hw4

Wenhua Bao 2512664
Xinrui Chen 2503398

August 2, 2020

Übung TUDaExercise
LastName, FirstName: _____ EnrollmentID: ␣␣␣␣␣␣␣␣

Task 1: Neural Networks

1a)

1.
Get the data: firstly load the data. The output of the original dataset is 1 dimension which includes number 0 9. We change it to 10 dimension, and describe the number with the index. The index of value 1 is the number.

Code:

```
mnist_train_in = np.loadtxt("./dataSets/mnist_small_train_in.txt",delimiter=',')
mnist_train_out = np.loadtxt("./dataSets/mnist_small_train_out.txt")
mnist_test_in = np.loadtxt("./dataSets/mnist_small_test_in.txt",delimiter=',')
mnist_test_out = np.loadtxt("./dataSets/mnist_small_test_out.txt")
train_y = np.zeros((mnist_train_out.shape[0],10))
test_y = np.zeros((mnist_test_out.shape[0],10))
for k in range(train_y.shape[0]):
   g = int(mnist_train_out[k])
   train_y[k][g] = 1
for k in range(test_y.shape[0]):
   g = int(mnist_test_out[k])
   test_y[k][g] = 1
```

2.
We built a network, in which input has 784 nodes, hidden layer has 30 nodes and output has 10 nodes.

Code:

```
class neuralNetwork:
   def __init__(self, num_input, num_output, hidden_layer):
      self.input = num_input
      self.output = num_output
      self.hidden = hidden_layer
      self.biases = [np.random.randn(hidden_layer, 1), np.random.randn(num_output, 1)]
      self.weights = [np.random.randn(num_input, hidden_layer),
          np.random.randn(hidden_layer,num_output)]
```

3.
Forward propagation: We choose sigmoid activation for hidden layer and output layer and $L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$ as the loss function. Here Nonlinear log-likelihood can also be used, because it is a classification problem.
Code:

```
   def fw(self, x):
      a0 = np.dot(x, self.weights[0]).T+self.biases[0]
      z0 = sigmoid(a0)
      a1 = np.dot(z0, self.weights[1]).T+self.biases[1]
      z1 = sigmoid(a1)
      return z1
```

```
def sigmoid(x):
   return 1.0/(1.0+np.exp(-x))
def d_sigmoid(x):
   return sigmoid(x)*(1-sigmoid(x))
```

4.
Backward propagation: update the weight and bias.

Code:

```
def bw(self, x, y):
    a0 = np.dot(x.T, self.weights[0])+self.biases[0].T
    z0 = sigmoid(a0)
    a1 = np.dot(z0, self.weights[1])+self.biases[1].T
    z1 = sigmoid(a1)
    # backward pass
    db1 = (z1 - y) * d_sigmoid(a1)
    db0 = np.dot(db1, self.weights[1].T) * d_sigmoid(a0)
    x = np.mat(x).T
    z0 = np.mat(z0).T
    dw0 = np.dot(x, db0)
    dw1 = np.dot(z0, db1)
    db = [db0, db1]
    dw = [dw0, dw1]
    return db, dw
```

```
def update_parameter(self, mini_batch, learn_rate):
    b_new= [np.zeros(b.shape) for b in self.biases]
    w_new = [np.zeros(w.shape) for w in self.weights]
    number = len(mini_batch[0])
    for k in range(number):
        print('k', k)
        x = mini_batch[0][k]
        y = mini_batch[1][k]
        db, dw = self.bw(x, y)

        b_new = [b + db.T for b, db in zip(b_new, db)]
        w_new = [w + dw for w, dw in zip(w_new, dw)]
    self.weights = [w - learn_rate * nw
                for w, nw in zip(self.weights, w_new)]
    self.biases = [b - learn_rate * nb
                for b, nb in zip(self.biases, b_new)]
```

5. Train the data: we use the Mini-Batch Gradient Descent to train the data, with the mini-batch size 10, 30, 100. Code:

```
def random_mini_batch(self, x, y, size):
    N = self.input
    N = x.shape[0]
    # shuffle
    x_y = list(zip(x, y))
    np.random.shuffle(x_y)
    x, y = zip(*x_y)
    x = np.array(x)
    y = np.array(y)
    mini_batch_num = N//size
    print('mini_batch_num:',mini_batch_num)
    mini_batch_list = []
    for i in range(mini_batch_num):
        mini_batch_x = x[i * size:(i+1) * size, :]
        mini_batch_y = y[i * size:(i+1) * size]
        mini_batch = (mini_batch_x, mini_batch_y) # mini_batch is a tuple
        mini_batch_list.append(mini_batch)
    return mini_batch_list
```

```
def train(self, x, y, iter_num, mini_batch_size, learn_rate, test_x, test_y):
```

```
        for j in range(iter_num):
            mini_batches = self.random_mini_batch(x, y, mini_batch_size)
            for mini_batch in mini_batches:
                self.update_parameter(mini_batch, learn_rate)
```

6.
set the network and train with the learning rate 1, 2 and 3. But unfortunately, there are still several errors when we run the code. So we don't have a evaluation for this network.

1b)

Difference:
What multi-layer neural networks do is feature mapping to values. Features are picked manually. What deep learning does is extract features from the signal and get results/value from the features. The features are selected by the network itself instead of people. In traditional neural networks, back propagation is used, which is an iterative algorithm to train the entire network, randomly setting the initial value, calculating the output of the current network, and then changing the parameters of the previous layers according to the difference between the current output and the label until convergence. In deep learning, we don't use this method, because there are too many layers in deep learning and then the gradient descent method makes less sense.

