Lab Six

COMP 219 - Advanced Artificial Intelligence

Xiaowei Huang
Cameron Hargreaves

November 5th 2018

1 Introduction to Deep Learning with Multi Layer Perceptrons

1.1 Reading

Begin by reading chapter twelve of Python Machine Learning found in the learning resources section of the vital page for COMP219

2 Implementing the code from the book

1. Begin by downloading the zipped code file from the course website into a local directory
2. Activate the environment we created in the last lab and navigate to the folder you have downloaded the code to
3. Open the code Lab6-1.py, update the global variable MNIST_PATH to point to the mnist folder you have downloaded the code to and verify it runs, it should print the rows and columns of X_test and y_test as well as displaying a 5x2 plot of ten of the handwritten digits.
4. Read through the code in NeuralNetMLP and add a brief comment to each function outlying what it does. Import the neural net MLP class into our Lab6-1.py code using `from NeuralNetMlp import NeuralNetMLP`
   The code for this is significantly more complicated than anything we have touched so far so don't worry if you do not understand it at first.
5. Initialise a neural net with 10 outputs, 784 inputs, 50 hidden neurons, L2 regularisation of 0.1, L1 regularisation of 0, **100** epochs, a learning rate of 0.001, an alpha of 0.001, a decrease constant of 0.00001, shuffle to be True, 50 minibatches, and a random_state of 1. Use the fit method of this neural network (set the print_progress flag to be True) and plot the cost_ property of the neural network class (set y limit to around 10,000).

6. As you can see even on modest hardware training neural networks can take significant time, and our cost is still falling significantly. To save time when working with neural networks it is almost always a good idea to save the state of the network after training so that we don't need to repeat this. Most deep learning frameworks have their own methods of doing this; here we will simply use the standard python object serialisation library, pickle, to save this network as an object. Import pickle and use the dump function to save the neural network to a local file, comment out the NeuralNetMLP.fit() function and simply use the load function from pickle to save to a variable. We have additionally trained a network for 1000 epochs, load this neural network from network/trainedMLP_1000.pkl and save this to another variable.

7. If we take the average of the two costs we can see it has fallen from 1443 to 665 after a further 900 epochs of training, unfortunately loss is a harder metric to visualise than accuracy. Create two new lists of predictions using the predict() method on our X_train data, create a function that returns the accuracy of two lists, and compare the accuracy of the two networks predictions compared to the y_train data. Use this same function to see the accuracy of our testing data (load this in with the load_mnist function with a kind of t10k)

8. Train a third network using 300 hidden neurons and 100 epochs. How does the loss and accuracy of this network compare? How does the time of this operation change?

3 Implementing with PyTorch

Due to the speed of using vanilla python, in practice we would never use this for deep learning, instead we would use a multi-threaded, GPU enabled C++ framework such as pytorch, below we will reimplement the above problem using this framework.

1. First we will create a new environment, follow the same steps from the last lab, give this environment the name pytorch, except this time install pytorch and torchvision additionally. These steps are for getting set up on the university computers, you may need to do slightly different steps on your personal machine
   ```
   conda create --name pytorch
   activate pytorch
   conda install -c pytorch pytorch-cpu
   pip install torchvision
   ```
   Navigate to the directory you have downloaded the example code in, update the variable MNIST_PATH to './mnist' and verify that this runs

2. Open the Lab6-2.py file, update the MNIST_PATH variable to your local mnist folder and verify that the code works, run the code and we should run through a single epoch with a testing accuracy of around 10-20%.
   Here we have created two pytorch data loader objects that will iterate through our training/testing data and feed it into our network, we have defined a neural network from the NeuralNetPytorch file with 784 inputs, 500 hidden layers and ten outputs, if we look into our NeuralNet class we can see that we have defined three functions,

two linear layers from our input size to our hidden size and from our hidden size to our output size, and a Sigmoid function. In our forward function in this class we have chained these together. In our main code we have defined our loss function as Cross Entropy Loss, and our optimizer (how we will update the gradients of our weights) as stochastic gradient descent. In our main loop we iterate through each image in our training set, make a prediction, which we save in out, calculate the loss against the actual label, then propagate this loss backwards through the network to update the weight. In the second loop we simply calculate the correct predictions in our training set and print the overall accuracy.

3. As you can see, in very few lines of code we have put together a very complex neural network, unfortunately right now it's terrible at predicting handwritten digits, we can fix this by tuning the hyper-parameters of our network, as well as completely changing the network architecture. To begin with we will look at the activation function, here we have used a Sigmoid function as this is what has been historically used for neural networks (as this most closely matches biological neurons). Instead change this to be a rectified linear unit (ReLU) function and re-run the code, these are the most currently used activation functions currently, look into how these work as they are reasonably simple.

4. We should have seen a solid improvement on our networks accuracy, but we can still do better, here we have used a stochastic gradient descent algorithm for updating the network weights as our optimizer. Run the network training again using the Adam algorithm to update the network weights.

5. We would normally run the network for many more than one epoch, however this can take significant time, make the learning rate something larger to compensate for this and see how the accuracy changes. When does increasing the learning rate start to become a bad thing?

