**67750: Advanced Practical Machine Learning** 2015/6

Exercise 4 — Deep learning and Convolutional Neural Nets

*Nadav Cohen* *TA: Alon Appleboim*

# 1 Theoretical Questions

Note that this section is not graded.

## 1.1 Parameterized ReLU

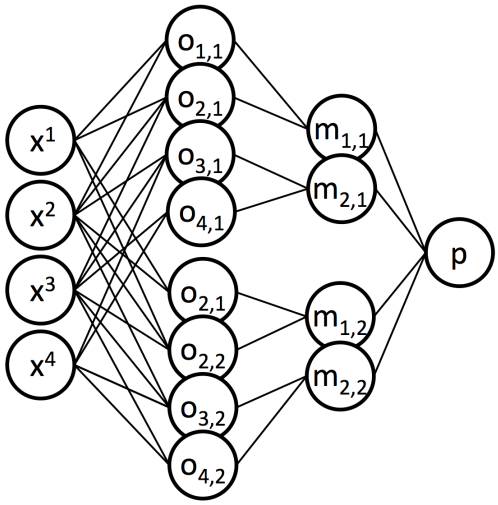
Define the following function:

$$f_i(o;t) = \max\{t, o_i\}$$

Will the incorporation of this function into a network define a larger class of hypotheses? explain your answer and/or give an example.

## 1.2 A Forward/Backward Pass

Consider the following CNN:




This network operates on single 4D instances (x). The first layer convolves the input with $w \in \mathbb{R}^{3x2}$ and passes the outputs through a ReLU to produce two channels: $o \in \mathbb{R}^{4x2}$.

Write the matrix form of the convolution for one channel.

The outputs of the convolutions are then max-pooled to $m \in \mathbb{R}^{2x2}$, and the prediction, $p$ is the result of a fully connected layer (parametrized by $u \in \mathbb{R}^{4,1}$). For the loss $L = (p-y)^2$, express $\frac{\partial L}{\partial w_{1,1}}$, and $\frac{\partial L}{\partial u_1}$ in terms of the forward pass variables $(x, o, m, p)$.

If you are not sure you know how the forward/backward iterations work, it's advisable to actually perform the following calculation: For the input $x = (1, -1, 2, -2)$, $y = 3$, calculate the forward pass variables, and the gradient, at the point $w_1 = (1, -1, 2)$, $w_2 = (0, 2, 1)$, and $u = (-1, 1, -1, 1)$ in parameter space.

# 2 Practical Exercise

Please submit a single tar file named "ex4_<ID1>_<ID2?>.tar.gz". This file should contain your code, and an "Answers.pdf" file in which you should write your answers and provide all figures/data to support your answers (and write your ID(s) in there as well, just in case). Alternatively, if you want to submit a Jupyter notebook, please contact the TA beforehand.

Exercise is written with Python3 in mind, but you can implement the solutions in any high-level language (Python, MATLAB, R, Ruby, etc.).

Finally, if you have any constructive remarks regarding the exercise (e.g. question X was annoying, question Y was too easy) we'll be happy to read them in your Answers file.

## 2.1 Exercise Requirements

1. Complete a quick and dirty code that learns a toy CNN (30 pts)
2. Transition to TensorFlow (TF), build a simple linear classifier, and apply it to the original MNIST set, a reduced MNIST set, and a noised MNIST set (15 pts)
3. Use TF to train a multilayer perceptron, and compare it to the linear model (15 pts)
4. Build a CNN and and compare it to the 2 previous models (20 pts)
5. Calibrate the entire learning process - network architecture, batch size, rates, regularization parameters, etc. Explain your choices, procedures, and demonstrate your results (20 pts)

## 2.2 A simple CNN

Since we are about to use TensorFlow for the bulk of the exercise, we start off with a simple network and implement an ad hoc learning algorithm, to understand the inner workings of computation graphs.

