Weekly Work Report

Hongzhi Liu

VISION@OUC

July 22, 2018

1 Research problem

During this period of week, I spend time studying deep learning courses and working about Faster R-CNN algorithm for URPC2018. Our team have difficulty in changing competition data sets into VOC format and training a model for contest.

2 Research approach

For deep learning, I watch videos and write down the issues which I think are much important for further research. As for URPC2018, I continue to learn about Faster R-CNN algorithm [1] and try to train the First model with VOC2007. Besides, our team have prepared competition data sets for training contest model. I will list details about weekly work in Tab. 1 below.

Table 1: Weekly work progress.

URPC2018	Finish training the first VOC2007 model and outcome is good. Finish changing the txt truth table into xml files which are used for training contest model. Finish putting confusing contest data sets in VOC2007 order from 000000.jpg to 017181.jpg.
Deep learning courses	Finish learning improving deep neural network course which is the second lesson.

3 Research progress

This week for deep learning course of Anandrew Ng, I know about how to set up train, dev and test sets. Besides, I can try to analyze bias and variance and know what things to do if we may have high bias and variance versus. Furthermore, I also understand how to apply different forms of regularization and dropout on neural network. Some tricks for speeding up the training of neural network as well. Last but not least, I learn a little about gradient checking.

3.1 Training Model and Test results

For URPC2018, I learn to train the first model with standard VOC2007 data set. First, I get the pre-trained models from internet and set them in the appointed folder. Then I modify python project codes about training model during which I solve a lot of bugs and errors that quite confused me. I get four relevant files and test them with photos. Fortunately, test outcomes with model is good that I achieve a little goal at the present stage as shown in Fig. 1.

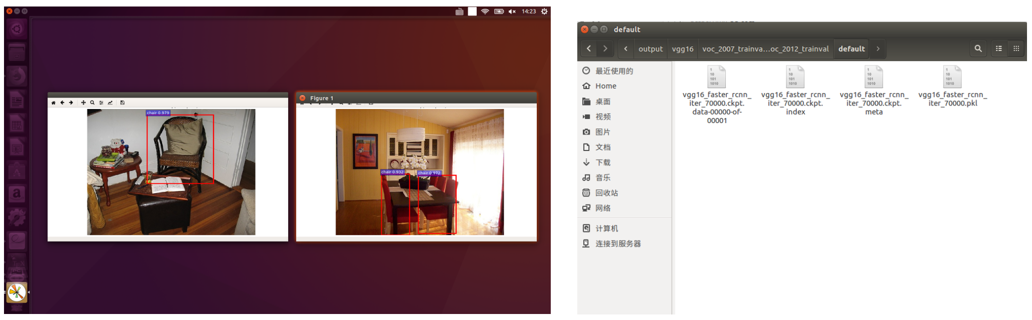
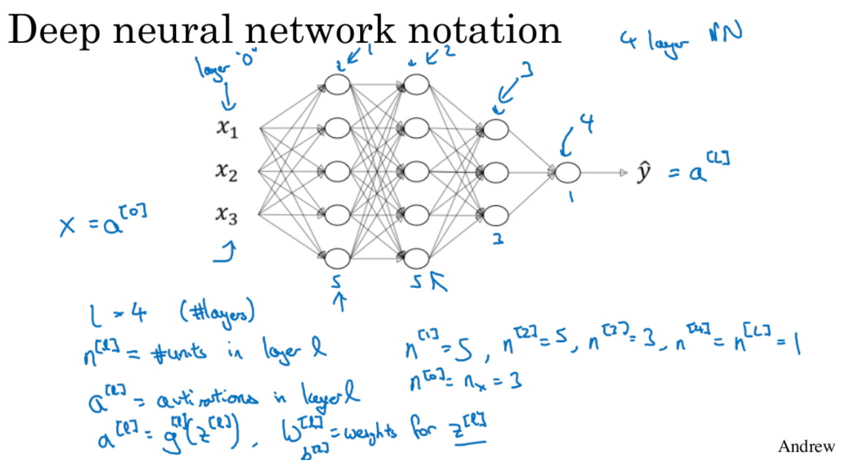


Figure 1
default
output
vgg16
voc_2007_trainva...oc_2012_trainval
default
最近使用的
Home
桌面
视频
图片
文档
下载
音乐
回收站
网络
计算机
连接到服务器
vgg16_faster_rcnn_iter_70000.ckpt.data-00000-of-00001
vgg16_faster_rcnn_iter_70000.ckpt.index
vgg16_faster_rcnn_iter_70000.ckpt.meta
vgg16_faster_rcnn_iter_70000.pkl

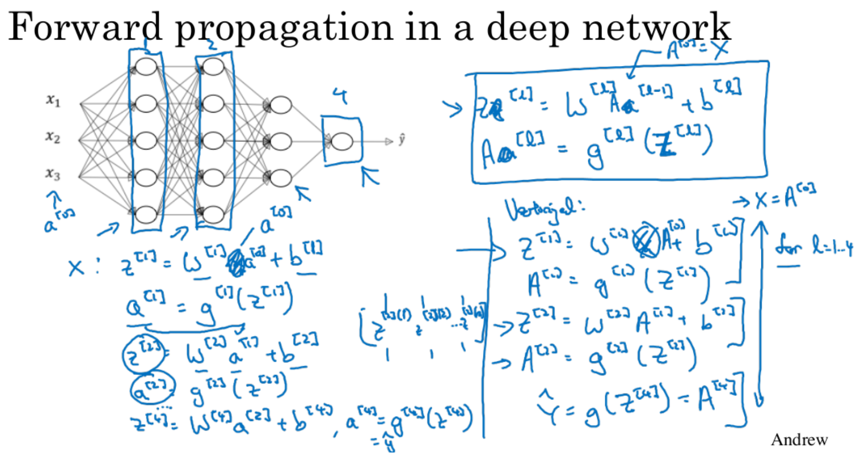
(a) Experimental results of VOC model (b) Model files