6. Using this framework it has become very easy to add network complexity (which usually increases accuracy) add another hidden layer of 500 neurons, connect this to the first hidden layer and the output (using the ReLU activation function) and run the network again. Notice that when we increased the number of hidden layer neurons in our multi-layer perceptron the code slowed down significantly, whereas here it is still very manageable.

4 Further Work

1. As previously mentioned, one of the main benefits of using a deep learning framework is that we can use GPU acceleration, install cuda on your machine (if you have an Nvidia graphics card) and try and get PyTorch to run. We have set the number of workers to be equal to 1 in this program, try increasing this (this will load data in extra threads on your machine)

2. There are currently many different frameworks available, try to re-implement the code above using the TensorFlow library, do you notice any difference in performance? What is the difference in the lines of code written?
3. We would not normally use this type of network for image recognition, instead we would use a convolutional neural network, look into what a convolutional layer does and implement this with the mnist dataset. How does network performance vary for this?

5 Code for Lab6-1

```
import os
import struct

import numpy as np
import matplotlib.pyplot as plt

MNIST_PATH = '/PATH/TO/YOUR/MNIST/FOLDER'

def load_mnist(path, kind='train'):
    """Load MNIST data from ‘path‘"""
    labels_path = os.path.join(path, '%s-labels-idx1-ubyte' % kind)
    images_path = os.path.join(path, '%s-images-idx3-ubyte' % kind)
    with open(labels_path, 'rb') as lbpath:
        magic, n = struct.unpack('>II', lbpath.read(8))
        labels = np.fromfile(lbpath, dtype=np.uint8)
    with open(images_path, 'rb') as imgpath:
        magic, num, rows, cols = struct.unpack(">IIII",
            imgpath.read(16))
        images = np.fromfile(imgpath,
            dtype=np.uint8).reshape(len(labels), 784)
    return images, labels

X_train, y_train = load_mnist(MNIST_PATH, kind='train')
print('Rows: %d, columns: %d' % (X_train.shape[0], X_train.shape[1]))

fig, ax = plt.subplots(nrows=2, ncols=5, sharex=True, sharey=True,)
ax = ax.flatten()

for i in range(10):
    img = X_train[y_train == i][0].reshape(28, 28)
    ax[i].imshow(img, cmap='Greys', interpolation='nearest')
    ax[0].set_xticks([])
    ax[0].set_yticks([])
```

```
plt.tight_layout()
plt.show()
```

