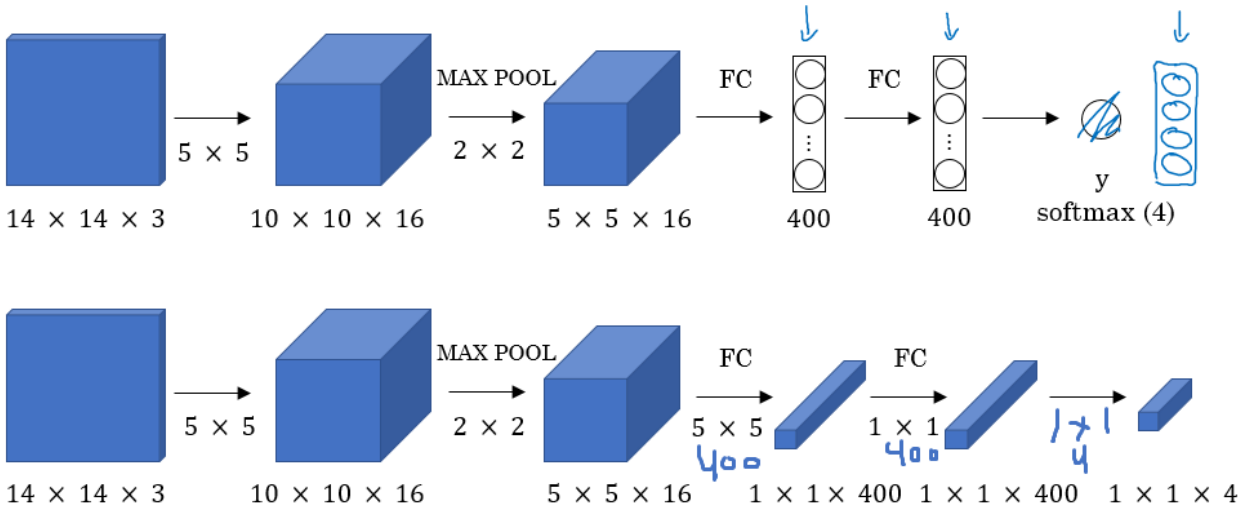


Figure 33: Convolutional implementation of sliding windows.

As you can see in the above image, we turned the FC layer into a Conv layer using a convolution with the width and height of the filter which is the same as the width and height of the input.

The convolution implementation of sliding windows:

- First lets consider that the Conv net you trained is like this (Figure 34. No FC all is Conv layers.)

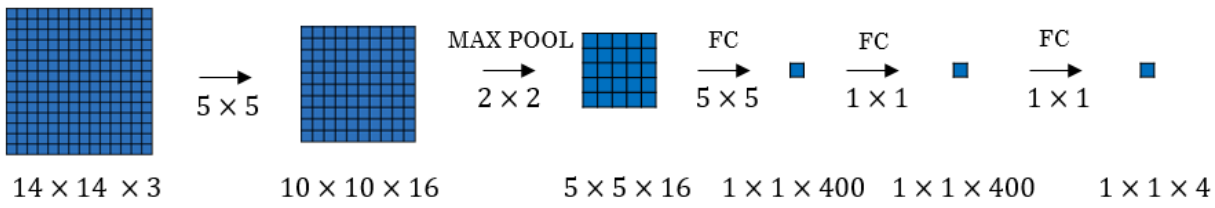


Figure 34: Convolutional implementation of sliding windows (Visualization).

- Say now we have a $16 \times 16 \times 3$ image that we need to apply the sliding windows in. By the normal implementation that has been mentioned in the section before, we would run this Conv net four times each rectangle size will be 14×14.
- The convolution implementation will be as follows:
- The left cell of the result "blue one" will represent the first sliding window of the normal implementation. The other cells will represent the others.
- Its more efficient because it now shares the computations of the four times needed.

The weakness of the algorithm is that the position of the rectangle won't be so accurate. Maybe none of the rectangles is exactly on the object you want to recognize (Figure 36).

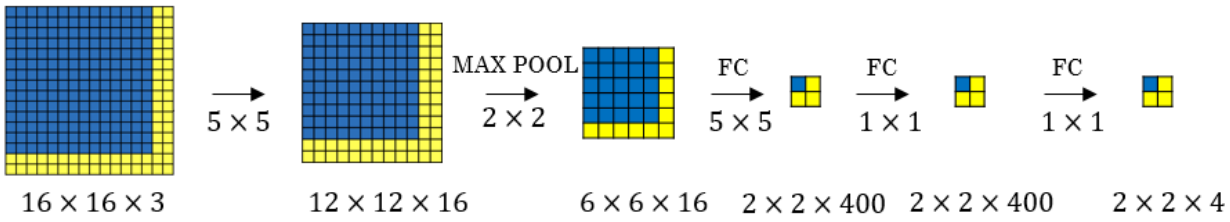


Figure 35: Convolutional implementation of sliding windows (Visualization).

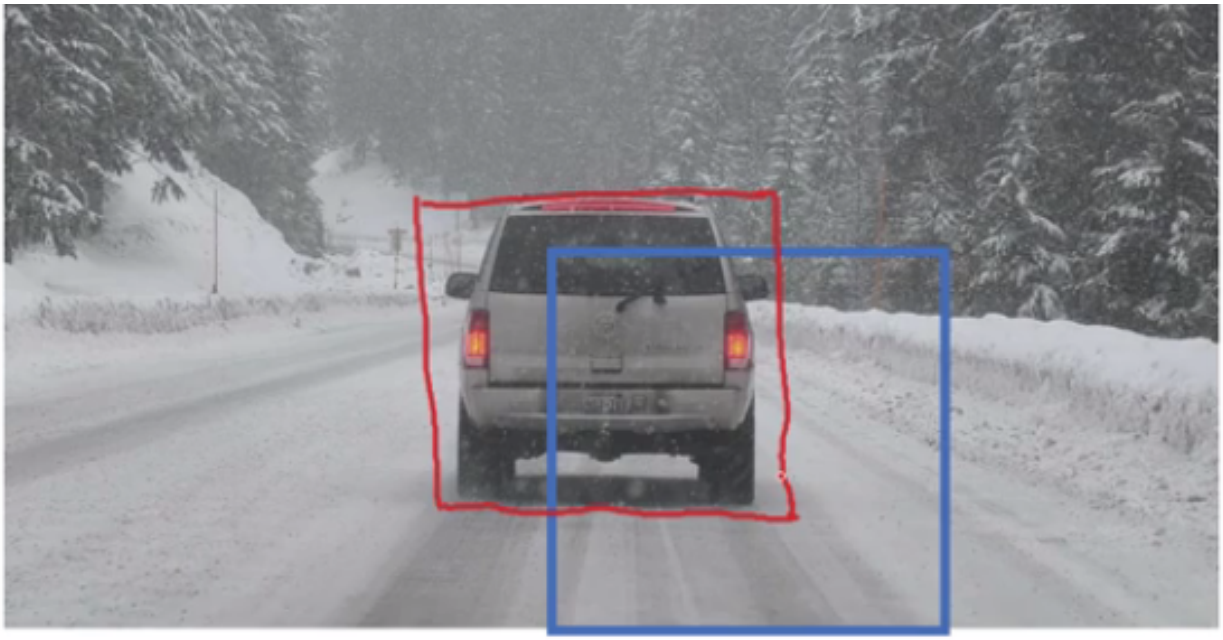


Figure 36: A failure example of convolutional sliding window algorithm.

The rectangle in red is what we want while the rectangle in blue is what the algorithm outputs.

4.4.5 Bounding Box Predictions

A better algorithm than the convolutional sliding windows in the last section is the YOLO algorithm [7].

