Scalable Machine Learning and Deep Learning - Review Questions 3

Anna Martignano, Daniele Montesi

May 4, 2020

1. **Is it OK to initialize all the weights of a neural network to the same value as long as that value is selected randomly using the initialization? Is it okay to initialize the bias terms to 0?**

   The right selection of a parameter initialization strategy is essential for stochastic gradient descent on non-convex cost functions since its performance are sensitive to the values of the initial parameters.
   A good practice is to set initial parameters which **break symmetry** between different units. Because if there are two hidden units with the same activation function connected to the same inputs, they will obtain exactly the same results unless different initial parameters are chosen.
   Therefore, it is not OK to initialize all the weights of a neural network to the same value. Instead, the bias term can be initialized to 0 since the assymetry breaking is only performed by the weights. The only issue with zero bias could be with ReLU activation function, it is better to use small constant value such as 0.01 for all biases because this ensures that all ReLU units fire in the beginning and therefore obtain and propagate some gradient.

2. **In which cases would you want to use each of the following activation functions: ELU, leaky ReLU, ReLU, tanh, logistic, and softmax?**

   Every activation function must be evaluated accordingly to these three factors:

   (a) The task of the prediction (i.e. Binary/Multi-class classification);
   (b) The Weight produced by the network (positive/negative);
   (c) The neural network depth.

   Here we describe for each activation function their pros and cons:

   - **Logistic**: The logistic is the generalization of the Sigmoid having function:

$y(z) = \dfrac{L}{1 + exp(-k(z - z_0))}$,
where L is the width, z_0 is the centered position, and k the scale. We obtain the Sigmoid function when k=1, L=1 and $z_0 = 0$. It is a s-shaped function that can be used for binary classification tasks. However, it suffers from Vanishing gradient and must not be used in deep neural networks.

- **TanH**: Similar to Sigmoid, but is Zero centered, —making it easier to model inputs that have strongly negative, neutral, and strongly positive values. However, it shouldn't be used with deep neural networks (Vanishing Gradient)
- **SoftMax**: Used in Multiclass classification tasks as it makes possible to choose the output neuron with the maximum value.
- **ReLU**: Solves the issue of vanishing gradient, so you can use with deep feed forward neural networks. Problem: not possible to use with negative weights as the network will never backpropagate negative values . (Derivative of negative values is 0 in ReLU, also known as *Dying ReLU*).
- **Leaky ReLU**: Solves the issue of ReLU, by taking into account a value for the negative weight multiplied by a constant: $\max(\alpha * z, z)$. However, there might be inconsistent values for negative weights. It can be used for the same types networks as for ReLU.
- **ELU**: The Exponential Linear Unit activartion function combines the approaches of Leaky Relu by substituting the $\max(\alpha * z, z)$ with $\max(\alpha * (exp(z) - 1), z)$. Using ELU, the predictions results to be accurate, but the training time much slower than the Leaky Relu.

3. **What is batch normalization and why does it work?**
One of the assumption of using gradient-based technique to update the parameter is that the following layers do not change. But this scenario in unrealistic in practice, since the layers are update simultaneously and it is very likely to obtain unexpected results.
Batch normalization is technique to **decouple** as much as possible the learning layers from each other by applying on the distribution of each layer's inputs:

   - normalization (divide by std_dev)
   - zero-centering (subtract the empirical mean)
   - scaling and shifting the result respectively by the factors γ and β

   If we only perform zero-centering and normalization, the weights in the next layer are no longer optimal. To tackle this problem, when performing batch normalization is essential to multiply the normalized output by a scaling parameter (i.e. standard deviation) and add a shifting paramter

(mean). In other words, batch normalization lets SGD do the denormalization by changing only these two weights for each activation, instead of losing the stability of the network by changing all the weights.

4. **Does dropout slow down training? Does it slow down inference (i.e., making predictions on new instances)?**

   Dropout is a regularization techniques used in Neural Networks to avoid the problem of overfitting. It is based in the dropping from a specific layer of one or more neurons that are dropped on the basis of a rate (probability between [0,1]).