Figure 1: Training model and test results



Deep neural network notation
4 layer NN
layer "0"
x_1
x_2
x_3
$\hat{y} = a^{[L]}$
$X = a^{[0]}$
$L = 4$ (#layers)
$n^{[l]}$ = #units in layer l
$a^{[l]}$ = activations in layer l
$a^{[l]} = g^{[l]}(z^{[l]})$, $W^{[l]}$ = weights for $z^{[l]}$
$n^{[1]} = 5$, $n^{[2]} = 5$, $n^{[3]} = 3$, $n^{[4]} = n^{[L]} = 1$
$n^{[0]} = n_x = 3$
Andrew

Figure 2: Notation in deep neural network in which has four layers with three hidden layers totally.



Forward propagation in a deep network
$Z^{[l]} = W^{[l]} A^{[l-1]} + b^{[l]}$
$A^{[l]} = g^{[l]}(Z^{[l]})$
$A^{[0]} = X$
$x: z^{[1]} = W^{[1]} a^{[0]} + b^{[1]}$
$a^{[1]} = g^{[1]}(z^{[1]})$
$z^{[2]} = W^{[2]} a^{[1]} + b^{[2]}$
$a^{[2]} = g^{[2]}(z^{[2]})$
$z^{[4]} = W^{[4]} a^{[3]} + b^{[4]}$, $a^{[4]} = g^{[4]}(z^{[4]}) = \hat{y}$
Vectorized:
$X = A^{[0]}$
$Z^{[1]} = W^{[1]} A^{[0]} + b^{[1]}$
$A^{[1]} = g^{[1]}(Z^{[1]})$
$Z^{[2]} = W^{[2]} A^{[1]} + b^{[2]}$
$A^{[2]} = g^{[2]}(Z^{[2]})$
$\hat{Y} = g(Z^{[4]}) = A^{[4]}$
for $l = 1..4$
Andrew

Figure 3: Forward propagation in a deep network.

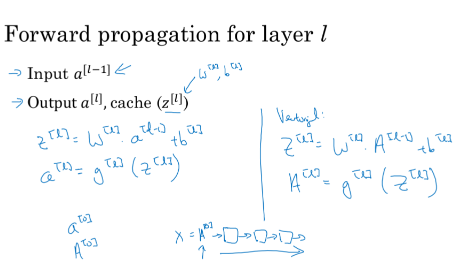
3.2 Deep L-layer neural network

In this secton, I go through the notation used to describe deep neural networks first. There is a four-layer neural network with three hidden layers as shown in Fig. 2, in which number of units in hidden layers are five, five and three. So we use L to denote the number of layers in the nerwork and $L = 4$ in this example. Besides, I use $n^{[l]}$ to denote the number of nodes or units in layer and $n^{[i]} = 5$ is the first hidden layer because we have 5 hidden units. So I can get that $n^{[4]} = n^{[l]} = 1$ which is the number of output units. The input layer is $n^{[0]} = n_x = 3$.

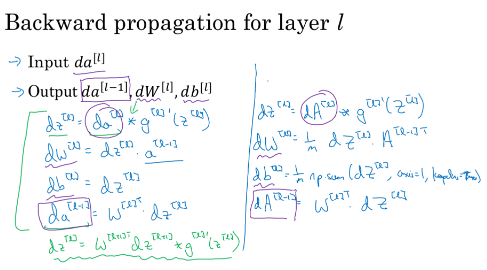
For each layer, we use $n^{[l]}$ to denote the activations in layer l and $w^{[l]}$ to compute the values $Z^{[l]}$ in the $a^{[l]}$ similatly the $b^{[l]}$ is. In summary, the input features are called x, which is alse the activstion of layer 0 that is $a^{[0]}$ as well and the activation of the final layer $a^{[L]} = \hat{y}$ which also means predicted output.

3.3 Forward and Backward Propagation

Then I learn how to perform for propagation in a deep network. As shown in Fig. 3, we can know the process and vectorized version on the left of diagram. $W^{[1]}$, which is a weight matrix, and $b^{[1]}$, which is the bias vector, are parameters affect the activations in layer 1 of the neural network. The activation function g depends on the layer. Besides, we can change x into $a^{[0]}$. And the general rule is that $Z^{[1]} = W^{[1]} \times A^{[l-1]} + b^{[l]}$ and $A^{[1]} = g^{[1]}(Z^{[1]})$ as shown in left diagram below, which gives the


Forward propagation for layer l
→ Input $a^{[l-1]}$
→ Output $a^{[l]}$, cache $(z^{[l]})$
$z^{[l]} = W^{[l]} \cdot a^{[l-1]} + b^{[l]}$
$a^{[l]} = g^{[l]}(z^{[l]})$
Vectorized:
$Z^{[l]} = W^{[l]} \cdot A^{[l-1]} + b^{[l]}$
$A^{[l]} = g^{[l]}(Z^{[l]})$

(a)


