Machine Learning
Exercise 6

Marc Toussaint
TAs: Janik Hager, Philipp Kratzer
Machine Learning & Robotics lab, U Stuttgart
Universitätsstraße 38, 70569 Stuttgart, Germany

May 21, 2019

PRELIMINARY: We will add more hints to solve the second exercise on Wednesday. Please check for a new version on Wednesday evening.
(DS BSc students can skip the exercise 2b and 3.)

1 Getting started with tensorflow (4 Points)

Tensorflow (https://www.tensorflow.org/) is one of the state-of-the-art computation graph libraries mostly used for implementing neural networks. We recommend using the Python API for this example. Install the tensorflow library with pip using the command `pip install tensorflow --user` or follow the instructions on the webpage for your platform/language.

For the logistic regression, we used the following objective function:

$$L(\beta) = -\sum_{i=1}^{n} \Big[y_i \log p_i + (1-y_i) \log[1-p_i] \Big] \tag{1}$$

$$\text{where} \quad p(x) := P(y{=}1 \,|\, x) = \sigma(x^\top \beta) \;, \quad p_i := p(x_i) \tag{2}$$

a) Implement the loss function by using standard tensor commands like `tf.math.sigmoid()`, `tf.tensordot()`, `tf.math.reduce_sum()`. Define the variables $X \in \mathbb{R}^{100\times 3}$, $y \in \mathbb{R}^{100}$ and $\beta \in \mathbb{R}^3$ as `tf.placeholder`. Store the computation graph and display it in a browser using tensorboard. (2 P)

Hints:

- You can save the computation graph by using the following command:
  `writer = tf.summary.FileWriter('logs', sess.graph)`
- You can display it by running tensorboard from the command line: `tensorboard --logdir logs`
- Then open the given url in a browser.

b) Run a session to compute the loss, gradient and hessian. Feed random values into the input placeholders. Gradient and Hessian can be calculated by `tf.gradients()` and `tf.hessians()`. Compare it to the analytical solution using the same random values. (2 P)
Code calculating the analytical solutions of the loss, the gradient and the hessian in python:

```
def numpy_equations(X, beta, y):
   p = 1. / (1. + np.exp(-np.dot(X, beta)))
   L = -np.sum(y * np.log(p) + ((1. - y) * np.log(1.-p)))
   dL = np.dot(X.T, p - y)
   W = np.identity(X.shape[0]) * p * (1. - p)
   ddL = np.dot(X.T, np.dot(W, X))
   return L, dL, ddL
```

2 Classification with NNs in tensorflow (6 Points)

Now you will directly use tensorflow commands for creating neural networks.

a) We want to verify our results for classification on the dataset "data2Class.txt" by implementing the NN using tensorflow. Create two dense layers with ReLU activation function and $h_1 = h_2 = 100$. Map to one output neuron (i.e. $h_3 = 1$) without activation function. Display the computation graph as in the previous exercise. Use `tf.losses.hinge_loss()` as a loss function and the Adam Optimizer to train the network. Run the training and plot the final result. (3 P)

Hints:

- There are many tutorials online, also on `www.tensorflow.org`. HOWEVER, most of them use the *keras* conventions to first abstractly declare the model structure, then compile it into actual tensorflow structure (Factory pattern). You can use this, but to really learn tensorflow we recommend using the direct tensorflow methods to create models instead.
- Here is an example that declares an input variable, two hidden layers, an output layer, a target variable, and a loss variable:

```
input = tf.placeholder(shape=[None,2], dtype=tf.float32)
target_output = tf.placeholder('float')

relu_layer_operation = tf.layers.Dense(100,
                     activation=tf.nn.leaky_relu,
                     kernel_initializer=tf.initializers.random_uniform(-.1,.1),
                     bias_initializer=tf.initializers.random_uniform(-1.,1.))

linear_layer_operation = tf.layers.Dense(1,
                     activation=None,
                     kernel_initializer=tf.initializers.random_uniform(-.1,.1),
                     bias_initializer=tf.initializers.random_uniform(-.01,.01))

hidden1 = relu_layer_operation(input)
hidden2 = relu_layer_operation(hidden1)
model_output = linear_layer_operation(hidden2)

loss = tf.reduce_mean(tf.losses.hinge_loss(logits=model_output, labels=target_output))
```

