Machine Learning - Homework 2

Academic year 2017/2018

Tufa Alexandru Daniel 1628927

19 December 2017

1 Description of the problem

Given 4774 images of boats taken from the Maritime Detection, Classification and Tracking data set [1], our goal is to be able, by using this photos, to classify new instances into the 24 offered classes. The evaluation will be done on 1969 photos still offered by the MarDCT dataset.

2 Introduction

To solve this problem we have decided to use Convolutional Neural Networks. Let's describe what are the components of a CNN[1] and what is the reasoning that lies behind it.
A Neural Network is a mathematical model made of neurons, which are the basic computational units. A neuron takes a certain number of inputs x_i that are multiplied by chosen weights w_j. The summation of these products together with a certain variable b called bias is given in input to a non-linear function called `activation function`. This function will produce the final output of the neuron, that is also called activation of the function. There are many types of activation functions: Sigmoid function, TanH activation function and the Rectified Linear Unit, or ReLU function that is the one that we'll use in our CNNs, which can be expressed as

$$f(x) = max(0, x)$$

[1] Convolutional Neural Network

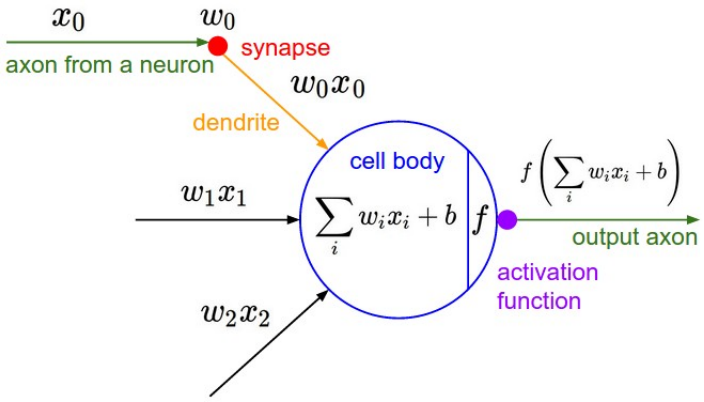

Figure 1: Neuron Model

2.1 Layers

If we stack multiple neurons in a single line we obtain a layer. Typically all the neurons in one layer do the same mathematical operation and this will give the name of the layer. Let's analyse the layers used in our NNs:

2.1.1 Convolutional Layer

The used operation in this layer is the convolution. This is a mathematical operation between two functions that produce a third function. In our case the input is a matrix of $width * height * depth$ that is convoluted, where the convolution operation is the dot product, to a filter matrix of $w_f * h_f * depth$. The result of this convolution is a single value stored in a new matrix that will be the output of the convolution. This single output is called feature map and in order for the network to learn different features we need multiple filters to produce many feature maps. For an image that has only 1 color channel, the output of the convolution layer will actually be 3 dimensional because we consider the width, the height and as depth the feature maps. If the input has multiple channels, we refer to an image with RGB layers, the output of the convolutional layer will be 4 dimensional. In order for the filter to cover all the image we need to add padding. The actual output of the convolution can be calculated as

$$w_{out} = \frac{w_{in} - w_k + 2p}{s} + 1 \quad h_{out} = \frac{h_{in} - h_k + 2p}{s} + 1$$

where w_k and h_k are the dimensions of the convolution, or kernel/filter matrix, p is the padding used and s is the stride, which is the step of sliding

of the kernel.

2.1.2 Pooling layer

In order to reduce the number of parameters in our network, which means to down-sample, and to make feature detection more robust by making it more impervious to scale and orientation changes we use the operation that is called pooling. The most common form of pooling is Max pooling where we take a filter of size $w_f * h_f$ and apply the maximum operation over the $w_f * h_f$ sized part of the image.

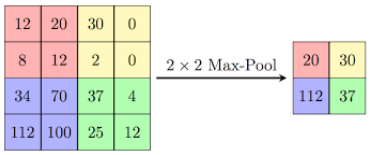

Figure 2: Pooling function

2.1.3 Fully connected layer

As its name implies this layer is fully connected with the output of the previous layer. It takes all neurons in the previous layer and connects it to every single neuron it has. These layers are mainly used because it's an easy way for the network to learn non-linear combinations of the features learned from convolutional and pooling layers.

2.2 Architecture of the network

Given these layers how do we arrange them? How many neurons do we use in each layer? Designing the architecture of a Neural Network is something that requires a lot of research. A typical guide line is to do use a number of convolutional layers followed by a maxpool layer and iterate this process. After some iterations all the layers are followed by one or two fully connected layers that will do the classification.

2.3 Training phase

Decided the architecture, we need to train the network to learn what features it must recognize in our images. The elements that will allow us to do this are the weights and biases used that we'll call the parameters of the network.