Backward propagation for layer l
→ Input $da^{[l]}$
→ Output $da^{[l-1]}, dW^{[l]}, db^{[l]}$
$dz^{[l]} = da^{[l]} * g^{[l]\prime}(z^{[l]})$
$dW^{[l]} = dz^{[l]} \cdot a^{[l-1]}$
$db^{[l]} = dz^{[l]}$
$da^{[l-1]} = W^{[l]T} \cdot dz^{[l]}$
$dz^{[l]} = W^{[l+1]T} dz^{[l+1]} * g^{[l]\prime}(z^{[l]})$
$dZ^{[l]} = dA^{[l]} * g^{[l]\prime}(Z^{[l]})$
$dW^{[l]} = \frac{1}{m} dZ^{[l]} \cdot A^{[l-1]T}$
$db^{[l]} = \frac{1}{m}$ np.sum$(dZ^{[l]}$, axis=1, keepdims=True)
$dA^{[l-1]} = W^{[l]T} \cdot dZ^{[l]}$

(b)

Figure 4: Forward and backward propagation

vectorized version of forward propagation. As we can see, the implementation of vectorization like a for loop, that is from 1 through the total number of layers and in neural network is perfectly okay to have an explicit for loop.

Then I know how to implement forward and backward propagation steps with basic blocks of a deep neural network in Fig. 4. The way to improve the forward funciton is $Z^{[l]} = W^{[l]} \times a^{[l-1]} + b^{[l]}$ and $a^{[l]} = g^{[l]} \times (Z^{[l]})$ which can be a vectorized implementation $Z^{[l]} = W^{[l]} \times A^{[l-1]} + b^{[l]}$ and $A^{[l]} = g^{[l]} \times (Z^{[l]})$ where the b being python broadcasting just as shown in Fig. 4(a). If we are processing the entire training set, $A^{[0]}$ is the entire training. So it is the input to the first forward function in the chain.

However, the backward propagation steps is shown in Fig. 4(b), which are $dZ^{[l]} = da^{[l]} \times g^{[l]\prime} \times (Z^{[l]})$, $dW^{[l]} = dZ^{[l]}$ and etc. In summary as Fig. 5, we may have the first and second layer maybe has a ReLU activation function but sigmoid activation function in the third layer and the output is $\hat{y}$ if we are doing binary classification. And then using $\hat{y}$ can compute the loss, this allows we to start backward iteration as backward propagation to compute the derivatives. We should know that $da^{[l]} = -\frac{y}{a} + \frac{1-y}{1-a}$ which turns out that the derivative of the loss function respect to the output.

3.4 Intuition about Deep Representation

If we are building a system for face recogniton or detection as shown in Fig. 6. Perhaps we input a picture of a face then the first layer of the neural network you can think of as maybe being a feature detector or an edge detector. In this example, we're plotting what a neural network with maybe 20 hidden units, might be kind of compute on this image. So the 20 hidden units visualize by these little square boxes. For example, this little visualization represents a hidden unit is trying to figure out if where the edges of that orientation in DMH. And maybe this hidden unit maybe trying to figure out where are the horizontal edges in this image. Now we think about where the edges in this picture by grouping together pixels to form edges. It can then detect the edges and group edges together to form parts of faces.

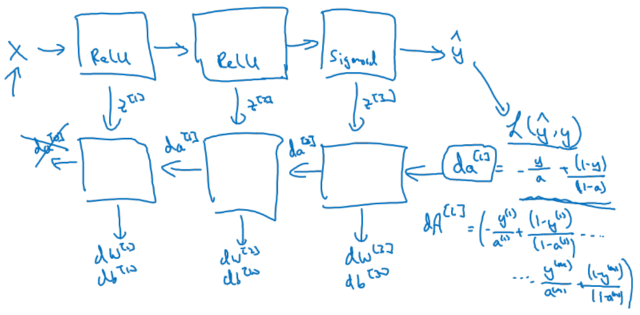

X
ReLU
ReLU
Sigmoid
$\hat{y}$
$z^{[1]}$
$z^{[2]}$
$z^{[3]}$
$\mathcal{L}(\hat{y}, y)$
$da^{[1]}$
$da^{[2]}$
$da^{[l]} = -\frac{y}{a} + \frac{(1-y)}{(1-a)}$
$dA^{[l]} = \left(-\frac{y^{(1)}}{a^{(1)}} + \frac{(1-y^{(1)})}{(1-a^{(1)})} \cdots \cdots -\frac{y^{(m)}}{a^{(m)}} + \frac{(1-y^{(m)})}{(1-a^{(m)})}\right)$
$dW^{[1]}$ $db^{[1]}$
$dW^{[2]}$ $db^{[2]}$
$dW^{[3]}$ $db^{[3]}$

Figure 5: Building blocks of deep neural networks.

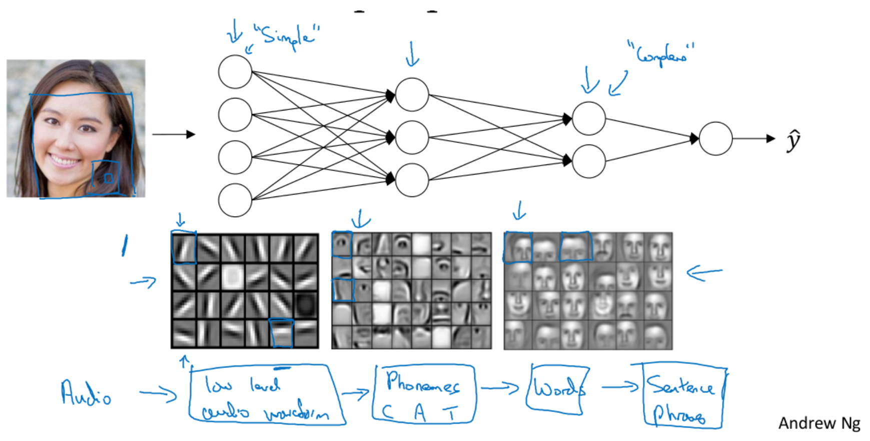


Figure 6: Face recogniton.