- Use any tutorial to realize the training of such a model.

b) Now we want to use a neural network on real images. Download the BelgiumTS[1] dataset from: https://btsd.ethz.ch/shareddata/BelgiumTSC/BelgiumTSC_Training.zip (Training data) and https://btsd.ethz.ch/shareddata/BelgiumTSC/BelgiumTSC_Testing.zip (Test data). The dataset consists of traffic signs according to 62 different classes. Create a neural network architecture and train it on the training dataset. You can use any architecture you want but at least use one convolutional layer. Report the classification error on the test set. (3 P)
Hints: Use `tf.layers.Conv2D` to create convolutional layers, and `tf.contrib.layers.flatten` to reshape an image layer into a vector layer (as input to a dense layer). The following code can be used to load data, rescale it and display images:

```
import os
import skimage
from skimage import transform
from skimage.color import rgb2gray
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf

def load_data(data_directory):
    directories = [d for d in os.listdir(data_directory)
                   if os.path.isdir(os.path.join(data_directory, d))]
    labels = []
    images = []
    for d in directories:
        label_directory = os.path.join(data_directory, d)
        file_names = [os.path.join(label_directory, f)
                      for f in os.listdir(label_directory)
                      if f.endswith(".ppm")]
        for f in file_names:
            images.append(skimage.data.imread(f))
            labels.append(int(d))
    return np.array(images), np.array(labels)

def plot_data(signs, labels):
    for i in range(len(signs)):
        plt.subplot(4, len(signs)/4 + 1, i+1)
        plt.axis('off')
        plt.title("Label {0}".format(labels[i]))
```

[1] Belgium traffic sign dataset; Radu Timofte*, Markus Mathias*, Rodrigo Benenson, and Luc Van Gool, Traffic Sign Recognition - How far are we from the solution?, International Joint Conference on Neural Networks (IJCNN 2013), August 2013, Dallas, USA

```
        plt.imshow(signs[i])
        plt.subplots_adjust(wspace=0.5)
    plt.show()


images, labels = load_data("./Training")

# display 30 random images
randind = np.random.randint(0, len(images), 30)
plot_data(images[randind], labels[randind])

images = rgb2gray(np.array([transform.resize(image, (50, 50)) for image in images]))  # convert to 50x50
```

3 Bonus: Stochastic Gradient Descent (3 Points)

(Bonus means: The extra 3 points will count to your total, but not to the required points (from which you eventually need 50%).)

We test SDG on a squared function $f(x) = \frac{1}{2}x^\top Hx$. A Newton method would need access to the exact Hessian H and directly step to the optimum $x^* = 0$. But SDG only has access to an estimate of the gradient. Ensuring proper convergence is much more difficult for SGD.

Let $x \in \mathbb{R}^d$ for $d = 1000$. Generate a sparse random matrix $J \in \mathbb{R}^{n\times d}$, for $n = 10^5$ as follows: In each row, fill in 10 random numbers drawn from $\mathcal{N}(0, \sigma^2)$ at random places. Each row either has $\sigma = 1$ or $\sigma = 100$, chosen randomly. We now define $H = J^\top J$. Note that $H = \sum_{i=1}^n J_i^\top J_i$ is a sum of rank-1 matrices.

a) Given this setup, simulate a stochastic gradient descent. (2P)

1. Initialize $x = \mathbf{1}_d$.
2. Choose $K = 32$ random integers $i_k \in \{1, .., n\}$, where $k = 1, .., K$. These indicate which data points we see in this iteration.
3. Compute the stochastic gradient estimate

$$g = \frac{1}{K}\sum_{k=1}^{K} J_{i_k}^\top (J_{i_k} x)$$

   where J_{i_k} is the i_kth row of J.
4. For logging: Compute the full error $\ell = \frac{1}{2n}x^\top Hx$ and the stochastic mini batch error $\widehat{\ell} = \frac{1}{2K}\sum_{k=1}^{K}(J_{i_k}x)^2$, and write them to a log file for later plotting.
5. Update x based on g using plain gradient descent with fixed step size, and iterate from (ii).

b) Plot the *learning curves*, i.e., the full and the stochastic error. How well do they match? In what sense does optimization converge? Discuss the *stationary distribution* of the optimum. (1P)

c) Extra: Test variants: exponential cooling of the learning rate, Nesterov momentum, and ADAM.