The objective is to find the best possible values for such parameters so that the final, or output, layer will predict the correct class of a test image. The process of finding the best set of parameters is called Backward Propagation. We start with a random set of parameters and keep changing these weights such that for every training image we get a correct output. The value that measures how fast we change the parameters of the network during training is called learning rate. There are many methods to change the weights, mainly called optimizers, and we'll use the GradientDescentOptimizer.
To define how far we are from the optimum we define a cost, or loss, function that tells us how correct our prediction is. A simple way of doing this is by using the mean squared error. Given y, the vector containing the actual values of the labeled images, and y', the vector containing the prediction of the network, a good indicator of the training would be the minimization of the distance between these two variables.

$$MSE = \frac{1}{2}\sum_{i=0}^{n}(y - y')^2$$

We'll be using a different cost function that derives from the concept of maximum likelihood. This function is the cross-entropy cost function. It can be derived in the following way. Let's suppose that our network predicts for our 24 classes their hypothetical occurence probabilities $y_1, y_2, ..., y_{24}$. Suppose that we observe k_1 instances of class 1, k_2 instances of class 2 and so on. The likelihood of this happening is:

$$P[data|model] = y_1^{k1} y_2^{k_2} ... y_2 4^{k_{24}}$$

Taking the logarithm and changing the sign we obtain:

$$-logP[data|model] = -k_1 log(y_1) - ... - k_{24} log(y_{24}) = -\sum_{i=0}^{24} k_i log(y_i)$$

If we divide the right-hand sum by the number of observations $N = k_1 + ... + k_{24}$ and denote the empirical probabilities as $y_i' = \frac{k_i}{N}$ we'll get the cross-entropy:

$$-\frac{1}{N} logP[data|model] = -\frac{1}{N}\sum_{i=0}^{24} k_i log(y_i) = -\sum_{i=0}^{24} y_i' log(y_i)$$

It's important to note that the training data will not be given to the network at once, but in small batches of 31 called batch-size. This value has been chosen so that no image is not taken in consideration. So, for the network to see all the images once, which is called an epoch, we would need 154 steps.

2.4 Testing phase

Once we have defined the values for the weights and biases, which means that we built our mode, we need to find out what are the predictions for each test image and finally calculate the accuracy.

3 Implementation

To build our CNN we have used the TensorFlow [2] library. There are multiple ways in which we could have implemented our Convolutional Neural Network and to focus more on the structure of the network and not on the implementational details we chose to use the "high-level API" by utilizing two main modules offered by TensorFlow:

- tf.layers which provides a set of neural networks layers
- tf.estimator which allows us to train and evaluate TensorFlow models

Let's start by analyzing the input to our model and how it's represented.

3.1 Dataset transformation and input representation

Given our dataset we will need to transform it in such a way that the network can elaborate. The input has to be a 4-dimensional array of the form

$$[batch\_size, image\_width, image\_height, num\_channels]$$

where for each item in the batch, given a width and height position we will have three values ranging from 0 to 255 that will indicate the values of the RGB color model. Three main functions were used to build the input:

- **encoder** which scans the training photos in order to do the encoding of the classes to transform the string names into integers
- **get_train_data** that scans the training directory and returns two numpy [3] arrays. The first one is of shape [4774,576000] where thanks to the Python Image Library [4] we stored the RGB byte representation of every image as an array. By using the encoder created before we also returned an array of labels.
- **get_eval_data** that does the same thing as *get_train_data*, but skips photos with a label that is not present in the training dataset.

3.2 Model construction

With the *cnn_model_fn* we will build our model that takes as input the images that we want to use as training or prediction, the labels of those images and the desired mode in which we want to use the model, be it training or prediction. This input, given that we pass it of shape $[-1, 576000]$[2] we need to reshape it to a 4-dimensional array, which is the input that we discussed before that the network needs. To build the layers discussed in chapter 2.1 we'll use three functions offered by the **tf.layers** module:

- **tf.layers.conv2d** that takes as input the 4-D Tensor, the number of filters that we'll apply, the size of the kernel matrix, the type of padding that we'll use and the activation function. This functions creates a 4-D weight variable of shape $[kernel\_width, kernel\_height, input\_channels, output\_channels]$ and computes the convolution with the input. Afterwards it sums the results with a bias variable that is an array of shape $[output\_channels]$ and computes the ReLU functions on the output of the summation. The `'same'` value for *padding* indicates that we want to pad the image so that, if stride 1 is used, the output image has the same dimensions of the input image.

- **tf.layers.max_pooling2d** which computes the max pooling operation on the input given the *pool_size* and *strides*.

- **tf.layers.dense** that allows us to create a fully connected layer of neurons.

Like every machine learning model it could suffer from overfitting and in order to avoid it we apply the dropout mechanism where, with a chosen probability, we set a fraction of input units to zero. This can be done with the **tf.layers.dense** function. The cost function is invoked with *tf.losses.softmax_cross_entropy* while, if in TRAINING mode we do the optimization using the passed cost function with the optimizer offered by *tf.train.GradientDescentOptimizer*, if in EVALUATION mode we compute the accuracy. In both cases a *tf.estimator.EstimatorSpec* is return which is a model to be run by an *Estimator*.