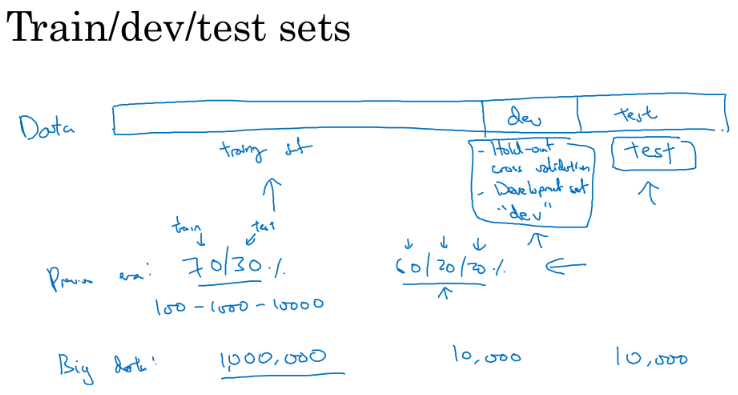


Figure 7: Train / Dev / Test sets.

3.5 Train / Dev / Test sets

From this section, I learn the practical aspects to make neural network work well, ranging from things like hyperparameter tuning to how to set up data and make sure optimization algorithm runs quickly, in order to get our learning algorithm to learn in a reasonable time.

Making good choices in how to set up training, development and test sets can make a huge different in helping us quickly find a good high performance neural network. So in practice applied machine learning is a highly iterative process. Even very experienced deep learning people find it almost impossible to correctly, and then applied deep leanring is a very iterative process where we just have to go around this cycle many times to hoprfully find a good choice of network for application.

Training data is carve off some portion of it to be training set. And some portion of it to be hold-out cross validation set which sometimes also called development set. Then some final portion of it to be test set as shown in Fig. 7. In order to get an unbiased estimate of how well algorithm is doing, it was common practice to take all data and split it according to maybe a 70/30% in terms of people often talk about 70% train and 30% test splits or maybe a 60% train, 20% dev, and 20% test which is widely considered best practice in machine learning. However, we also seen applications where if we have even more than a million examples, we might with 99.5% train, 0.25% and 0.25% test. Besides, the development and test sets should come from the same distribution. Furthermore, having set up a train dev and test set will allow to integrate more quickly, which allow to more efficiently measure the bias and variance of the algorithm.

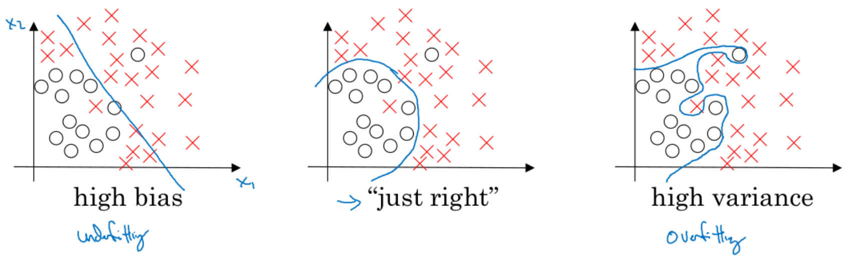


Figure 8: Bias and variance.

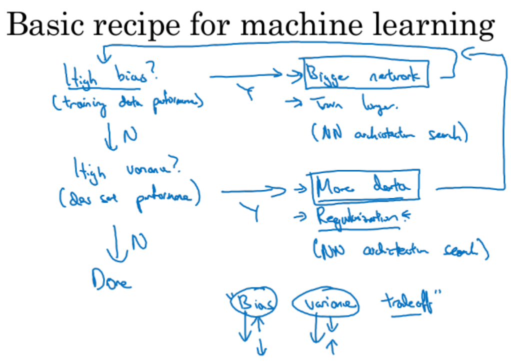


Figure 9: Basic recipe for machine learning.

3.6 Bias and Variance

Bias and Variance is one of those concepts that is easily learned but difficult to master. We see the data set that looks like this in Fig. 8. From class of a high bias, if we fit a straight line to the data, maybe get a logistic regression but is not a very good fit to the data, what we say that is underfitting the data. On the oppsite end, if we fit an incredibly complex classifier, here is a classifier of high variance and is overfitting the data. And there might be some classifier in between, with a medium level of complexity and fits it correctly, so we call that just right, it is somewhere in between.

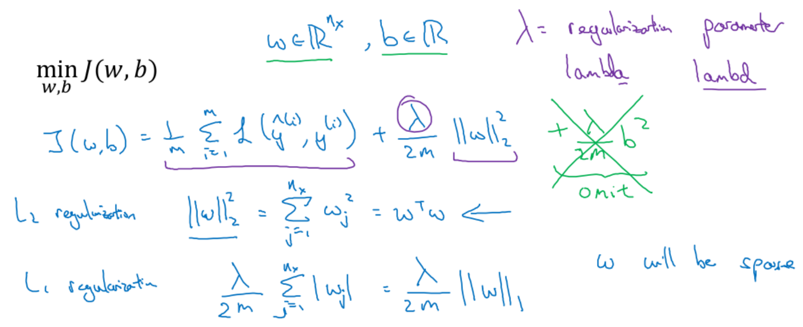
More generally, the optimal error called Bayes error. So the case of how to analyze bias and variance, the takeway is that by looking at your training set error. We can get a sense of how well you are fitting and looking at how much higher error goes. When you go from the training set to the dev set, that should give us a sense of how bad is the variance problem. Training and depth error can help us wether algorithm has a bias or variance problem or maybe both, which turns out that this information lets us much more systematically using what they call a basic recipe for machine learning.

