Deep Learning

F. Song

1 Multinomial Logistic Classifier

1. Input X, normalize Inputs, so that the input has zero mean and unit variance. $\mu(X_i) = 0, \sigma(X_i) = \sigma(X_j)$.which makes your cost function more round, rather than elliptical, and easier and faster to optimize.
   $\mu = \frac{1}{m}\sum X^{(i)}; X = X - \mu\sigma^2 = \frac{1}{m}\sum x^{(i)} **2(elementwisesquaring); X = X/\sigma^2$
2. logit/score: $y = wX + b$, w is called the weight, b is called the bias
3. Softmax: $S(y_i) = \frac{e^{y_i}}{\sum e^{y_i}}$
4. Using one-hot encoding to represent labels L
5. cross entropy method to measure distance between S and L. $D(S, L) = -\sum_i (L_i log S_i)$

To find w and b, one thing we can do is to use the average of the cross entropy as loss :

$$l = \frac{1}{N}\sum_i D(S(wx_i + b), L_i) \tag{1}$$

Then we can use gradient descent to find w and b. To initialize the optimization problem, draw the weights randomly from an Gaussian distribution with $\mu = 0, \sigma$, because of softmax on top of it, small σ means your S is very uncertain, so we start with small σ, and let the estimation becomes more confident as the training progress. But gradient descent is very difficult to scale. A rule of thumb, if computing your loss takes n floating point operation,computing its gradient takes about three times that compute. So instead, we will use stochastic gradient descent.

Instead of computing the loss, we will compute an estimate of it, a very bad estimate, which is simply computing the average loss for a very small random fraction of the training data.(**Random** is very important here, if the way you pick your sample is not random enough, it no longer works at all.). Take a very small fraction of the data, computer the average loss of the sample, compute the derivative for that sample, and pretend that the derivative is the right direction to use to do gradient descent. Although it is not at all the right direction. We will compensate this by doing this many many time, by taking very small

steps each time. Doing this vastly more efficient than doing gradient descent. Stochastic gradient descent scales well with both data and model size.

Important tricks for SGD:

1. Make your inputs have 0 mean, and equal variance
2. Initialize with random weights, 0 mean, small variance: To help exploding or exponential decay of gradient or activation function. If the activation function is a Relu function, make $Var(w^L) = \frac{2}{n^{L-1}}$; $w^{[L]} = np.random.rand(shape) * np.sqrt(f\frac{2}{n^{[L-1]}})$
3. Momentum: take advantage of the knowledge from previous steps about where we should be headed. A cheap way to do that is to keep a running average of the gradients$M = 0.9M + \delta L$ and to use that running average instead of the direction of the current batch of the data $\alpha \Delta L(w_1, w_2)$.
4. Learning rate decay: lowering α over time.

2 AdaGrad

Adaptive Gradient descent is a modification of SGD which implicitly does momentum and learning rate decay for you.

3 Regulation

Deep learning models have so much flexibility and capacity that over fitting can be a serious problem, if the training set is large. Regulation is necessary to reduce variance.

3.1 L2 regulation

L1 regulation to make your model sparse(a lot of parameters become zero) helps only a little bit, Thus for the purpose of compressing the model, L2 regulation is used much more often than L1 regulation.

$$J(w^{[1]}, b^{[1]}, ...) = \frac{1}{m}\sum_{i=1}^{m} l(\hat{y(i)}, y^{(i)}) + \frac{\lambda}{2m}\sum_{l=1}^{L} ||w^{[l]}||^2 \tag{2}$$

where $||w^{[l]}||^2 = \sum_{i=1}^{n^{l-1}} \sum_{j=1}^{n^{[l]}} (w_{ij}^{[l]})^2$ is called the "Frobenius norm", which is the sum of the squares of the elements of a matrix

The update rule for gradient descent becomes:

$$w^{[l]} := w^{[l} - \alpha dw^{[l]} + \frac{\lambda}{m} w^{[l]} = w^{[l]} - \frac{\alpha\lambda}{m} w^{[L]} - \alpha dw^{[l]} (frombackprop)$$

Thus, L2 regulation is also called "weight decay". L2 regulation relies on the assumption that a model with small weights is simpler than a model with large

weights. Thus, by penalizing the square values of the weights in the cost function you drive all the weights to smaller values. This leads to a smoother model in which the output changes more slowly as the input changes.