YOLO algorithm (Figure 37):

- Lets say we have an image of 100×100.
- Place a 3×3 grid on the image. For more smother results you should use 19×19 for the 100×100.
- Apply the classification and localization algorithm we discussed in a previous section to each section of the grid. bx and by represent the center point of the object in each grid and will be relative to the box so the range is between 0 and 1 while bh and bw will represent the height and width of the object which can be greater than 1.0 but still a floating point value.
- Do everything at once with the convolution sliding window. If Y shape is 1×8 as we discussed before then the output of the 100×100 image should be $3 \times 3 \times 8$ which corresponds to 9 cell results.
- Merging the results using predicted localization mid point.

We have a problem if we have found more than one object in one grid box.

One of the best advantages that makes the YOLO algorithm popular is that it has a great speed and a Conv net implementation.

Figure 37: Example of YOLO algorithm.

How is YOLO different from other Object detectors? YOLO uses a single CNN network for both classification and localizing the object using bounding boxes.

In the next section we will see some ideas that can make the YOLO algorithm better.

4.4.6 Intersection Over Union

Intersection Over Union (IoU) is a function used to evaluate the object detection algorithm. It computes the size of intersection and divide it by the union. More generally, IoU is a measure of the overlap between two bounding boxes.

4.4.7 Non-Max Suppression

One of the problems we have addressed in YOLO is that it can detect an object multiple times. Non-max suppression is a way to make sure that YOLO detects the object just one.

For example 38:

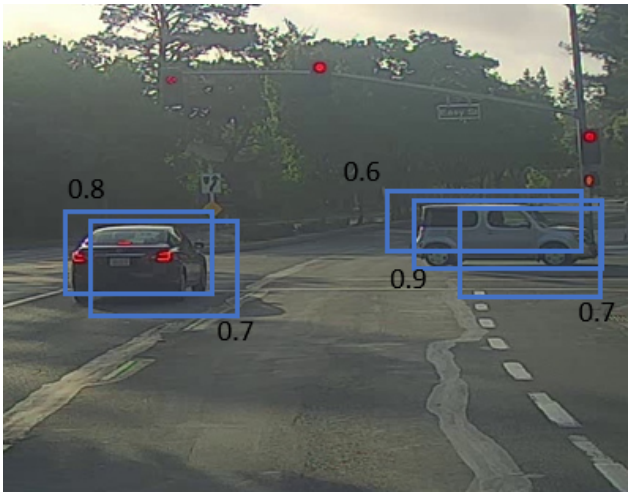


Figure 38: Non-max suppression.

Each car has two or more detections with different probabilities. This came from some of the grids that thinks that this is the center point of the object.

Non-max suppression algorithm:

- Lets assume that we are targeting one class as an output class.
- Y shape should be $[P_c, b_x, b_y, b_h, b_w]$. Where P_c is the probability if that object occurs.
- Discard all boxes with $P_c < 0.6$.
- While there are any remaining boxes:
  i. Pick the box with the largest P_c output that as a prediction.
  ii. Discard any remaining box with $IoU > 0.5$ with that box output in the previous step. i.e. any box with high overlap (greater than overlap threshold of 0.5).

If there are multiple classes types c you want to detect, you should run the non-max suppression c times, once for every output class.

4.4.8 Anchor Boxes

In YOLO, a grid only detects one object. What if a grid cell wants to detect multiple object?

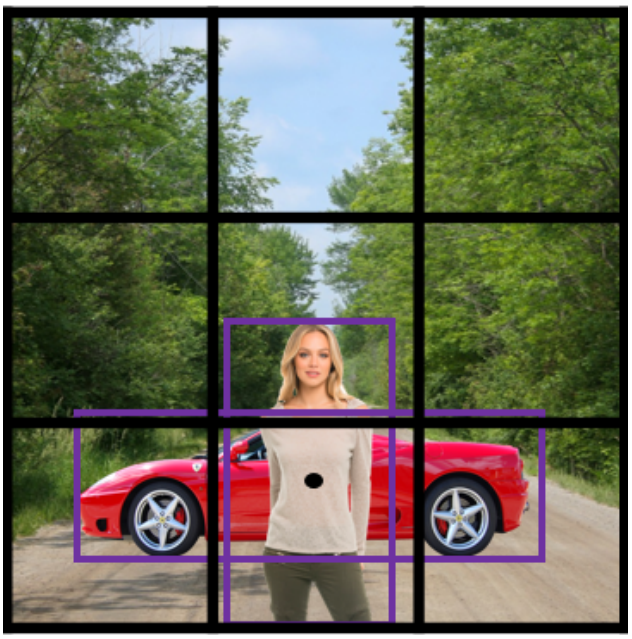


Figure 39: Example of anchor boxes.

As we can see from Figure 39, car and person grid is same here. In practice this actually happens rarely.

The idea of anchor boxes helps us solving this issue.

If $Y = [Pc, bx, by, bh, bw, c1, c2, c3]$. Then to use two anchor boxes like this:

- $Y = [Pc, bx, by, bh, bw, c1, c2, c3, Pc, bx, by, bh, bw, c1, c2, c3]$. We simply have repeated the one anchor Y.
- The two anchor boxes you choose should have known shape (Figure 40):

With no anchor box, each object in training image is assigned to grid cell that contains that object's midpoint. With two anchor boxes, each object in training image is assigned to grid cell that contains object's midpoint and anchor box for the grid cell with highest IoU. We suppose that every object detected is similar to an anchor box which has similar shape.

You may have two or more anchor boxes but you should know their shapes.

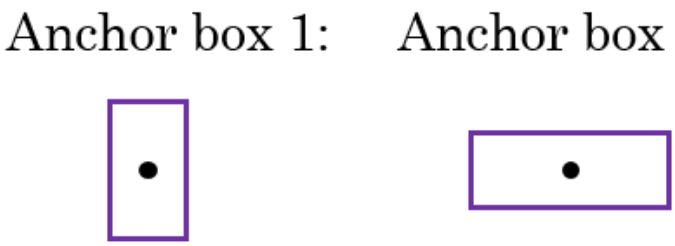


Figure 40: Visualization of two anchor boxes.

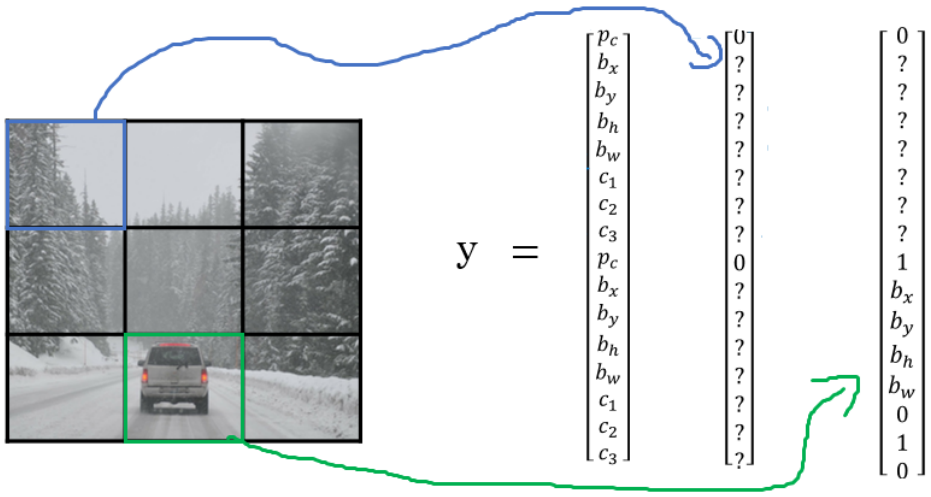


Figure 41: An extension example of YOLO Ny.

- How do you choose the anchor boxes and people used to just choose them by hand. Maybe five or ten anchor box shapes that spans a variety of shapes that cover the types of objects you seem to detect frequently.
- You may also use a k-means algorithm on your dataset to specify that.