When training a neural network, here a basic recipe in Fig. 9. After training a initial model, we should know how algorithm is high or not and so to try and evaluate if there is high bias. Besides, the training set or data performance. If it has high bias, even does not fit in the training set, we could try pick a network such as more hidden layers or more hidden units.

So when training a learning algorithm, we should try these things until at least get rid of the bias problem or fit the training set pretty well. And so to evaluate variance, we would look at dev set performance. If if it is high, the best way to solve a high variance problem is to ger more data. However, when we cannot get more, we could try regularization to reduce overfitting. So being clear on how much of a bias or variance problem or both can help us focus on selecting the most useful things to try.

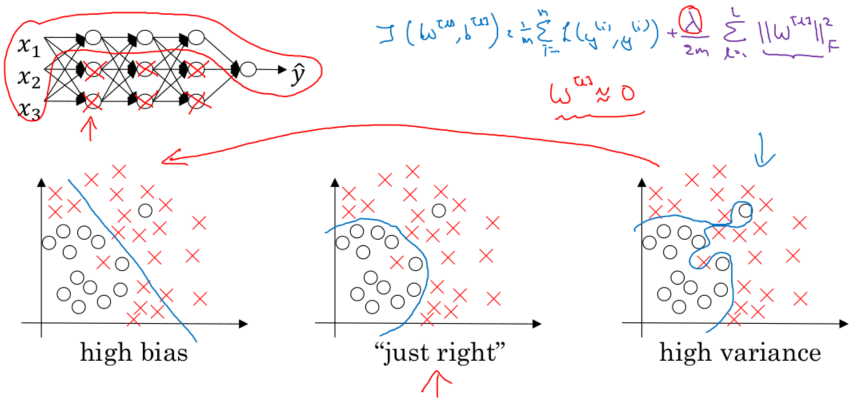
3.7 Regularization Reduces Overfitting

If we suspect neural network is overfitting your data, one of the first things we should try is probably regularization. But adding it will often help to prevent overfitting. The other way to address high



$\min_{w,b} J(w,b)$
$w \in \mathbb{R}^{n_x}, b \in \mathbb{R}$
λ = regularization parameter
lambda
lambd
$J(w,b) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\|w\|_2^2$
$+ \frac{\lambda}{2m} b^2$
omit
L2 regularization $\|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w$
L1 regularization $\frac{\lambda}{2m}\sum_{j=1}^{n_x}|w_j| = \frac{\lambda}{2m}\|w\|_1$
w will be sparse

Figure 10: Logistic regression.



x_1
x_2
x_3
$\hat{y}$
$J(w^{[l]}, b^{[l]}) = \frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L}\|w^{[l]}\|_F^2$
$w^{[l]} \approx 0$
high bias
"just right"
high variance

Figure 11: Examples of regularization help with overfitting.

variance is to get more training data which is alse quite reliable. We develop these prevent ideas using logistic regression. And so to add regularization to the logistic regression, what you do is add to it like $J(w, b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \parallel w \parallel_2^2$ where λ is the rregularization parameter and $\parallel w \parallel_2^2$ is norm of w squared is a square Euclidean norm of the parameter vector w as shown in Fig. 10, which is called L2 regularization. Because w is usually a high dimensional parameter vector, especially with a high variance problem, we just omit b. In a word, L2 regularization is the most common type of regularization which is called weight decay as well. We can also call it Frobenius norm of the matrix as $\parallel w^{[l]} \parallel_F^2 = \sum_{i=1}^{n^{[l-1]}}\sum_{j=1}^{n^{[l]}}(w_{ij}^{[l]})^2$.

We can know the reason why regularization help with overfitting from a couple example in Fig. 11. The cost function $J(w^{[l]}, b) = \frac{1}{m}\sum_{i=1}^{m}\mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2m} \parallel w \parallel_2^2$ which can penalize the weight matrices from being too large. The Frobenius morm might cause less overfiting because of one piece of intuition is that if you crank regularisation lambda to be really big, they will be incentivized to set the weigth matrices W to be reasonably close to zero that is setting the weight to be so close to zero for a lot of hidden units, which basically zeroing out a lot of the impact of these hidden units. If implementing regularization, we should remember J now has a new definition above. The new funciton is to debug gradient descent otherwise might not see J decrease monotonically on every single elevation.

In addition to L2 regularization, there is another powerful regularization techniques which is called dropout. With dropout, what we are going to do is to go through each of the layers of the network and set some probability of eliminating a node in neural network as shown in Fig. 12. So we end up with a much smaller, really much diminished network and then do back propagation training. Inverted dropout is one of ways of implementing dropout but if we are implementing dropout at test time, that just add noise to predictions. By spreading all the weights, this will tend to have an effect of shrinking the squared norm of the weights, and so, similiar to what we saw with L2 regularization, the effect of implementing dropout is that it shrinks the weights and does some of those outer regularization that helps prevent

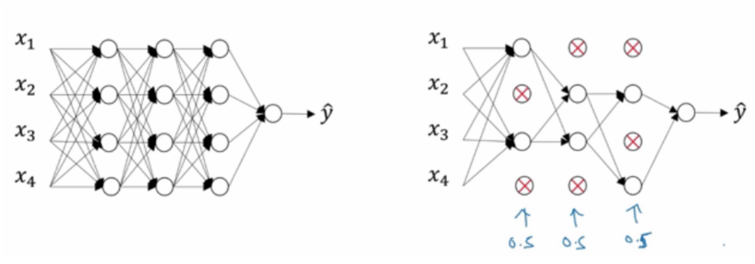


Figure 12: Dropout regularization technique.