   (a) The dropout usually slows down training speed since it roughly doubles the number of iterations required to converge even though training time for each epoch is less.

   (b) Instead, the time remains the same for inference.
   The only shrewdness needed to be taken into account is to multiply the input neurons by the dropping rate. Otherwise, each neuron will get a total input signal much larger as what the network was trained on and will be unlikely to perform well.

5. **What may happen if you set the momentum hyperparameter too close to 1? E.g., keras.optimizers.SGD(momentum=0.99999)**

   The momentum optimization is a technique that is able to speed-up the convergence of the machine learning algorithm without modifying the learning rate. It directly modifies the weight update formula introducing a term βm_i as the following:

$$w_i^{t+1} = w_i^t - (\beta m_i + \eta(\frac{\partial J(w)}{\partial w_i}))$$

   The idea of Momentum is, similarly to the meaning it has in physics, to keep going on the same direction defined by the previous step, in this case, by the gradient at the previous iteration. Here, β is the Momentum, m_i is the weight at the previous step. In the standard Momentum algorithm, if we set momentum very close to 1, we will have that the direction is going very fast to the optimal minimum, However, when it goes very close the point, the algorithm suffers of oscillations and do not converge. This problem is solved by Nesterov Momentum which takes into account the acceleration of the direction by measuring the gradient slightly ahead of an addendum βm:

$$w_i^{t+1} = w_i^t - (\beta m_i + \eta(\frac{\partial J(w + \beta m)}{\partial w_i}))$$

   As result, in case of big values for the momentum, the contribution will be starting fast to slow down early.

6. **Consider a CNN composed of three convolutional layers, each with 3 x 3 filters, a stride of 2, and SAME padding. The lowest layer outputs 100 feature maps, the middle one outputs 200, and the top one outputs 400. The input images are RGB images of 200 x 300 pixels. What is the total number of parameters w in the CNN?**

   Since we have a network made of only convolution layers, here we do reasoning focused on every single layer calculating their parameters with the following formula (m*n*l+1)*k, where m and n represents the dimension of the filter + 1 for the bias, and l and k represents respectively the input feature maps and the output feature maps of the considered layer:

Layer	l	k	Param.	Reasoning
Conv1	3 (RGB)	100	2800	(m*n*l+1)*k = (3*3*3+1)*100
Conv2	100	200	180200	(m*n*l+1)*k = (3*3*100+1)*200
Conv3	200	400	720400	(m*n*l+1)*k = (3*3*200+1)*400

   In total there are **903400** parameters for this network. Image size and stride are not relevant in this computation.

7. **Consider a CNN with one convolutional layer, in which it has a $3x3$ filter (as shown below) and a stride of 2. Please write the output of this layer for the given input image (the left image in the following figure)?**

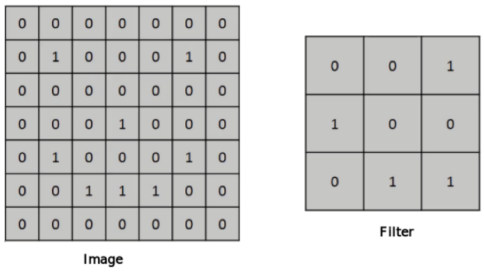

Figure 1: CNN

Given:

- Filter order n: number of row/columns
- image order N: number of row/columns
- stride s: the cells to "slide" in order to apply the filter next

In case of **Zero Padding**, it must hold the property:

$$(N - n)mod(s) == 0$$

The output of the filter applied to the image is a matrix having shape equal to (3,3), due to the zero padding in the borders and the striding of 2. We can observe that in the presence of padding and stride = 1, the output layer must have maintained the same shape of the input image. The 9 points where the filter is centered in the image are shown below.

Image centers for the filter:

	1		4		7	
	2		5		8	
	3		6		9	

The resulting output layer is computer by applying the **Hadamand product** (or element-wise product) between each $3x3$ matrix centered in the points shown in the picture (from 1 to 9) and the $3x3$ filter matrix. The resulting filter is shown here below:

Output:

0	0	0
1	0	1
0	1	1