Anchor boxes allows your algorithm to specialize, means in our case to easily detect wider images or taller ones.

4.4.9 YOLO Algorithm

Suppose we need to do object detection for our autonomous driving system. It needs to identify three classes: Pedestrain, Car and Motorcycle.

We decided to choose two anchor boxes, a tall one and a wide one. (Like we said in practice they use five or more anchor boxes hand made or generated using k-means.)

Our labeled Y shape will by $[Ny, HeightOfGrid, WidthOfGrid, 16]$, where Ny is the number of instances and each row is as follows:

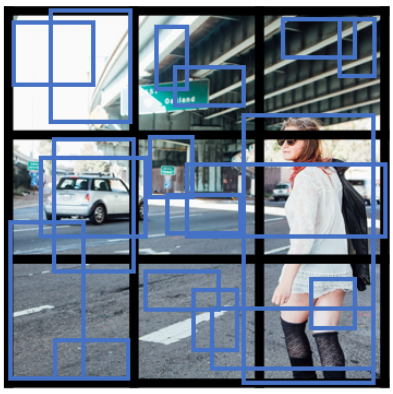
- $[Pc, bx, by, bh, bw, c1, c2, c3, Pc, bx, by, bh, bw, c1, c2, c3]$

Your dataset could be an image with a multiple labels and a rectangle for each label. For example (Figure 41):

- We first initialize all of them to zeors and ?, then for each label and rectangle choose its closest grid point then the shape to fill it and then the best anchor point based on the IoU, so that the shape of Y for one image should be $[HeightOfGrid, WidthOfGrid, 16]$.

Train the labeled images on a Conv net, you should receive an output of $[HeightOfGrid, WidthOfGrid, 16]$ of our case.

To make predictions, run the Conv net on an image and run non-max suppression algorithm for each class you have. In our case, there are 3 classes (Figure 42a). Total number of generated boxes are grid_width * grid_height * num_anchors = $3 \times 3 \times 2$. By removing the low probability predictions you should have (Figure 42b). Then get the best probability followed by the IoU filtering (Figure 42c). The details network set up could refer to Github.



(a) Predicted anchor boxes.

(b) Predicted anchor boxes (removing low probability predictions).

(c) Predicted anchor boxes (removing low probability predictions and using IoU filtering).

4.4.10 Region Proposals (R-CNN)

YOLO is fast but the downside of it is that it process a lot of areas where no objects are presented. R-CNN tries to pick a few windows and run a Conv net (your confident classifier) on top of them.

The algorithm R-CNN [8] uses to pick windows is called a segmentation algorithm. Outputs something like this (Figure 43):

If for example the segmentation algorithm produces 2000 blob then we should run our classifier/CNN on top of these blobs.

There has been a lot of work regarding R-CNN tries to make it faster:

- R-CNN[8]: Propose regions. Classify proposed regions one at a time. Output label + bounding box. Downside is that it's slow.
- Fast R-CNN[9]: Propose regions. Use convolution implementation of sliding windows to classify all the proposed regions.
- Faster R-CNN[10]: Use convolutional network to propose regions.
- Mask R-CNN[11].

Most of the implementations of faster R-CNN are still slower than YOLO. Andrew Ng thinks that the idea behind YOLO is better than R-CNN because you're able to do all the things in just one stage instead of two stages.

4.5 Face Recognition

This section and the next section are aimed at discovering how CNNs can be applied to multiple fields, including art generation and face recognition. Besides, try to implement you own algorithm to generate art and recognize faces. So first we will talke about face recognition.

4.5.1 What Is Face Recognition?

Face recognition system identifies a person's face. It can work on both images or videos.

Face verification vs. face recognition:

- Verification:
  - Input: image, name/ID. (1:1)

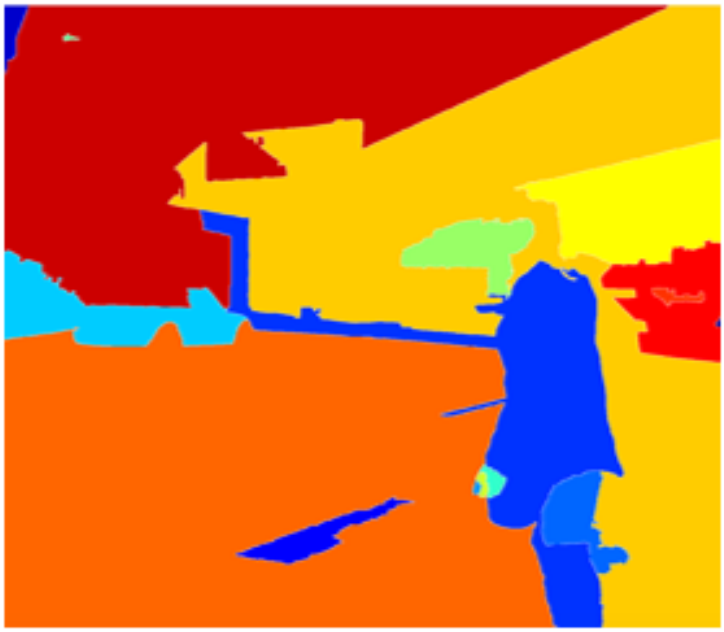


Figure 43: Segmentation result by using R-CNN to pick windows.

  - Output: whether the input image is that of the claimed person.
  - "Is this the claimed person?"
- Recognition:
  - Has a database of K persons and get an input image.
  - Output ID if the image is any of the K persons (or not recognized).
  - "Who is this person?"

We can use a face verification system to make a face recognition system. The accuracy of the verification system has to be high (around 99.9% or more) to be use accurately within a recognition system because the recognition system accuracy will be less than the verification system given K persons.

4.5.2 One-Shot Learning

One of the face recognition challenges is to solve one shot learning problem.

One Shot Learning: A recognition system is able to recognize a person, learning from one image.

Historically deep learning doesn't work well with a small number of data.

Instead to make this work, we will learn a **similarity function**:

- d(img1, img2) = degree of difference between images.
- We want d result to be low in case of the same faces.
- We use τ as a threshold for d: if d(img1, img2) $\leq \tau$, then the faces are the same.

Similarity function helps us solving the one shot learning. Also its robust to new inputs.

4.5.3 Siamese Network

We will implement the similarity function using a type of NNs called Siamese Network [1] in which we can pass multiple inputs to the two or more networks with the same architecture and parameters.

Siamese network architecture is as the following (Figure 44):

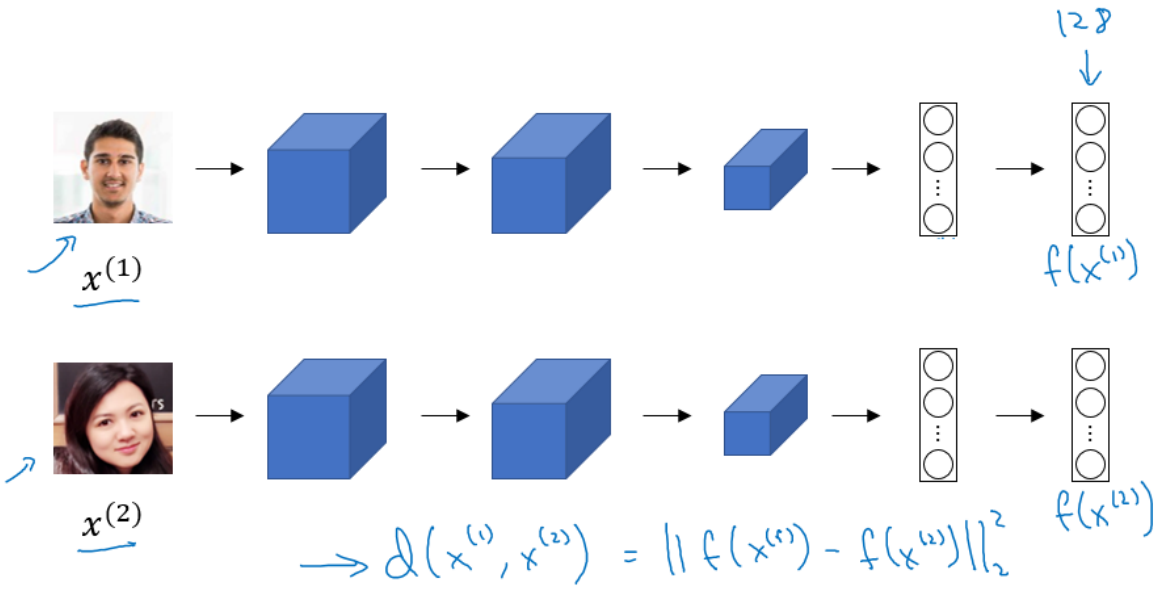


Figure 44: Siamese network.