3.2 Dropout Regulation

Dropout is a regularization techniques that is specific to deep learning. It randomly shuts down some neurons in each iteration. At each iteration, you shut down each neuron of a layer with probability $1 - keep_prob$ or keep it with probability $keep_prob$. The drop out neurons doesn't contribute to the training. The idea behind drop out is that at each iteration, you train a different model that uses only a subset of your neurons. With dropout, your neurons thus become less sensitive to the activation of one other specific neuron, because that other neuron might be shut down at any time. By adding the "inverted dropout " step, it ensures that the expected value of $A[l]$ still remains the same.

Drop out needs to be applied during forward and backward propagation. but not during testing time.

3.3 Data Augmentation

In order to obtain more training data to help with overfitting, we can augment the training set by fliping the image, taking random distortions and transformation, which is an inexpensive to get more data and sort of regularize your model.

4 Speeding up optimization

4.1 Normalizing training set

If the input features have different scale, the cost function will be elongated and less symmetric, which implies an slow learning. By normalizing the training set, we are making the cost function more symmetric, thus speed up learning.

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}; x = x - \mu : \sigma^2 = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} **2; x = x/\sigma^2$$

Remember to use the same μ, σ^2 to normalize the test set.

4.2 Vanishing/exploding gradients

Very deep networks suffering from vanishing/exploding gradients, a partial solution to this problem is initialize the weight carefully.

As more and more features are added to compute Z, to avoid the explosion of Z, we would like to have w small. One thing we can do in practice is

set$Var(w) = \frac{1}{n}$, where n is the number of inputs. It turns out for relu activation, $\frac{2}{n}$ works better(which is called the *He initialization*).

$$w^{[l]} = np.random.rand(shape) * np.sqrt(\frac{2}{n^{[l-1]}})$$

4.3 gradient checking

Take $W^{[l]}, b^{[l]}$..... and reshape into a big vector θ, take$dW^{[l]}, db^{[l]}$..... then reshape into a big vector $d\theta$

$$J(w^{[l]}, b^{[l]}...) = J(\theta)$$

To check whether $d\theta$ is the gradient of $J(\theta)$, to do gradient checking:

$$\mathrm{d}\theta_{appro}[i] = \frac{J(\theta_1,\theta_2,...\theta_i+\epsilon,...)-J(\theta_1,\theta_2,...\theta_i-\epsilon,...)}{2\epsilon} \approx d\theta[i] = \frac{\partial J}{\partial \theta_i}$$

Then check

$$\frac{||d\theta_{approx} - d\theta||_2}{||d\theta_{approx}||_2 + ||d\theta||_2} \approx 10^{-7}$$

1. Don't use gradient checking at training, only to debug
2. if algorithm fails grad check, look at the components to try to identify bug
3. Remember regularization
4. Doesn't work with dropout

5 Hyper-parameter tuning

1. α
2. β for momentum,number of hidden units, mini batch size
3. number of layers, learning rate decay

Strategies:

1. If searching among a large number of hyper parameters, you should use random sampling rather than than grid search. The reason is that it's difficult to know in advance which hyper-parameters are the more important ones,by sampling randomly, you are more richly determining which are the more important hyper parameters. Another technique you can use is coarse to fine search.
2. Use an appropriate scale to pick hyper parameters.

6 Batch Normalization

Batch Normalization makes your hyper parameter search problem much easier and your neural network more robust. We know that normalizing the input features can speed up learning, since it turns the contours of your learning problem from something that might be very elongated to something that is more round, which is easier for an algorithm to optimize. Batch normalization normalize an hidden layer. Although there are some debates in the literature about whether you should normalize the value before the activation function, so Z_n or after applying the activation function, so a_n. In practice, normalizing $Z - 2$ is done much more often. Implementing Batch Norm: Given some intermediate values in certain hidden layer $z^{[}n](1), z^{[}n](2)...$ To normalize $z^{[n]}$

$$\mu = \frac{1}{m}\sum_i z^{(i)} \tag{3}$$

$$\sigma^2 = \frac{1}{m}\sum_i (z_i - \mu)^2 \tag{4}$$

$$z^{(i)}_{norm} = \frac{z^{(i)-\mu}}{\sqrt{\sigma^2 + \epsilon}} \tag{5}$$

$$\tilde{z^{(i)}} = \gamma z^{(i)_{norm}} + \beta \tag{6}$$

7 ML strategy

7.1 use single number evaluation metric to evaluate the performance of your algorithm