Which limitations of classical NN does deep learning overcome?
1. when the dataset is very big, it needs to extract features of the raw data as input manually. Irrelevant variables must be ignored, while useful information is retained. But what can be considered as the usefull information is hard to decide. Then using the deep learning is better.
2. Classical NN cannot process time series data (e.g. audio) because classical neural networks do not contain time parameters.
3. In the area of image recognition ,natural language processing and semantic analysis translation, deep learning is better.

1 Task2

1.0.1 a

1. SVM is the linear classifier with the largest spacing defined on the feature space . 2.Advantages: The classifier only depends on a few data points. Both the original SVM formulation (primal) as well as the derived dual formulation are quadratic programming problems (quadratic cost, linear constraints), which have unique solutions that can be computed efficiently

1.0.2 b

$$\begin{aligned} &\arg\min_{w,b} \frac{1}{2}\|w\|^2 \\ &\quad \text{s.t. } y_i\left(w^\top x_i + b\right) - 1 \geqslant 0 \quad \forall i \end{aligned}$$

1.0.3 c

In reality the modes have noise to make models not linear separable, slack variables make the model can tolerant noise and become linear separable.

$$\begin{aligned} &\arg min \tfrac{1}{2}\|w\|^2 + C\sum_{i=1}^N \zeta_i \\ &\text{sit. } y_i\left(w^\top x_i + b\right) \geqslant 1 - s_i \quad (i = 1, 2, \ldots N) \\ &\zeta_i \geqslant 0 \end{aligned}$$

1.0.4 d

$$\begin{aligned} L(\omega, b, \alpha) &= \tfrac{1}{2}\|w\|^2 - \alpha_i \textstyle\sum_{i=1}^N (y_i\left(w^\top x_i + b\right) - 1\Big) \\ &= \tfrac{1}{2}\|w\|^2 - \textstyle\sum_{i=1}^N \alpha_i y_i w^\top x_i - \sum_{i=1}^N \alpha_i y_i b + \sum_{i=1}^N \alpha_i \\ &\frac{\partial L}{\partial w} = w - \textstyle\sum \alpha_i y_i x_i = 0 \\ &w = \textstyle\sum \alpha_i y_i x_i \\ &\frac{\partial L}{\partial b} = -\textstyle\sum \alpha_i y_i = 0 \\ &\textstyle\sum \alpha_i y_i = 0 \end{aligned}$$

$$\begin{aligned} L(w, b, \alpha) &= \frac{1}{2}\sum_{i=1}^N\sum_{j=1}^N \alpha_i\alpha_j y_i y_j \left(x_j^\top x_i\right) - \sum_{i=1}^N\sum_{j=1}^N \alpha_i a_j y_i y_j \left(x_j^\top x_i\right) + \sum_{i=1}^N a_i \\ &= \sum_{i=1}^N a_i - \frac{1}{2}\sum_{i=1}^N\sum_{j=1}^N \alpha_i\alpha_j y_i y_j \left(x_j^\top x_i\right) \end{aligned}$$

result

$$\begin{aligned} &\min \sum_{i=1}^N \alpha_i - \frac{1}{2}\sum_{i=1}^N\sum_{j=1}^N \alpha_i\alpha_j y_i y_j \left(x_i^\top x_j\right) \\ &\qquad\qquad \text{s.t. } \alpha_i \geqslant 0 \\ &\qquad\qquad \sum_{i=1}^N \alpha_i y_i = 0 \end{aligned}$$

1.0.5 e

we can skip the derivation

1.0.6 f

1.Replace every occurrence of a scalar product between features with a kernel function

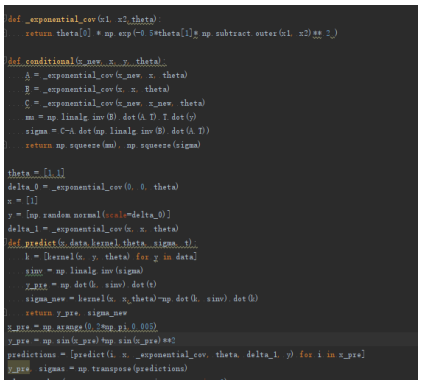
$$K\left(\mathbf{x}_i, \mathbf{x}_j\right) = \phi\left(\mathbf{x}_i\right)^{\top} \phi\left(\mathbf{x}_j\right)$$

2.A kernel $K\left(\mathbf{x}_i, \mathbf{x}_j\right)$that allows us to compute the scalar product without making the mapping explicit

1.0.7 g

2 Task3

2.1 a



```
def _exponential_cov(x1, x2,theta):
    return theta[0] * np.exp(-0.5*theta[1]* np.subtract.outer(x1, x2)** 2)

def conditional(x_new, x, y, theta):
    A = _exponential_cov(x_new, x, theta)
    B = _exponential_cov(x, x, theta)
    C = _exponential_cov(x_new, x_new, theta)
    mu = np.linalg.inv(B).dot(A.T).T.dot(y)
    sigma = (C-A.dot(np.linalg.inv(B)).dot(A.T))
    return np.squeeze(mu), np.squeeze(sigma)

theta = [1,1]
delta_0 = _exponential_cov(0, 0, theta)
x = [1]
y = [np.random.normal(scale=delta_0)]
delta_1 = _exponential_cov(x, x, theta)
def predict(x,data,kernel,theta, sigma, t):
    k = [kernel(x, y, theta) for y in data]
    sinv = np.linalg.inv(sigma)
    y_pre = np.dot(k, sinv).dot(t)
    sigma_new = kernel(x, x,theta)-np.dot(k, sinv).dot(k)
    return y_pre, sigma_new
x_pre = np.arange(0,2*np.pi,0.003)
y_pre = np.sin(x_pre)+np.sin(x_pre)**2
predictions = [predict(i, x, _exponential_cov, theta, delta_1, y) for i in x_pre]
y_pre, sigmas = np.transpose(predictions)
```