We make 2 identical ConvNets which encoders an input image into a vector. In the above image the vector shape is (128,).

The loss function will be $d(x1, x2) = ||f(x1) - f(x2)||^2$.

If x1 and x2 are the same person, we want d to be low. If they are different persons, we want d to be high.

4.5.4 Triplet Loss

Triplet loss is one of the loss functions we can use to solve the similarity distance in a Siamese Network.

Our learning objective in the triplet loss function is to get the distance between an **anchor** image and a positive or a negative image (positive means same person, while negative means different person).

The triplet name came from that we are comparing an anchor A with a positive P and a negative N image.

Formally we want:

- Positive distance to be less than negative distance: $||f(A) - f(P)||^2 \leq ||f(A) - f(N)||^2$.
- Then $||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 \leq 0$.
- To make sure the NN won't get an output of zeros easily: $||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 \leq -\alpha$. α is a small number. Sometimes its called the margin.
- Then $||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha \leq 0$.

Final loss function:

- Given 3 images (A, P, N)
- $L(A, P, N) = \max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0)$.
- $J = \sum(L(A[i], P[i], N[i]))$ for all triplets of images.

You need multiple images of the same person in your dataset. Then get some triplets out of your dataset. Dataset should be big enough.

Choosing the triplets A, P, N:

- During training, if A, P, N are chosen randomly (subject to A and P are the same and A and N aren't the same), then one of the problems is the following constrain is easily satisfied ($d(A, P) + \alpha \leq d(A, N)$). So the NN won't learn much.

- What we want to do is to choose triplets that are hard to train on. So for all the triplets we want the above equation to be satisfied. Details are in FaceNet[12].

4.5.5 Face Verification and Binary Classification

Triplet loss is one way to learn the parameters of a ConvNet for face recognition. However, there's another way to learn these parameters as a straight binary classification problem.

Learning the similarity function another way[13] (Figure 45):

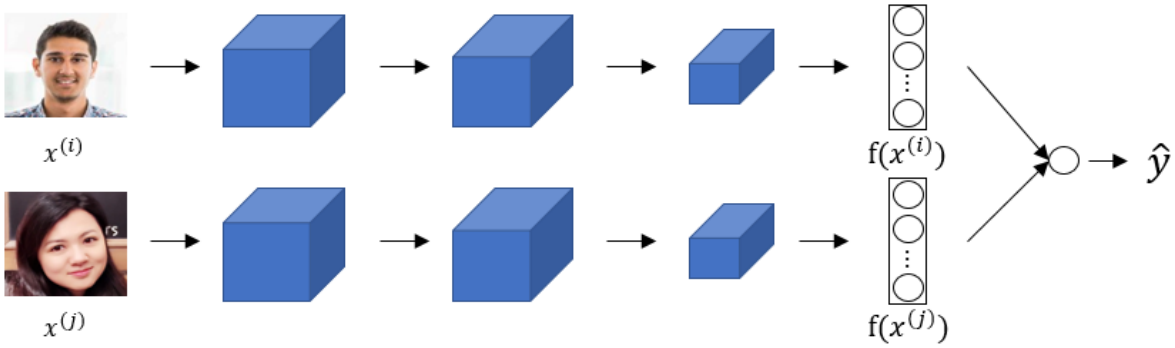


Figure 45: Learning the similarity function in one network.

- The final layer is a sigmoid layer.
- $\hat{Y} = w_{ij} sigmoid(f(x_i) - f(x_j)) + b$, where the substraction is the Manhattan distance between $f(x_i)$ and $(f(x_j)$.
- Some other similarities can be Euclidean distance similarity.

A good performance/deployment trick:

- Pre-compute all the images that you're using as a comparison to the vector $f(x_j)$.
- When a new image that needs to be compared, get its vector $f(x_i)$ then put it with all the pre-computed vectors and pass it to the sigmoid function.

This version works quite as well as the triplet loss function.

4.6 Neural Style Transfer

4.6.1 What Is Neural Style Transfer?

Neural style transfer takes a content image c and a style image s and generates the content image g with the style of style image. e.g. Figure 46.

In order to implement this you need to look at the features extracted by the ConvNet at the shallower and deeper layers.

It uses a previously trained convolutional network like VGG, and builds on top of that. The idea of using a network trained on a different task and applying it to a new task is called transfer learning.

4.6.2 What Is Deep ConvNets Learning?

Visualizing what a deep network is learning. Firstly, give this AlexNet like ConvNet (Figure 47).

Pick a unit in layer l. Find the nine image patches that maximize the unit's activation. Notice that a hidden unit in layer 1 will see relatively small portion of NN, so if you plotted it it will match a msall image in the shallower layers while it will get larger image in deeper layers.

Repeating for other units and layers.

It turns out that layer 1 is learning the low level representations like colors and edges.

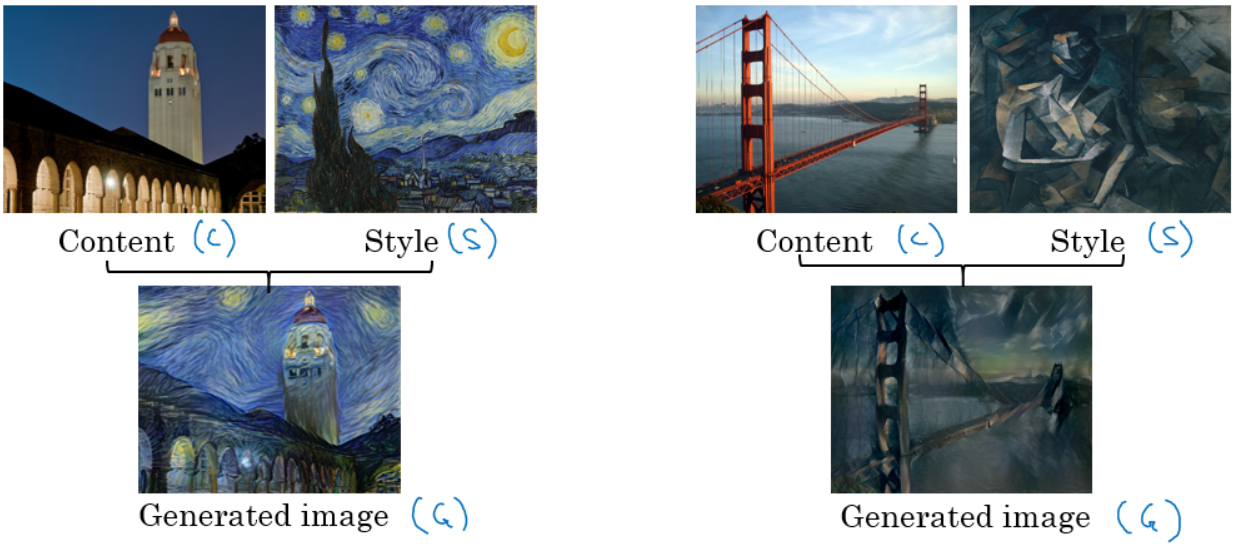


Figure 46: Examples of neural style transfer.