6 Code for Multi Layer Perceptron

```
import numpy as np
from scipy.special import expit
import sys


class NeuralNetMLP(object):
    def __init__(self, n_output, n_features, n_hidden=30,
                 l1=0.0, l2=0.0, epochs=500, eta=0.001,
                 alpha=0.0, decrease_const=0.0, shuffle=True,
                 minibatches=1, random_state=None):
        np.random.seed(random_state)
        self.n_output = n_output
        self.n_features = n_features
        self.n_hidden = n_hidden
        self.w1, self.w2 = self._initialize_weights()
        self.l1 = l1
        self.l2 = l2
        self.epochs = epochs
        self.eta = eta
        self.alpha = alpha
        self.decrease_const = decrease_const
        self.shuffle = shuffle
        self.minibatches = minibatches

    def _encode_labels(self, y, k):
        onehot = np.zeros((k, y.shape[0]))
        for idx, val in enumerate(y):
            onehot[val, idx] = 1.0
        return onehot

    def _initialize_weights(self):
        w1 = np.random.uniform(-1.0, 1.0,
        size=self.n_hidden*(self.n_features + 1))
        w1 = w1.reshape(self.n_hidden, self.n_features + 1)
        w2 = np.random.uniform(-1.0, 1.0,
        size=self.n_output*(self.n_hidden + 1))
        w2 = w2.reshape(self.n_output, self.n_hidden + 1)
        return w1, w2
```

```
def _sigmoid(self, z):
    # expit is equivalent to 1.0/(1.0 + np.exp(-z))
    return expit(z)

def _sigmoid_gradient(self, z):
    sg = self._sigmoid(z)
    return sg * (1 - sg)

def _add_bias_unit(self, X, how='column'):
    if how == 'column':
        X_new = np.ones((X.shape[0], X.shape[1]+1))
        X_new[:, 1:] = X
    elif how == 'row':
        X_new = np.ones((X.shape[0]+1, X.shape[1]))
        X_new[1:, :] = X
    else:
        raise AttributeError('`how` must be `column` or `row`')
    return X_new

def _feedforward(self, X, w1, w2):
    a1 = self._add_bias_unit(X, how='column')
    z2 = w1.dot(a1.T)
    a2 = self._sigmoid(z2)
    a2 = self._add_bias_unit(a2, how='row')
    z3 = w2.dot(a2)
    a3 = self._sigmoid(z3)
    return a1, z2, a2, z3, a3

def _L2_reg(self, lambda_, w1, w2):
    return (lambda_/2.0) * (np.sum(w1[:, 1:] ** 2) \
        + np.sum(w2[:, 1:] ** 2))

def _L1_reg(self, lambda_, w1, w2):
    return (lambda_/2.0) * (np.abs(w1[:, 1:]).sum() \
        + np.abs(w2[:, 1:]).sum())

def _get_cost(self, y_enc, output, w1, w2):
    term1 = -y_enc * (np.log(output))
    term2 = (1 - y_enc) * np.log(1 - output)
    cost = np.sum(term1 - term2)
    L1_term = self._L1_reg(self.l1, w1, w2)
    L2_term = self._L2_reg(self.l2, w1, w2)
    cost = cost + L1_term + L2_term
    return cost
```

```
def _get_gradient(self, a1, a2, a3, z2, y_enc, w1, w2):
    # backpropagation
    sigma3 = a3 - y_enc
    z2 = self._add_bias_unit(z2, how='row')
    sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)
    sigma2 = sigma2[1:, :]
    grad1 = sigma2.dot(a1)
    grad2 = sigma3.dot(a2.T)
    # regularize
    grad1[:, 1:] += (w1[:, 1:] * (self.l1 + self.l2))

    grad2[:, 1:] += (w2[:, 1:] * (self.l1 + self.l2))
    return grad1, grad2

def predict(self, X):
    a1, z2, a2, z3, a3 = self._feedforward(X, self.w1, self.w2)
    y_pred = np.argmax(z3, axis=0)
    return y_pred

def fit(self, X, y, print_progress=False):
    self.cost_ = []
    X_data, y_data = X.copy(), y.copy()
    y_enc = self._encode_labels(y, self.n_output)
    delta_w1_prev = np.zeros(self.w1.shape)
    delta_w2_prev = np.zeros(self.w2.shape)

    for i in range(self.epochs):
        # adaptive learning rate
        self.eta /= (1 + self.decrease_const*i)
        if print_progress:
            sys.stderr.write('\rEpoch: %d/%d' % (i+1, self.epochs))
            sys.stderr.flush()

        if self.shuffle:
            idx = np.random.permutation(y_data.shape[0])
            X_data, y_enc = X_data[idx], y_enc[:,idx]
            mini = np.array_split(range(y_data.shape[0]),
                self.minibatches)

        for idx in mini:
            # feedforward
            a1, z2, a2, z3, a3 = self._feedforward(
            X_data[idx], self.w1, self.w2)
            cost = self._get_cost(y_enc=y_enc[:, idx],
                                  output=a3,
                                  w1=self.w1,
                                  w2=self.w2)
            self.cost_.append(cost)

            # compute gradient via backpropagation
            grad1, grad2 = self._get_gradient(a1=a1, a2=a2,
                                              a3=a3, z2=z2,
                                              y_enc=y_enc[:, idx],
                                              w1=self.w1,
                                              w2=self.w2)

            # update weights
            delta_w1, delta_w2 = self.eta * grad1, self.eta * grad2
            self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
            self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
            delta_w1_prev, delta_w2_prev = delta_w1, delta_w2
    return self
```

7 Code for Lab6-2.py

```
    import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms

from NeuralNetPyTorch import NeuralNet

MNIST_PATH = '/PATH/TO/MNIST'

input_size = 784
hidden_size = 500
output_size = 10
num_epochs = 1
learning_rate = 0.001

train_dataset = torchvision.datasets.MNIST(root=MNIST_PATH, train=True,
        transform=transforms.ToTensor(), download=False)
test_dataset = torchvision.datasets.MNIST(root=MNIST_PATH,
        train=False, transform=transforms.ToTensor(), download=False)
train_loader = torch.utils.data.DataLoader(dataset=train_dataset,
        batch_size=100, shuffle=True)
test_loader = torch.utils.data.DataLoader(dataset=test_dataset,
        batch_size=100, shuffle=False)
```

```
model = NeuralNet(input_size, hidden_size, output_size)

lossFunction = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

total_step = len(train_loader)

for epoch in range(num_epochs):
    for i, (images,labels) in enumerate(train_loader):
        images = images.reshape(-1, 28*28)
        out = model(images)
        loss = lossFunction(out, labels)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if (i+1) % 100 == 0:
            print('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}' .format(epoch+1,
                num_epochs, i+1, total_step, loss.item()))

with torch.no_grad():
    correct = 0
    total = 0
    for images,labels in test_loader:
        images = images.reshape(-1,28*28)
        out = model(images)
        _, predicted = torch.max(out.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    print('Accuracy of the network on the 10000 test images: {} %'
        .format(100 * correct / total))
```

8 NeuralNetPytorch.py

```
import torch.nn as nn

class NeuralNet(nn.Module):
    """A Neural Network with a hidden layer"""
    def __init__(self, input_size, hidden_size, output_size):
        super(NeuralNet, self).__init__()
        self.layer1 = nn.Linear(input_size, hidden_size)
        self.layer2 = nn.Linear(hidden_size, output_size)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        output = self.layer1(x)
        output = self.sigmoid(output)
        output = self.layer2(output)
        return output
```