Review Questions 3

Group 1
Chuan Su
Diego Alonso Guillen Rosaperez

November 26, 2019

1. If all the initial weights have identical value, despite this value was randomly selected, all the initial neurons will have identical influence on the cost, which will lead to identical gradients. Thus, both neurons will evolve symmetrically throughout training, effectively preventing different neurons from learning different things. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same.Furthermore, a too large initial value will lead to exploding gradients (too large), while a too small one will lead to vanishing ones. With Xavier initialization, a normal distribution is used with mean $\mu = 0$ and variance $\sigma^2 = \frac{1}{n^{l-1}}$, where n^{l-1} is the number of neurons in layer $l-1$. He initialization parts from the same principle, but also considers that the activation function is ReLU using as variance $\sigma^2 = \frac{2}{n_l}$. In this sense, the non-linearity of the ReLU functions are considered.

   It is ok and common to initialize the biases to be zero, since the asymmetry breaking is provided by the small random numbers in the weights

2. (a) **ReLU** : It avoids and rectifies vanishing gradient problem and it is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. Nevertheless, it can only be used in the hidden layers of a neural network model. Additionally, some gradients and neurons can "die" sharply since it only considers positive values. Also, it can "blow up" the activations since it doesn't have an upper limit $[0, inf)$.

   (b) **Leaky ReLU**: Similar to ReLU, but it allows a small, non-zero, constant gradient α. In this way, it attempts to fix the "dying neurons" problem. Since it posses linearity, it can't be used for complex classification problems.

   (c) **ELU**: It is very similar to ReLU on the non-negative side, but on the negative side, ELU can produce negative outputs which smooth slowly. The negative values converge to a $-\beta$ constant.

   (d) **Sigmoid or Logistic** : it is especially used for models for binary classification problems where we have to predict the probability as an output.Since probability of anything exists only between the range of 0 and 1. However logistic activation function may cause vanishing gradients problem when inputs become large (negative or positive), the function saturates at 0 or 1, with a derivative extremely close to 0. Thus when backpropagation kicks in, it has virtually no gradient to propagate back through the network.

(e) **Tanh**: Its output is not linear, and have some similar properties as the sigmoid, but tanh make stepper gradients. It also have the banishing problem.

(f) **Softmax**: It calculates the probabilities distribution of the event over 'n' different elements, which makes it suitable for the output layer. It will calculate the probability of each target class over all possible target classes, which is helpful for determining the target class for the given inputs.

3. Batch normalization is a technique to address the problem that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. Bach normalization consists of adding an operation in the model just before the activation function of each layer, simply zero-centering and normalizing the inputs, then scaling and shifting the result using two new parameters per layer (one for scaling, the other for shifting). Batch normalization lets the model learn the optimal scal and mean of the inputs for each layer and makes the learning of layers in the network more independent of each other, which reduces the vanishing gradients problem and make the network less sensitive to the weight initialization.

4. Dropout slows down training. A dropout network typically takes 2-3 times longer to train than a standard neural network of the same architecture. A major cause of this increase is that the parameter updates are very noisy. Each training case effectively tries to train a different random architecture. Therefore, the gradients that are being computed are not gradients of the final architecture that will be used at test time. Dropout does not slow down inference since dropout is not applied during test time. However we do need to pay the cost of scaling down versions of the trained weights. If a unit is retained with probability p during training, the outgoing weights of that unit are multiplied by p at test time

5. if $\beta = 0.9$ then the terminal velocity (i.e. the maximum size of the weight updates) is equal to 10 times the gradient times the learning rate, so Momentum optimization ends up going to 10 times faster than Gradient Descent. Similarly if $\beta = 0.99999$ then Momentum optimization will go 100000 times faster than Gradient Descent.

6. ConvLayer 1, Given input size = 3, output size = 100

$$w1_{total} = 3 \times (3 \times 3) \times 100 = 2700$$

ConvLayer 2, Given input size = 100, output size = 200

$$w2_{total} = 100 \times (3 \times 3) \times 200 = 180000$$

ConvLayer 3, Given input size = 200, output size = 400

$$w3_{total} = 200 \times (3 \times 3) \times 400 = 720000$$

$w_{total} = w1_{total} + w2_{total} + w3_{total} = 902700$

7.
```
A = np.array([
[0,0,0,0,0,0,0],
[0,1,0,0,0,1,0],
[0,0,0,0,0,0,0],
[0,0,0,1,0,0,0],
[0,1,0,0,0,1,0],
[0,0,1,1,1,0,0],
[0,0,0,0,0,0,0]])

F = np.array([
[0,0,1],
[1,0,0],
[0,1,1]])

# (W - F + 2 * P) / 2 + 1 = (5 - 3 + 2*1) /2 + 1 = 3
V = np.zeros((3, 3))

V[0,0] = np.sum(A[:3,:3] * F)
V[0,1] = np.sum(A[:3, 2:5] * F)
V[0,2] = np.sum(A[:3, 4:7] * F)

V[1,0] = np.sum(A[2:5, :3] * F)
V[1,1] = np.sum(A[2:5, 2:5] * F)
V[1,2] = np.sum(A[2:5, 4:7] * F)

V[2,0] = np.sum(A[4:7, :3] * F)
V[2,1] = np.sum(A[4:7, 2:5] * F)
V[2,2] = np.sum(A[4:7, 4:7] * F)
# Result
V = array([
[0., 0., 0.],
[1., 0., 1.],
[0., 1., 1.]])
```