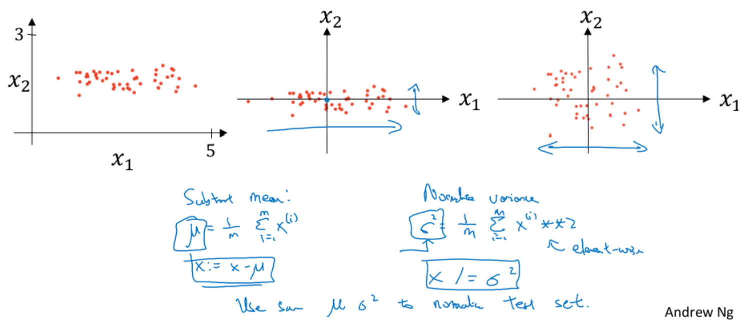

3
x_2
x_1
5
x_2
x_1
x_2
x_1
Subtract mean:
$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$
$x := x - \mu$
Normalize variance
$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} ** 2$
element-wise
$x /= \sigma^2$
Use same μ σ^2 to normalize test set.
Andrew Ng

Figure 13: Normalizing training sets.

overfitting. These could be different keep probs for different layers. Notice that the keep prob of one means that we are keeping every units. However, for layers where we are more worried about ever-fitting and layers with lots of parameters, we can set keep prob to be smaller to apply a more powerful form of dropout. If we are more worried about some layers overfitting than others, we can set lower keep prob for some layers than others. So in computer vision, the input size is so big, putting all these pixels that you almost never have enough data, so dropout is very frequently used by computer vision. However, ont big downside of dropout is that the cost function J is no longer well-defined. On every iteration, we are randomly killing off a bunch of nodes.

There are a few other techniques for reducing overfitting in your neural network, in addition to L2 regularization and dropout. If we are fitting a cat classifier, we can augment training set by taking image like flipping it horizontally or rotate. And double the size of training set. We can do this without needing to pay the expense of going out to take more pictures of cats as fake training examples.

When training a neural network, one of the techniques that will speed up our training is normalize inputs which corresponds to two steps as shown in Fig. 13. The first is to subtract out or to zero out the mean. And then the second step is to normalize the variances, which X1 and X2 are both equal to one. We use this to scale train data, then should use the same μ and σ^2 to normalize test set other than estimate μ and σ^2 separately on your training and test set. Because we want our data, both training and test examples to go through the same transformation defined by the same μ and σ^2. If input features came from very different scales then it is important to normalize your features.

3.8 Gradient Checking

One of the problems of training neural network, especially very deep neural network is data vanishing or exploding gradients. At the weights W, if they are all just a little bit bigger than 1 or identity matrix, then with a very deep network, the activations can explode. In contrast, if W is just a little bit less than identity, so the activations will decrease exponentially.

To prevent vanishing or exploding, we start with the example of initializing the weights for a single neuron in Fig. 14. In order to make z not blow up and become too small. One reasonable thing to do would be set the variance of $W^{[i]} = \frac{1}{n}$. So in practice, what you can do is set the weight matrix W for a

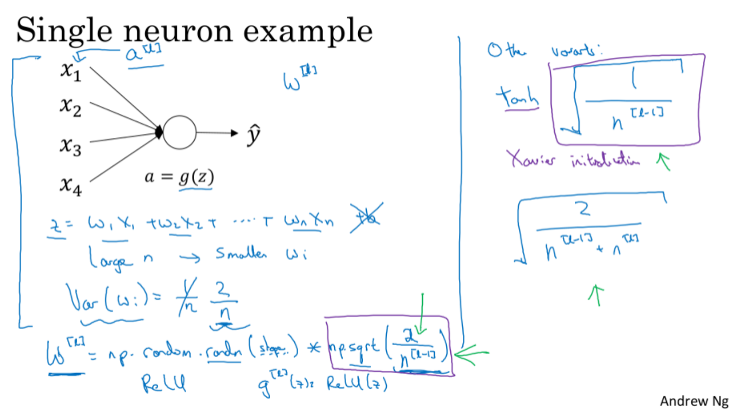

Single neuron example
x_1 x_2 x_3 x_4
$\hat{y}$
$a = g(z)$
$a^{[l-1]}$
$w^{[l]}$
$z = w_1x_1 + w_2x_2 + \cdots + w_nx_n$
Large $n \rightarrow$ Smaller w_i
$\text{Var}(w_i) = \frac{1}{n}\ \frac{2}{n}$
$W^{[l]} = \text{np.random.randn(shape)} * \text{np.sqrt}\left(\frac{2}{n^{[l-1]}}\right)$
ReLU $g^{[l]}(z) = \text{ReLU}(z)$
Other variants:
tanh $\sqrt{\frac{1}{n^{[l-1]}}}$
Xavier initialization
$\sqrt{\frac{2}{n^{[l-1]} + n^{[l]}}}$
Andrew Ng

Figure 14: Single neuron example.