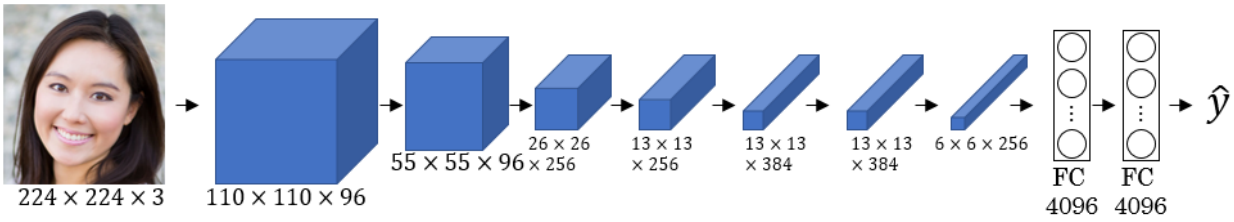


Figure 47: AlexNet like ConvNet.

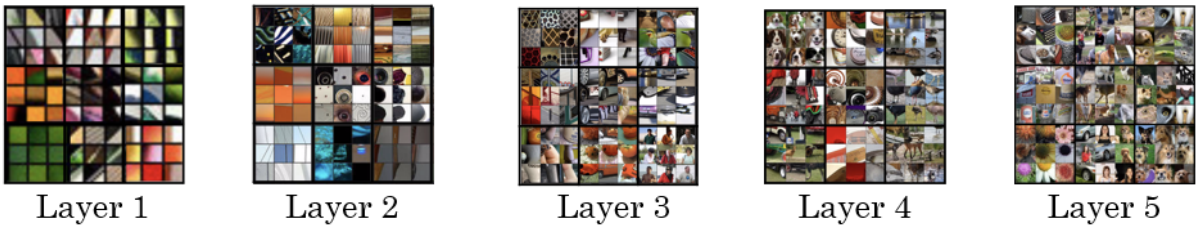


Figure 48: Visualization of ConvNets.

You will find out that deeper layers are learning more complex representations[14] (Figure 48).

A good explanation on how to get receptive field given a layer[3] (Figure 49):

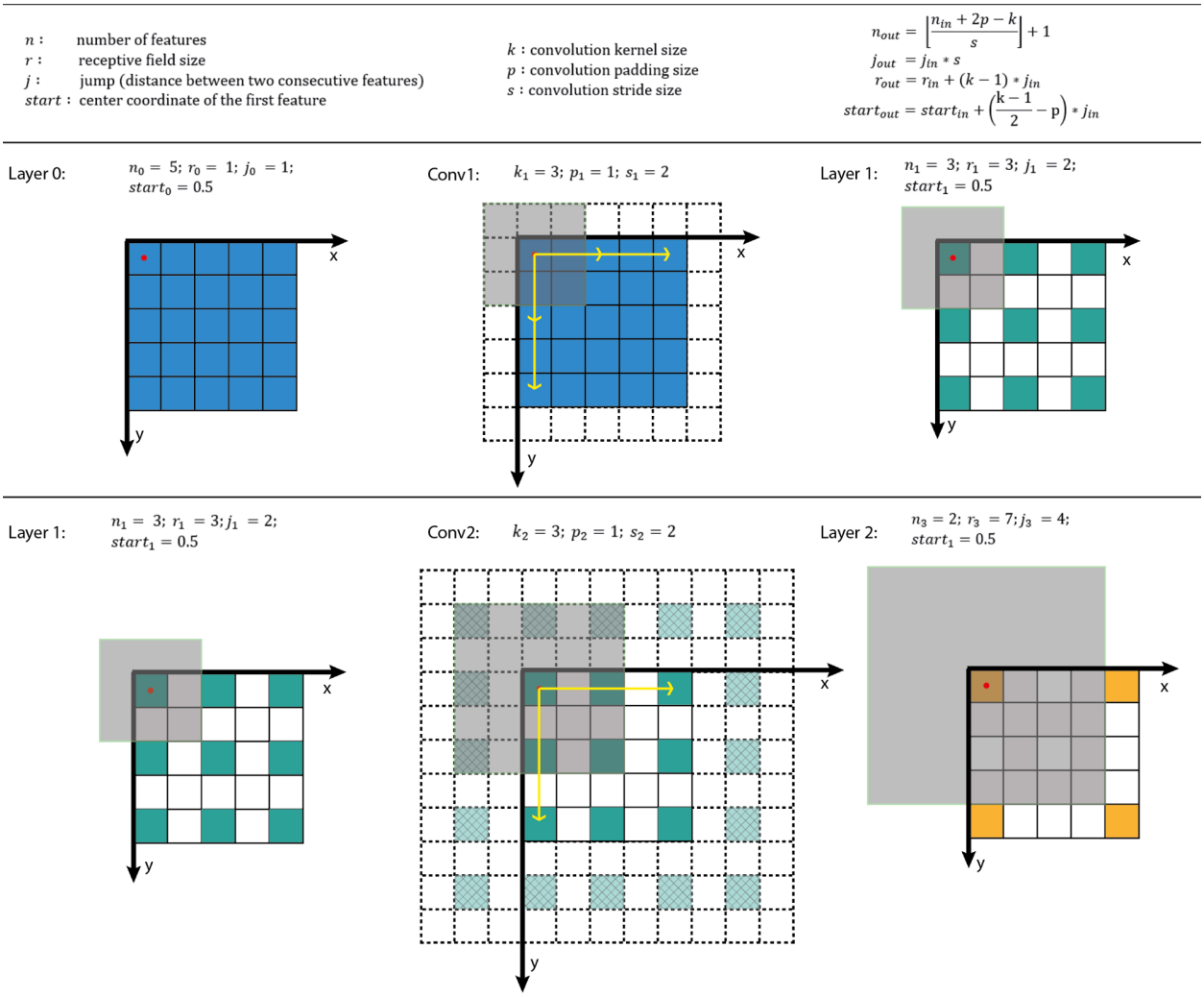


Figure 49: An explanation of how to get receptive field.

4.6.3 Cost Function

We will define a cost function for the generated image that measures how good it is.

Given a content image C, a style image S, and a generated image G:

- $J(G) = \alpha J(C, G) + \beta J(S, G)$
- $J(C, G)$ measures how similar is the generated image to the content image.
- $J(S, G)$ measures how similar is the generated image to the style image.
- α and β are relative weighting to the similarity and these are hyperparameters.

Find the generated image G:

i. Initialize G randomly.

[3] https://medium.com/mlreview/a-guide-to-receptive-field-arithmetic-for-convolutional-neural-networks-e0f514068807

ii. Use gradient descent to minimize $J(G)$. $G = G - dG$. We compute the gradient image and use gradient descent to minimize the cost function.