For a matrix $W \in \mathbb{Z}^{4x4}$, Define the functions:

$$y_1 = \sum_i (Wx)_i$$
$$y_2 = (W_{1,1}x_1 + W_{2,2}x_2)(W_{3,3}x_3 + W_{4,4}x_4)$$
$$y_3 = \log(\sum_i e^{(Wx)_i})$$

We'll try to learn these functions with two models: a linear model, and a toy CNN, as in the theoretic part. Our loss will be the weight-decayed $l_2$ loss:

$$l = (p - y)^2 + \lambda \|w\|^2$$

Where $p$ is the model's prediction. Complete the code that performs stochastic gradient descent for both the linear case and the convolutional net (found in *toy_example.py*). Demonstrate that your algorithm is indeed learning the functions, discuss the results and limitations of the two different models with respect to the different functions. Provide graphical comparisons of the results, and relate to them in your answers.

### 2.3 TensorFlow

After the last section, it should be obvious that building an arbitrarily complex neural net is trivial - it's simply a composition of functions. Furthermore, to learn the network parameters, one has to be able to perform gradient descent, which simply boils down to index accounting, as long as each layer is differentiable w.r.t to its input and/or the parameters. These insights led to the development of several packages that abstract the inner-workings of the network. Usually the networks are called computation graphs, each node in the graph is a function, and the edges are the output of one function, redirected to another function, in the form of a multi-dimensional array (or tensor). Google's package for python - TensorFlow (TF) - is the industry standard today, and we will now turn to use it.

#### 2.3.1 Data

There are tons of tutorials out there, you can start off with *the official tutorial*. Use TF to build a simple linear model, and apply gradient descent on a cross-entropy loss to learn a model for the MNIST dataset. Since the MNIST problem is quite easy, but hard problems require too much computation, we'll take the middle ground, and make the MNIST problem a bit more challenging: Transform the MNIST dataset to a reduced and noisy set (e.g. 14x14 instead of 28x28 with white noise). You can generate different reduction schemes (e.g. only 8x8 center pixels) and or different noising schemes (e.g. blurring) to make the learning task a more challenging. Generate at least three sets of increasing difficulty, and provide examples of the images before and after their corruption.

#### 2.3.2 Linear Model (15 pts)

Learn a simple linear model and test it your three datasets. Does your model generalize? How does it perform on the different data sets? Plot the learning curves: train/test error as a function of the iteration for the different data sets, and shortly discuss the results.

#### 2.3.3 First Deep Model (15 pts)

A multi-layer perceptron is a neural net consisting of several fully-connected layers, interleaved with activation functions. Build and train a multi-layer perceptron. You'll need to use:

- Fully connected layer - a linear dimensionality reduction to the prediction space - see *tf.matmul*
- Bias/Offset layer - Bias each channel by a single parameter - *simple variable addition will do the trick*
- Activation layer - the first non-linear operator of the network - see, e.g. *tf.nn.relu*

Select the model's depth, a loss function, and some optimization algorithm, and perform gradient descent to learn the model. Discuss your choices.

#### 2.3.4 A Convolutional Neural Net

A Convolutional Neural Net (CNN) is similar to multilayer perceptron, with the a sequence of convolution layers replacing the fully-connected layers, and the occasional pooling layer, ending with a fully connected layer just before the network's output layer. Using the guidelines presented in class, build one or more CNNs, and train them. You"l probably require:

- Reshaping layer - reshape each sample to a 3d object (height, width, channels) - see *tf.reshape*
- Convolution layer - convolve each sample with a kernel (and a specific stride) and map each patch from one channel space to another channel space - see *tf.nn.conv2d*
- Pooling layer - where the space-dimensions of instances is reduced, allowing the detection of less local features. for example, with max-pooling, the maximum value over a certain patch is taken (determined by kernel and stride again) - see *tf.nn.max_pool*

Again, select a loss function, and some optimization algorithm, and perform gradient descent to learn the model. Explain your choices.

Compare all your models on the different sets data sets. For each model/dataset plot the learning curve, and the final classification accuracy. plot the accuracy as a function of the number of model parameters.

Finally, for a single model, test the effects of batch size, and input normalization (whitening, as discussed in class) on the properties of the learning process and on the model's accuracy. Discuss any interesting results you encounter.