3.3 Training and evaluation

To do the actual training and evaluation we'll use the *tf.estimator.Estimator* class offered by TensorFlow. It offers functions like *train* and *evaluate* that

[2] -1 stands for the batch_size that can be anything

allows us to train the model for how many *epochs* and *steps* we prefer. It allows us to also do training on a fixed *batch_size*.

4 Architecture of the utilized CNNs

We shall see the different CNNs that we utilized for doing our training.

4.1 V1

Inspired by the MNIST TensorFlow tutorial [5], we have built a CNN with 2 main blocks:

- Convolution 1 with 32 filters, kernel size 5, padding='same'
- Max Pooling 1 of pool size 2 and strides=2
- Convolution 2 with 64 filters, kernel size 5, padding='same'
- Max Pooling 2 of pool size 2 and strides=2

In order to have as input to the fully connected layer not too many neurons we have decided to initially resize the image to 100∗30 so that, after applying 2 max pools, we would arrive at having $25 * 7 * 64 * batch\_size$ neurons. These neurons are fully connected to the dense layer of 1024 neurons where we apply a droupout of rate 0.4 in order to avoid overfitting. This layer is connected to the output layer of 24 neurons where classification will be done.

4.2 V2

This CNN is similar to the previous one but instead of 2 blocks of convolution followed by max pooling, we use 5 such blocks so that we don't need to resize the input image given that the convolutional layers will not alter the dimensions of their input while the max pooling will always halve it. Because we do it 5 times the input to the fully connected layer will still be of size $25 * 7 * num\_filters * batch\_size$. The convolutional layers will have respectively 32, 64, 128, 256, 256 filters. Differently from version 1 we will use one more fully connected layer of the same size.

4.3 V3

Instead of relying on max pooling for reducing the dimensions of the input image, we will use 5 convolutions of stride 2 with filters respectively 32, 32, 64, 128, 128. From now on all CNNs will have the same configuration after the flattening of the last convolution or max pooling similar to V2.

4.4 V4

- Convolution 1 with 32 filters, kernel size 5, padding='same', strides=2
- Convolution 2 with 32 filters, kernel size 5, padding='same', strides=2
- Convolution 3 with 64 filters, kernel size 5, padding='same'
- Max Pooling 1 of pool size 2 and strides=2
- Convolution 4 with 64 filters, kernel size 5, padding='same', strides=2
- Convolution 5 with 128 filters, kernel size 5, padding='same'
- Convolution 6 with 128 filters, kernel size 5, padding='same'
- Max Pooling 2 of pool size 2 and strides=2

4.5 V5

- Convolution 1 with 32 filters, kernel size 5, padding='same', strides=2
- Convolution 2 with 32 filters, kernel size 5, padding='same'
- Max Pooling 1 of pool size 2 and strides=2
- Convolution 3 with 64 filters, kernel size 5, padding='same', strides=2
- Convolution 4 with 64 filters, kernel size 5, padding='same'
- Max Pooling 2 of pool size 2 and strides=2
- Convolution 5 with 128 filters, kernel size 5, padding='same'
- Convolution 6 with 128 filters, kernel size 5, padding='same'
- Max Pooling 3 of pool size 2 and strides=2

4.6 V6

- Convolution 1 with 32 filters, kernel size 11, padding='same'
- Max Pooling 1 of pool size 2 and strides=2
- Convolution 2 with 64 filters, kernel size 9, padding='same'
- Max Pooling 2 of pool size 2 and strides=2
- Convolution 3 with 128 filters, kernel size 7, padding='same'
- Max Pooling 3 of pool size 2 and strides=2
- Convolution 4 with 256 filters, kernel size 5, padding='same'
- Max Pooling 4 of pool size 2 and strides=2
- Convolution 5 with 256 filters, kernel size 3, padding='same'
- Max Pooling 5 of pool size 2 and strides=2

5 Results

The training on all networks has been done with $batch\_size = 31$ so that no image is left out. We also trained on 50 epochs so in a total of 7700 steps.

CNNs	accuracy
V1	0.51831502
V2	0.63003665
V3	0.62454212
V4	0.64652014
V5	0.63736266
V6	0.65201467

The lowest value is of course the one generated by the first network because it had fewer convolutional layers so it couldn't learn many features. All other CNNs have similar results with the last one resulting in having the highest accuracy, probably given by the fact that we decrease the kernel size gradually from 11 to 3, instead of keeping it fixed at 5.

References

[1] Maritime detection, classification and tracking data set. http://www.dis.uniroma1.it/ labrococo/MAR/classification.htm.

[2] Tensorflow - an open-source software library for machine intelligence. https://www.tensorflow.org/.

[3] Numpy - package for scientific computing with python. http://www.numpy.org/.

[4] Python image library. http://www.pythonware.com/products/pil//.

[5] Mnist tensorflow tutorial. https://www.tensorflow.org/get$_started/mnist/pros$.