The iterations might be as following image:

To generate by using the following (Figure 50):

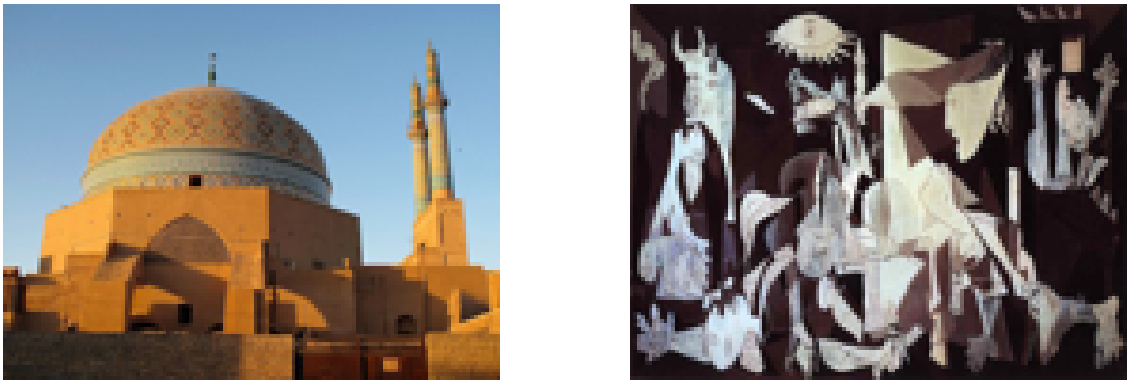


Figure 50: Provided content image C and style image G.

You will go through this (Figure 51):

Figure 51: Staged pictures of generated image G.

4.6.4 Content Cost Function

In the previous section we showed that we need a cost function for the content image and the style image to measure how similar is them to each other.

Say you use hidden layer l to compute content cost. If we choose l to be small (like layer 1), we will force the network to get similar output to the original content image. In practice l is not too shallow or not too deep but in the middle.

Use pre-trained ConvNet like VGG network.

Let $a(C)[l]$ and $a(G)[l]$ be the activation of layer l on the images.

If $a(C)[l]$ and $a(G)[l]$ are similar then they will have the same content:

$$J(C,G)atlayerl = \frac{1}{2}||a(C)[l] - a(G)[l]||^2 \tag{23}$$

4.6.5 Style Cost Function

Meaning of the **style** of an image:

- Say you're using layer l's activation to measure **style**.

- Define style as correlation between activations across channels. That means given an activation like Figure 52. How correlate is the orange channel with the yellow channel?

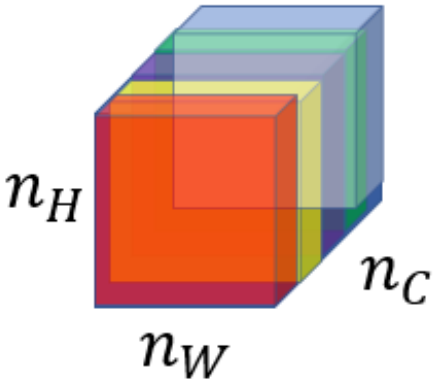


Figure 52: Illustration of image style.

- Correlated means if a value appeared in a specific channel a specific value will appear too (depend on each other). While uncorrelated means if a value appeared in a specific channel doesn't mean that the another value will appear (not depend on each other). The correlation tells you how a components might occur or not occur together in the same image.

The correlation of style image channels should appear in the generated image channels.

Style matrix (Gram matrix):

- Let $a(l)[i, j, k]$ be the activation at layer l with $i = H, j = W, k = C$.
- Also $G(l)(s)$ is the matrix of shape $n_C(l) \times n_C(l)$ (We call this matrix style matrix or Gram matrix). In this matrix, each cell will tell us how correlated is a channel to another channel.

$$
\begin{aligned}
G^{[l](S)}_{kk'} &= \sum_{i=1}^{n^l_H} \sum_{j=1}^{n^{[l}_W} a^{[l](S)}_{ijk} a^{[l](S)}_{ijk'} \\
G^{[l](G)}_{kk'} &= \sum_{i=1}^{n^l_H} \sum_{j=1}^{n^{[l}_W} a^{[l](G)}_{ijk} a^{[l](G)}_{ijk'}
\end{aligned}
\tag{24}
$$

Steps to be made if you want to create a tensorflow model for neural style transfer.

i. Create an Interactive Session.
ii. Load the content image, load the style image
iii. Randomly initialize the image to be generated.
iv. Load the VGG16 model
v. Build the TensorFlow graph: 1) Run the content image through the VGG16 model and compute the content cost; 2) Run the style image through the VGG16 model and compute the style cost; 3) Compute the total cost; 4) Define the optimizer and the learning rate.
vi. Initialize the TensorFlow graph and run it for a large number of iterations, updating the generated image at every step.

4.6.6 Keras Tutorial

To train and test a model in Keras there are four steps:

i. Create the model.

ii. Compile the model by calling *model.compile(optimizer = "...", loss = "...", metrics = ["accuracy"])*.

iii. Train the model on train data by calling *model.fit(x = ..., y = ..., epochs = ..., batch_size = ...)*. (We can add a validation set while training too.)

- Test the model on test data by calling *model.evaluate(x = ..., y = ...)*.

Summarize of steps in Keras: Create → Compile → Fit/Train → Evaluate/Test.

Model.summary() gives a lot of useful informations regarding your model including each layers inputs, outputs, and number of parameters at each layer.

To choose the Keras backend you should go to $HOME/.keras/keras.json and change the file to the desired backend like Theano or Tensorflow or whatever backend you want.

After you create the model you can run it in a tensorflow session without compiling, training, and testing capabilities.

You can save your model with model_save and load your model using model_load This will save your whole trained model to disk with the trained weights.

5 Sequence Models

This is the fifth course and last course of deep learning specialization at Coursera taught by Professor Andrew Ng. Here is my certificate after finishing this course (Fig. No, currently not earned).

5.1 Course Overview

According to the official site of this course, you will learning the following after you finish this course:

- Understand how to build and train Recurrent Neural Networks (RNNs), and commonly-used variants such at GRUs and LSTMs.
- Be able to apply sequence models to natural language problems, including text synthesis.
- Be able to apply sequence models to audio applications, including speech recognition and music synthesis.

5.2 Recurrent Neural Networks

In this section, you'll learn about recurrent neural network. This type of model has been proven to perform extremely well on temporal data. It has several variants including LSTMs, GRUs and Bidirectional RNNs, which you're going to learn about in this section.

5.2.1 Why Sequence Models

Sequence models like RNNs and LSTMs have greatly transformed learning on sequences in the past few years.

Examples of sequence data in applications:

- Speech recognition (*sequence to sequence*): X→ wave sequence; Y→ text sequence.
- Music generation (*one to sequence*): X→ nothing or an integer; Y→ wave sequence.
- Sentiment classification (*sequence to one*): X→ text sequence; Y→ integer rating from one to five.
- DNA sequence analysis (sequence to sequence): X→ DNA sequence; Y→ DNA labels.
- Machine translation (sequence to sequence): X→ text sequence (in one language); Y→ texe sequence (in other language).
- Video activity recognition (sequence to one): X→ video frames; Y→ label (activity).
- Name entity recognition (sequence to sequence): X→ text sequence; Y→ label sequence.

All of these problems with different input and output (sequence or not) can be addressed as supervised learning with label data X, Y as the training set.

The goal is given this representation for x to learn a mapping using a sequence model to then target output y as a supervised learning problem.

5.2.2 Recurrent Neural Network Model

Why not to use a standard network for sequence tasks? There are two problems (needs to be clarified):

- Inputs, outputs can be different lengths in different examples. This can be solved for normal NNs by paddings with the maximum lengths but its not a good solution.
- Doesn't share features learned across different positions of text/sequence. Using a feature sharing NN like in CNNs can significantly reduce the number of parameters in your model. That's what we will do in RNNs.

Recurrent neural network doesn't have either of the town mentioned problems.

Lets build a RNN that solves **name entity recognition** task (Figure 53):

We should note:

- In this problem, $T_x = T_y$. In other problems where thy aren't equal, the RNN architecture may be different.
- $a^{<0>}$ is usually initialized with zeros, but some others may initialize it randomly in some cases.