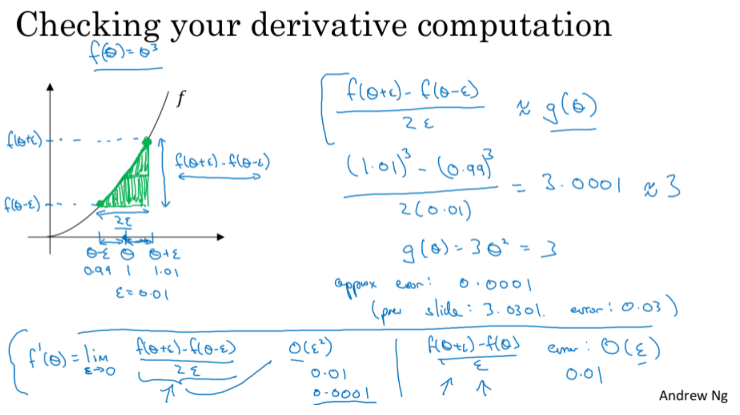

Checking your derivative computation
$f(\theta) = \theta^3$
f
$f(\theta+\epsilon)$
$f(\theta-\epsilon)$
$f(\theta+\epsilon) - f(\theta-\epsilon)$
2ϵ
$\theta-\epsilon$ θ $\theta+\epsilon$
0.99 1 1.01
$\epsilon = 0.01$
$\frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon} \approx g(\theta)$
$\frac{(1.01)^3 - (0.99)^3}{2(0.01)} = 3.0001 \approx 3$
$g(\theta) = 3\theta^2 = 3$
approx error: 0.0001
(prev slide: 3.0301. error: 0.03)
$f'(\theta) = \lim_{\epsilon \to 0} \frac{f(\theta+\epsilon) - f(\theta-\epsilon)}{2\epsilon}$
$O(\epsilon^2)$ 0.01 0.0001
$\frac{f(\theta+\epsilon) - f(\theta)}{\epsilon}$ error: $O(\epsilon)$ 0.01
Andrew Ng

Figure 15: Checking derivative computation.

certain layer $W^{[l]} = np.random.randn(shape) \times np.sqrt(\frac{1}{n^{[l-1]}})$ because that is the number of units that we are fitting in to each of the units of layer l. And it definitely helps reduce the vanishing, exploding gradients problem because it is trying to set each of the weight matrices w.

When implement back propagation, we will find that there's a test called creating checking that can really help make sure that implementation of back prop is correct. The first stage is to numerically approximate computations of gradients in Fig. 15 from which demonstrates two sided difference way of approximating the derivative is extremely close to 3. And so this gives you a much greater confidence that $g(\theta)$ is a correct implementation of the derivative of f. So when doing gradient checking, we rather use this two-sided difference when you compute $g(\theta) = \frac{f(\theta+\epsilon)-f(\theta-\epsilon)}{2\epsilon}$. And the takeway is that this teo-sided difference formula is much more accurate.

Then we should know that function J is a giant parameter. Whatever's the dimension of this giant parameter vector θ. So to implement grad check, what we are going to do is implements a loop just as equations $d\theta_{appear}[i] \approx \frac{\partial J}{\partial \theta_i}$ in Fig. 16. Then check $\parallel d\theta_{appear} - d\theta \parallel_2$ with no square on top which si Euclidean distance and then to normalize by the lengths of these vectors. So we implement this in practice, ϵ maybe 10^{-7} then value of formular equals to 10^{-7} or smaller, then that's great. We use it to track down whether or not some of your derivative computations might be incorrect. And after some amounts of debugging, it finally ends up being a very small value which turns out a correct implementation. When inplement neural network, we often implement foreprop and backprop and debug for a while, we find if passes grad check with a small value. Note that, grad check is used to debug rather than train.

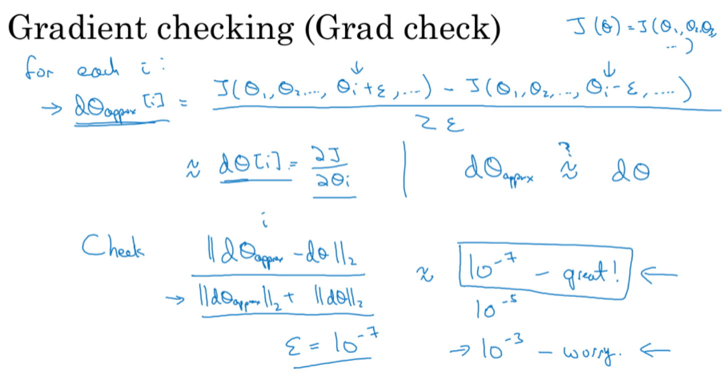

Gradient checking (Grad check)
$J(\theta) = J(\theta_1, \theta_2, \theta_3, \ldots)$
for each i:
$d\theta_{approx}[i] = \frac{J(\theta_1, \theta_2, \ldots, \theta_i + \varepsilon, \ldots) - J(\theta_1, \theta_2, \ldots, \theta_i - \varepsilon, \ldots)}{2\varepsilon}$
$\approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$
$d\theta_{approx} \overset{?}{\approx} d\theta$
Check $\frac{\|d\theta_{approx} - d\theta\|_2}{\|d\theta_{approx}\|_2 + \|d\theta\|_2}$
$\approx 10^{-7}$ - great!
10^{-5}
10^{-3} - worry.
$\varepsilon = 10^{-7}$

Figure 16: Checking derivative computation.

