Neural Networks

Lab 5

Juan Jose Mendez Torrero (s3542416)
Sharif Hamed (s2562677)

June 6, 2018

1 Implementing the convolutional layer

1. In order to go through all the images, we have created the following for-loop.

```
for i= 1: numImages
```

2. Then, to keep the selected image, we have kept it as the following code shows:

```
im=images(:,:, i);
```

3. Now, it is time to go through the filters (or kernels), hence this should be the result:

```
for j= 1 : numFilters
```

4. In order to calculate the convolved feature, first, we have to keep into a variable the filter that we are going to use. Secondly, we have to rotate it 180Âž, that means that we have to used the *rot90* function. Finally, we have used the built-in function *conv2* in order to calculate the convolved feature. Finally, we use the sigmoid function to calculate the output of the layer so we can keep it in the correct position.
The code that correspond to this explanation is as follows:

```
filter = W(:,:,j);
filter = rot90(rot90(filter));
result = conv2(im, filter, 'valid');
bias = b(j,1);
activation = sigmoid(bias+result);
convolvedFeatures(:, :, j, i) = activation;
```

2 Implementing the mean pooling layer

1. As when we have created the convolutional layer, now we have to create a for-loop in order to go through all the images. To do so, we have done this:

```
    for i= 1:numImages
```

2. The same has to be done in order to go through all the filters. Hence, we have created the following for-loop:

```
    for j = 1:numFilters
```

3. In order to compute the mean pooling layer, we have to split the matrix into *PoolDim x PoolDim* matrix, and then, we have to take the mean between all the components of this splitted matrix. Then, we keep this value into the pooledFeatures variable. We have to do this through all the original matrix. The following code shows how we have reached this solution.

```
for k=1:(convolvedDim/poolDim)
    for l=1:(convolvedDim/poolDim)
    pooledFeatures(k,l,j,i) = mean2(convolvedFeatures((poolDim*(k-1)+1):poolDim*k,
    (poolDim*(l-1)+1):poolDim*l,j,i));
    end
end
```

3 Implementing the forward pass

1. In order to calculate the activation of the convolutional layer, we have to use the method that we have done earlier, the method **cnnConvole**. After this, we have to pass to that method the dimension of the filter, the number of filters, all the images, the weight matrix and finally the bias of the neurons. The following code shows the result.

```
activations = cnnConvolve(filterDim, numFilters, images, Wc, bc);
```

2. To obtain the pooled activation, as before, we have used a method that we have done before, **cnnPool**. This time, what we have to pass to this method is, the dimension of the pool and the activations that have been calculated in the previous step. The resulting code is as follows.

```
activationsPooled = cnnPool(poolDim, activations);
```

3. In order to change the matrix output into a vector, we have first, to calculate the dimension of the activation matrix. Then we have to multiply the first three dimensions and set its value as the number of rows of the vector. Secondly we have to calculate the numbers of columns, to do so, we have set the last dimension of the matrix as the columns of the vector. Finally, we have reshaped the matrix and we have set it as a vector. The following code shows the process.

```
activationSize = size(activationsPooled);
rowsDim = activationSize(1)*activationSize(2)*activationSize(3);
columnsDim = activationSize(4);
activationsPooled = reshape(activationsPooled, rowsDim, columnsDim);
```

4. (a) To calculate the activation, we just only have to use a matrix multiplication, between the weights and the inputs of the layer. The solution should looks like the following code.

```
probs = zeros(numClasses,numImages);
Y_wx = Wd*activationsPooled;
```

   (b) To add the bias, we have used the built-in function bsxfun, with the parameter @plus. What this do is to go element by element inside of the activation matrix and add it the value of the bias. This can be do like follows.

```
Y_wxb = bsxfun(@plus, Y_wx, bd);
```

   (c) This step is as follows.

```
Y_num = exp(Y_wxb);
```

(d) For this section, we have use the function bsxfun as before, but this time, we have use the variable @rdivide, which does the division between the Y_{num} matrix and the sum of all the items in Y_{num}. In order to do so, we have done this:

```
norm_Y_num = bsxfun(@rdivide, Y_num, sum(Y_num, 1));
```

(e) Finally, we have keep the normalized value into the variable probs. We have done this as follows:

```
probs = norm_Y_num;
```

4 Experiments

1. In our case we have 20 filters. When the network is training it applies a convolution to all filters and the image. In 'convolvedFeatures' we store the activation, in this activation a bias is added. 'convolvedFeatures' is located at the 'filterNumber' an the 'imageNumber'. As for the parameters of the network we have a vector with weights called 'theta'. Furthermore we have a struct that contains all the learn parameters called 'options'. We have a parameter called 'images' that contains a training set of 28*28*60000 images. At last we also have a parameter called 'labels' which contains labels that belong to images so that the network can test itself. In the end we want to compute opttheta which should contain all the optimized parameters.
2. The final result of the filter is showed in Figure 1. As we can observe, now, the filter does not present many random noisy patches as the initial one. Also, after the training we can see that there is a lot more structure to each filter.

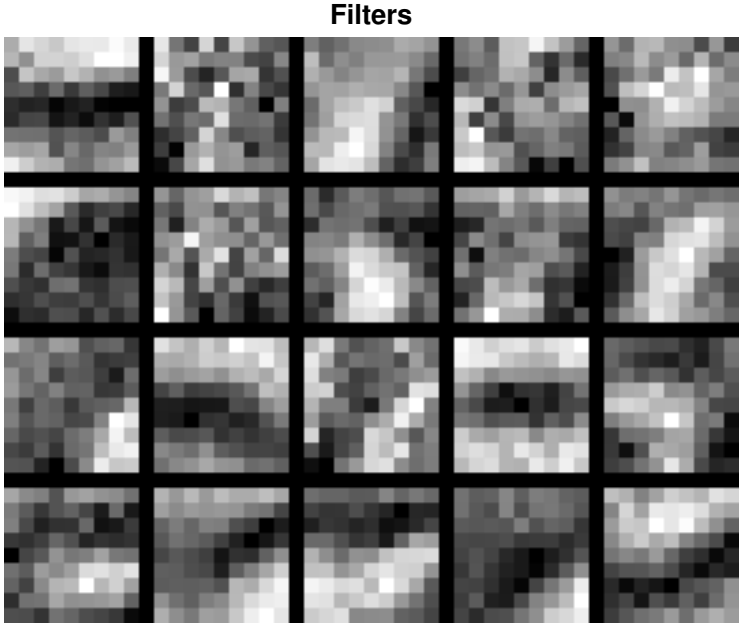

Figure 1: Resulted Filter

3. The accuracy, after three epochs, of the network is 97'16%, being this greater than 97%.