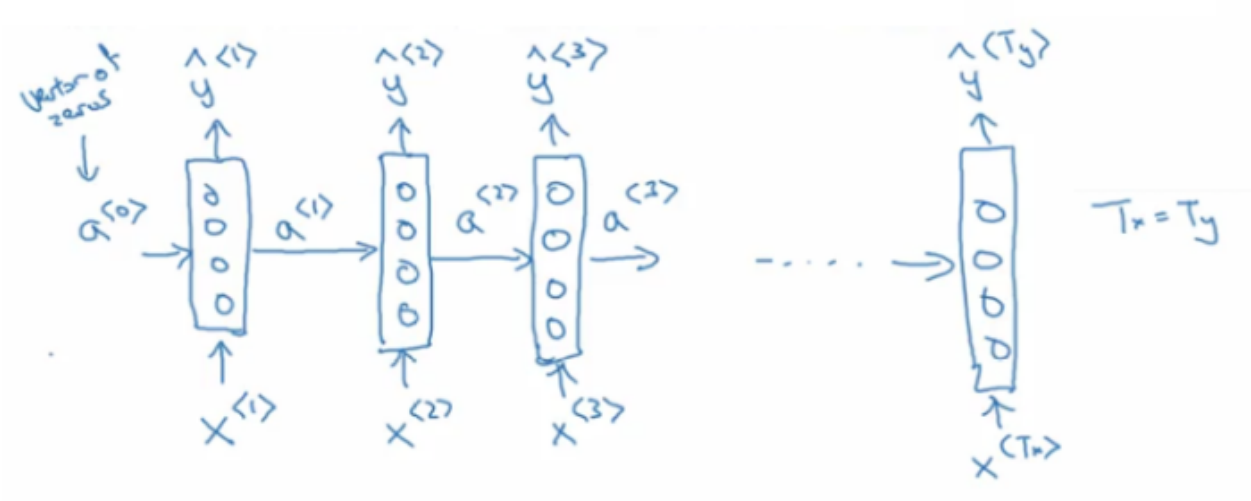


Figure 53: Example of name entity recognition task.

- There are three weight matrices here: W_{ax}, W_{aa} and W_{ya} with shapes: $W_{ax} \rightarrow (n_h, n_x), W_{aa} \rightarrow (n_h, n_h), W_{ya} \rightarrow (n_y, n_h)$.

The weight matrix W_{aa} is the memory the RNN is trying to maintain from the previous layers.

In the discussed RNN architecture, the current output $y^{<t>}$ depends on the previous inputs and activations.

Let's have this example 'He Said, "Teddy Roosevelt was a great president"'. In this example Teddy is a person name but we know that from the word *president* that came after Teddy not from *He* and *said* that were before it.

So limitation of the discussed architecture is that it can not learn from elements later in the sequence. To address this problem we will later discuss **Bidirectional RNN (BRNN)**.

Now let's discuss the forward propagation equations on the discussed architecture (Figure 54):

The activation function for a is usually $\tanh$ or $ReLU$ and for y depends on your task choosing some activation functions like $sigmoid$ and $softmax$. In name entity recognition task we will use sigmoid because we only have two classes.

In order to help us develop complex RNN architectures, the last equations needs to be simplified a bit (Figure 55):

w_a is w_{aa} and w_{ax} stacked horizontally. $[a^{<t-1>}, x^{<t>}]$ is $a^{<t-1>}$ and $x^{<t>}$ stacked vertically. w_a shape: $(n_h, n_h + n_x)$. $[a^{<t-1>}, x^{<t>}]$ shape: $(n_h + n_x, 1)$.

5.2.3 Backpropagation Through Time

Usually deep learning frameworks do backpropagation automatically for you. But it's useful to know it works in RNNs.

We will use the cross-entropy loss function:

$$L^{<t>}(\hat{y}^{<t>}, y^{<t>}) = -y^{<t>} \log \hat{y}^{<t>} - (1 - y^{<t>}) \log(1 - \hat{y}^{<t>}) \tag{25}$$

The total loss is:

$$L(\hat{y}, y) = \sum_{t=1}^{T_y} L^{<t>}(\hat{y}^{<t>}, y^{<t>}) \tag{26}$$

The backpropagation here is called backpropagation through time because we pass activation a from one sequence element to another like backwards in time.

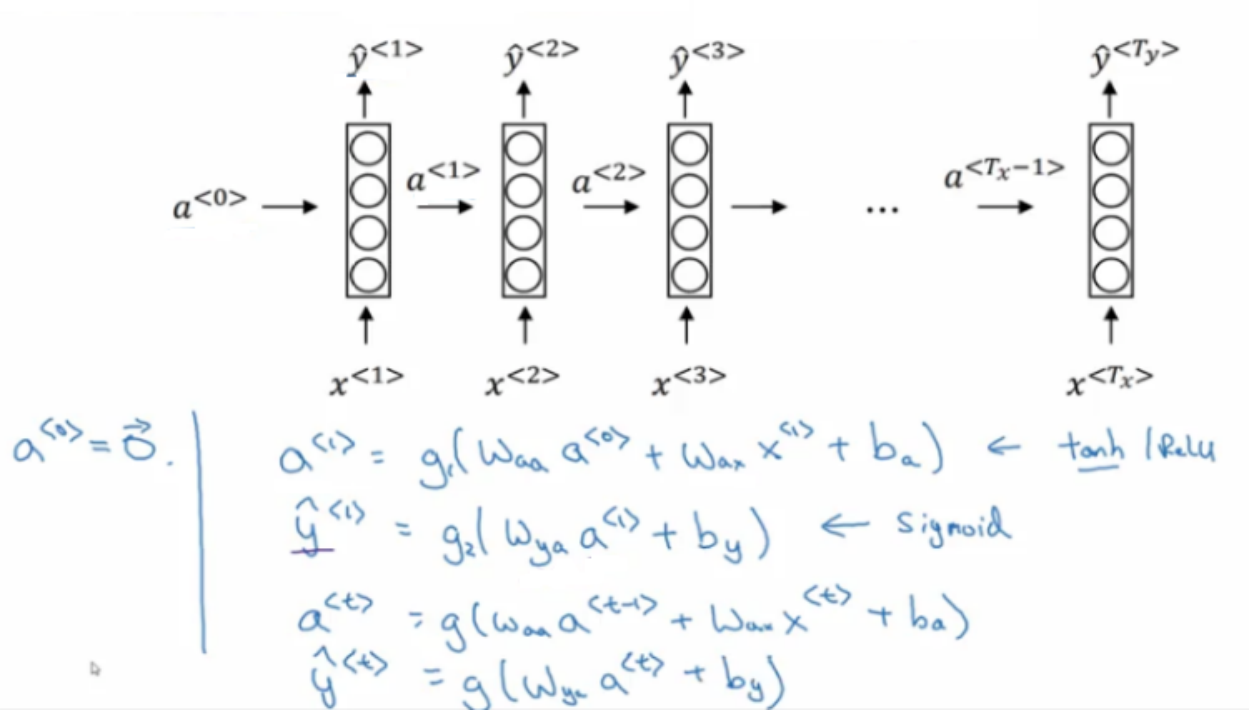


Figure 54: Forward propagation of a RNN.