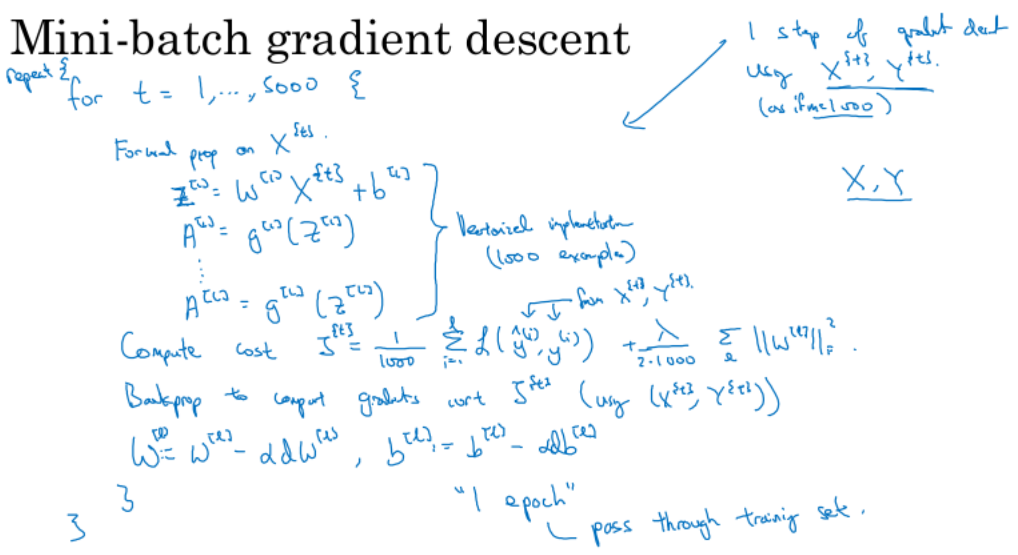

Mini-batch gradient descent
1 step of gradient descent using $X^{\{t\}}, Y^{\{t\}}$. (as if m=1000)
repeat {
for t = 1, ..., 5000 {
Forward prop on $X^{\{t\}}$.
$Z^{[1]} = W^{[1]} X^{\{t\}} + b^{[1]}$
$A^{[1]} = g^{[1]}(Z^{[1]})$
$A^{[L]} = g^{[L]}(Z^{[L]})$
Vectorized implementation (1000 examples)
X, Y
from $X^{\{t\}}, Y^{\{t\}}$.
Compute cost $J^{\{t\}} = \frac{1}{1000} \sum_{i=1}^{l} \mathcal{L}(\hat{y}^{(i)}, y^{(i)}) + \frac{\lambda}{2 \cdot 1000} \sum_{l} \|W^{[l]}\|_F^2$.
Backprop to compute gradients wrt $J^{\{t\}}$ (using $(X^{\{t\}}, Y^{\{t\}})$)
$W^{[l]} := W^{[l]} - \alpha dW^{[l]}$, $b^{[l]} := b^{[l]} - \alpha db^{[l]}$
}
}
"1 epoch"
pass through training set.

Figure 17: Mini-batch gradient descent.

3.9 Mini-batch Gradient Descent

Optimizating algorithms helps to train models quickly because training on a large data set is just slow. From courses before, I can know that vectorization allows to process all examples relatively fast but if they are very large then it can still be slow. What we should do is to split up training set into smaller and little baby training sets which are called mini-batches. Batch gradient descent refers to viewing that as processing entire batch of training samples all ai the same time and which we orocess is single mini batch rather than processing entire train set the same time. We get a new notation $X^{\{t\}}$ as shown in Fig. 17. If we have 5 million train samples total, each of these little mini batches has a thousand examples, we would have 5000 of these mini batches, so it ends with $X^{\{5000\}}$ and similarly to Y. Mini batch number t is comprised of $X^{\{t\}}$ and $Y^{\{t\}}$ that is a thousand samples with the corresponding input and output pairs. In summary, $X^{(i)}$ is the i training sample, [l] of $X^{[l]}$ is to index the different layers of the neural network.

The process is one pass through training set using mini-batch gradient descent. The code is also called doing one epoch of training and epoch is a word that means a single pass through the train set. With mini-batch gradient descent, a single pass through the training set, allows to take 5000 gradient descent steps.

On mini batch gradient descent though, if plot progress on cost function, it may not decrease on every iteration. In particular, on every iteration we are processing some $X^{\{t\}}$ and $Y^{\{t\}}$. If we plot the cost funciton $J^{\{t\}}$, we more likely to see lines trend downwards but is going to be a little bit noisier.

4 Progress in this week

This week for deep learning course of Anandrew Ng, I know about how to set up train, dev and test sets. Besides, I can try to analyze bias and variance and know what things to do if we may have high bias and variance versus. Furthermore, I also understand how to apply different forms of regularization and dropout on neural network. Some tricks for speeding up the training of neural network as well. Last but not least, I learn a little about gradient checking. As for URPC2018, I learn to train the first model with standard VOC2007 data set and test it with photos. Fortunately, test outcomes with model is good that I achieve a little goal at the present stage as shown in table blow.

Step 1 Finish training the first VOC2007 model and getting good outcome.

Step 2 Finish changing the txt truth table into xml files which are used for training contest model.

Step 3 Finish putting confusing contest data sets in VOC2007 order.

Step 4 Finish learning improving deep neural network course which is the second lesson.

5 Plan

Objective:	Finish training a model with contest data sets for URPC2018
Deadline:	2018.07.24

2018.07.16—2018.07.22 Finish neural networks and Deep Learning.

2018.07.23—2018.07.29 Finish improving deep neural networks courses.

2018.07.30—2018.08.05 Finish structuring machine learning projects courses.

2018.08.06—2018.08.12 Finish convolutional neural networks courses.

2018.08.13—2018.08.19 Finish sequence models courses.

References

[1] S. Ren, K. He, R. Girshick, and J. Sun. Faster R-CNN: Towards real-time object detection with region proposal networks. In *NIPS*, 2015. 1