Precision:Of examples recognized as cat, what % actually are cats? Recall: What % of actual cats are correctly recognized. FI score:

$$F1score = \frac{1}{\frac{1}{P} + \frac{1}{R}}, \tag{7}$$

which is the so called "harmonic mean" of precision and recall. Train
dev
test distributions Setting up the training, dev and test sets has a huge impact on productivity. It is important to choose the dev and test sets from the same distribution and it must be taken randomly from all the data. Guideline

(a) Choose a dev set and test set to reflect data you expect to get in the future.

(b) :things you can try to reduce bias:training a bigger network, training longer, use a better optimization algorithms(add momentum, add RMSprop, use Adam...), NN architecture/hyper-parameters search

(c) Things you can try to reduce variance: use regulation(L2, dropout, data augmentation), more data, NN architecture/hyper-parameters search

8 Convolutions Neural Network

volume convolutions: apply each of the filter **to the whole volume**, stack the convolution result of each of the filter together to get output Z. Then apply the activation to get A. Convolution layer contains an activation.

pooling: slides an (f,f) window over the input **at each layer**, and stores the max(max pooling) or ave(average pooling) value of the window in the output

9 ResNets

The goal in ResNet is to have a deep network where the gradient descent will still work after many layers. Typically, the more layers, the less effective the gradient descent gets, so that's something we have to fight against. The solution to achieving a deep network that avoids "vanishing gradient descent" is to include layers that don't have any effect on the gradient descent. That's what the identify block is for. The identify block helps estimate an identify function(a function whereby the output is exactly the same as the input). It turns out figuring out an identify function H(x) is actually hard. Think about it, you'd have to calibrate all the weights and biases so that H(x) =x. A work around is to do what you're doing in this assignment: process input X the regular way(convolve, batch)normalize, activate with Relu . We often call this output F(x), Then when you're done add the original unprocessed value of X to F(x). This results is something close to the identity function.

10 YOLO

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

For each bounding box of each grid cell, it's described by an vector $(P_c, b_x, b_y, b_h, b_w, c_1, c_2, c_3, ...)$.

3. P_c is the probability that there is an object in the cell.It is called the confidence probability

4. b_x x coordinate of bounding box center inside cell([0:1] with respect to grid cell size)

5. b_y y coordinate of bounding box center inside cell([0:1] with respect to grid cell size)

6. b_h is the bounding box height([0:1] with respect to grid cell size)

7. b_w is the bounding box width([0:1] with respect to grid cell size)

8. c_1 is the conditional probability of object belongs to class i, if an object is present in the box. $P_c * c_m$ is called the class score

   Summary for yolo:

   (a) Say Input image (608, 608, 3)

   (b) The input image goes through a CNN, resulting in a (19,19,5,85) dimensional output.

   (c) After flattening the last two dimensions, the output is a volume of shape (19, 19, 425), Each cell in a 19x19 grid over the input image gives 425 numbers. 425 = 5 x 85 because each cell contains predictions for 5 boxes, corresponding to 5 anchor boxes.85 = 5 + 80 where 5 is because (pc,bx,by,bh,bw) has 5 numbers, and and 80 is the number of classes we'd like to detect

   (d) For each grid, you then select only few boxes based on Score-thresholding, throw away boxes that have detected a class with a score less than the threshold

   (e) Even after filtering by thresholding over the classes scores, you still end up a lot of overlapping boxes. A second filter for selecting the right boxes is called non-maximum suppression (NMS).In Non-max suppression: Compute the Intersection over Union and avoid selecting overlapping boxes This gives you YOLO's final output.

   The key steps for Non-maximum suppression are:

   (a) Select the box that has the highest score.

   (b) Compute its overlap with all other boxes, and remove boxes that overlap it more than iou threshold.

   (c) Go back to step 1 and iterate until there's no more boxes with a lower score than the current selected box.