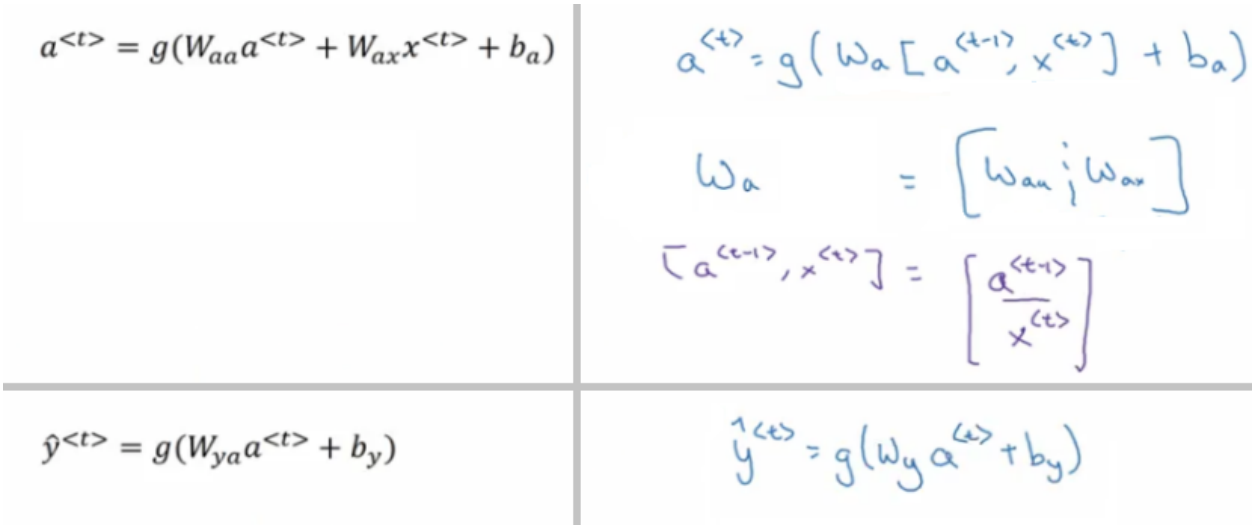


Figure 55: Simplified RNN notations.

5.2.4 Different Types of RNNs

Here are different types of RNNs (Figure 56, inspired by Andrej Karpathy's blog[4]):

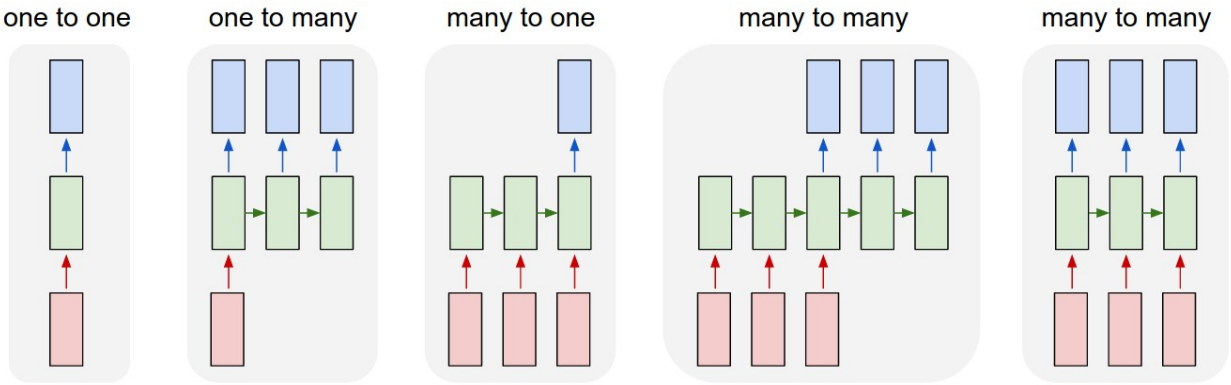


Figure 56: Different kinds of RNNs.

5.2.5 Language Model and Sequence Generation

What is a language model:

- Let's say we're solving a speech recognition problem and some says a sentence that can be interpreted into two sentences: a) The apple and *pair* salad; b) The apple and *pear* salad.
- *Pair* and *pear* sounds exactly the same, so how would a speech recognition application choose from the two.
- That's where the language model comes in. It gives a probability for the two sentences and the application decides the best based on this probability.

The job of a language model is to give a probability of any given sequence of words.

How to build language models with RNNs

- The first thing is to get a training set: a large corpus of target language text.
- Then tokenize this training set by getting the vocabulary and then one-hot each word.
- Put an end of sentence token <*EOS*> with the vocabulary and include it with each converted sentence. Also, use the token <*UNK*> for the unknown words.

Given the sentence "Cats average 15 hours of sleep a day. <EOS>"

In training time we will use this (Figure 57):

The loss function is defined by cross-entropy loss.

To use this model:

i. For predicting the chance of next word, we feed the sentence to the RNN and then get the final $y^{<t>}$ hot vector and sort it by maximum probability.

ii. For taking the probability of a sentence, we compute this: $p(y^{<1>}, y^{<2>}, y^{<3>}) = p(y^{<1>}) * p(y^{<2>}|y^{<1>}) * p(y^{<3>}|y^{<1>}, y^{<2>})$. This is simply feeding the sentence into the RNN and multiplying the probabilities (outputs).

5.2.6 Sampling Novel Sequences

After a sequence model is trained on a language model, to check what the model has learned you can apply it to sample novel sequence.

To be continued.

[4] http://karpathy.github.io/2015/05/21/rnn-effectiveness/

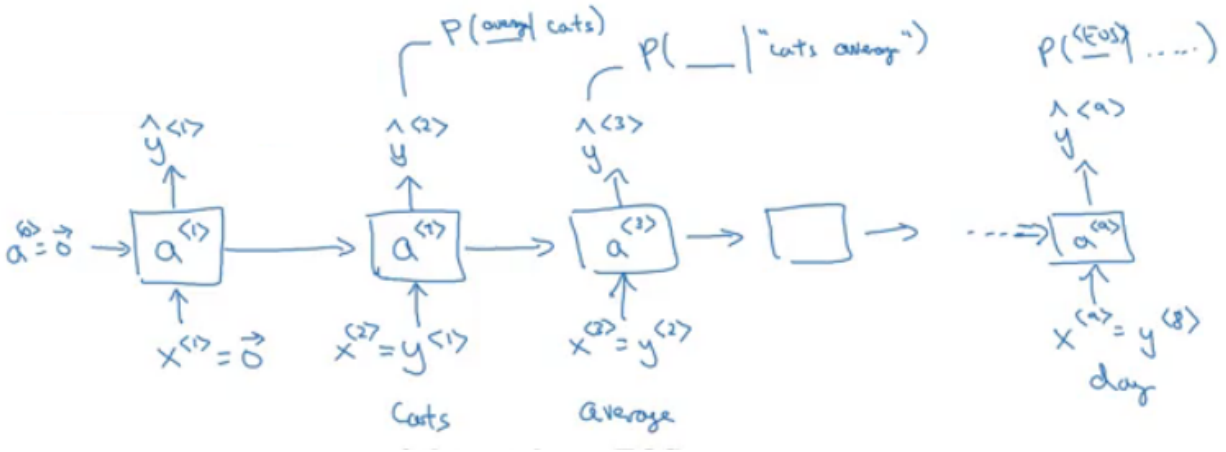


Figure 57: Example of language model.

Acknowledgements

This note is heavily based on Prof. Andrew Ng's course at Coursera. Thanks Andrew for taking this wonderful course, I've learnt a lot from that. Besides, thanks Coursera for providing me full-funded financial aids to finish all the five courses in the specialization. Moreover, parts of the note are borrowed from Mahmoud Badry's github repository[5]. It gives me a clear panorama of what is the most important.

References

[1] Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov. Siamese neural networks for one-shot image recognition. In *ICML deep learning workshop*, volume 2. Lille, 2015.

[2] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[4] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[5] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network. *arXiv preprint arXiv:1312.4400*, 2013.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[7] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.

[8] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 580–587, 2014.

[9] Ross Girshick. Fast r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 1440–1448, 2015.

[10] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, pages 91–99, 2015.

[11] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross Girshick. Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision*, pages 2961–2969, 2017.

[5] https://github.com/mbadry1/DeepLearning.ai-Summary

[12] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[13] Yaniv Taigman, Ming Yang, Marc'Aurelio Ranzato, and Lior Wolf. Deepface: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1701–1708, 2014.

[14] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.