Coursework 1: ML basics and fully-connected networks

Instructions

Please submit a version of this notebook containing your answers on CATe as *CW1*. Write your answers in the cells below each question.

We recommend that you work on the Ubuntu workstations in the lab. This assignment and all code were only tested to work on these machines. In particular, we cannot guarantee compatibility with Windows machines and cannot promise support if you choose to work on a Windows machine.

You can work from home and use the lab workstations via ssh (for list of machines: [https://www.doc.ic.ac.uk/csg/facilities/lab/workstations) (https://www.doc.ic.ac.uk/csg/facilities/lab/workstations)](https://www.doc.ic.ac.uk/csg/facilities/lab/workstations)).

Once logged in, run the following commands in the terminal to set up a Python environment with all the packages you will need.

```
export PYTHONUSERBASE=/vol/bitbucket/nuric/pypi
export PATH=/vol/bitbucket/nuric/pypi/bin:$PATH
```

Add the above lines to your `.bashrc` to have these enviroment variables set automatically each time you open your bash terminal.

Any code that you submit will be expected to run in this environment. Marks will be deducted for code that fails to run.

Run `jupyter-notebook` in the coursework directory to launch Jupyter notebook in your default browser.

DO NOT attempt to create a virtualenv in your home folder as you will likely exceed your file quota.

DEADLINE: 7pm, Tuesday 5th February, 2019

Part 1

1. Describe two practical methods used to estimate a supervised learning model's performance on unseen data. Which strategy is most commonly used in most deep learning applications, and why?
2. Suppose that you have reason to believe that your multi-layer fully-connected neural network is overfitting. List four things that you could try to improve generalization performance.

**ANSWERS FOR PART 1 IN THIS CELL*

1. Cross Validation and Train / test Split. Cross Validation involves partitioning a sample of data into complementary subsets, performing the analysis on training set, and validating the analysis on the validation set or testing set. And Train / test Split is to split the data into two sets. Train the model on the training set and then test the model on the testing set, and evaluate the performance of that model. Since the training expense is really high when it comes to train the deep learning models, so train / test split may be most commonly used in deep learning applications instead of cross validation

2. Data Dropout, Add more data, Add regularisation and Add data augmentation.

Part 2

1. Why can gradient-based learning be difficult when using the sigmoid or hyperbolic tangent functions as hidden unit activation functions in deep, fully-connected neural networks?
2. Why is the issue that arises in the previous question less of an issue when using such functions as output unit activation functions, provided that an appropriate loss function is used?
3. What would happen if you initialize all the weights to zero in a multi-layer fully-connected neural network and attempt to train your model using gradient descent? What would happen if you did the same thing for a logistic regression model?

ANSWERS FOR PART 2 IN THIS CELL

1. Using sigmoid functions cause vanishing gradients problem, preventing the network to learn further. Besides that, Since tangent functions are like scaled sigmoid functions, the vanishing gradients problem exists too.
2. As the ouput unit activation functions, the sigmoid and tanh intervals are 0 to 1, or -1 to 1 respectively, meaning that these two functions have an advantage in expression of the output layer. Besides that, the vanishing gradients problem is caused by the largest gradient of sigmoid function is 0.25 and all gradients on each layer multiply to nearly zero. If we only use the sigmoid as output unit activation function, we may not encounter with vanishing gradients problem.
3. In neural network, if all the weights are zero, in backpropogation, all the weights of neurons in each layer will stay the same. So the training process will not work. In logistic regression model, because the cost function of logistic regression model is convex, it does not matter where we start. the starting point just changes the number of iterations to reach to that optimal point.

Part 3

In this part, you will use PyTorch to implement and train a multinomial logistic regression model to classify MNIST digits.

Restrictions:

- You must use (but not modify) the code provided in `utils.py` . **This file is deliberately not documented**; read it carefully as you will need to understand what it does to complete the tasks.
- You are NOT allowed to use the `torch.nn` module.

Please insert your solutions to the following tasks in the cells below:

1. Complete the `MultinomialLogisticRegressionClassifier` class below by filling in the missing parts (expected behaviour is prescribed in the documentation):

    - The constructor
    - `forward`
    - `parameters`
    - `l1_weight_penalty`
    - `l2_weight_penalty`

2. The default hyperparameters for `MultilayerClassifier` and `run_experiment` have been deliberately chosen to produce poor results. Experiment with different hyperparameters until you are able to get a test set accuracy above 92% after a maximum of 10 epochs of training. However, DO NOT use the test set accuracy to tune your hyperparameters; use the validation loss / accuracy. You can use any optimizer in `torch.optim`.

In [1]:

```
from utils import *
```

In [2]:

```
# *CODE FOR PART 3.1 IN THIS CELL*
from torch.distributions import normal

class MultinomialLogisticRegressionClassifier:
    def __init__(self, weight_init_sd=100.0):
        """
        Initializes model parameters to values drawn from the Normal
        distribution with mean 0 and standard deviation `weight_init_sd`.
        """
        self.weight_init_sd = weight_init_sd

        #########################################################################
        #                          ** START OF YOUR CODE **
        #########################################################################
        m1 = torch.normal(mean=0.0, std = torch.zeros(784, 10)
                          + self.weight_init_sd)
        self.theta = m1.requires_grad_(True)
        m2 = torch.normal(mean=0.0, std = torch.zeros(10) + self.weight_init_sd)
        self.bias = m2.requires_grad_(True)
        #########################################################################
        #                          ** END OF YOUR CODE **
        #########################################################################

    def __call__(self, *args, **kwargs):
        return self.forward(*args, **kwargs)

    def forward(self, inputs):
        """
        Performs the forward pass through the model.

        Expects `inputs` to be a Tensor of shape (batch_size, 1, 28, 28) containing
        minibatch of MNIST images.

        Inputs should be flattened into a Tensor of shape (batch_size, 784),
        before being fed into the model.

        Should return a Tensor of logits of shape (batch_size, 10).
        """
        #########################################################################
        #                          ** START OF YOUR CODE **
        #########################################################################

        tmp = inputs.view(-1,784)@(self.theta)
        tmp +=self.bias
        return F.log_softmax(tmp,dim=1)
        #########################################################################
        #                          ** END OF YOUR CODE **
        #########################################################################

    def parameters(self):
        """
        Should return an iterable of all the model parameter Tensors.
        """
        #########################################################################
        #                          ** START OF YOUR CODE **
        #########################################################################
        yield self.theta
        yield self.bias
```

```
        ###########################################################################
        #                          ** END OF YOUR CODE **
        ###########################################################################

    def l1_weight_penalty(self):
        """
        Computes and returns the L1 norm of the model's weight vector (i.e. sum
        of absolute values of all model parameters).
        """
        ###########################################################################
        #                          ** START OF YOUR CODE **
        ###########################################################################
        l1_regularization = 0
        l1_regularization +=  torch.sum(torch.abs(self.theta))
        l1_regularization +=  torch.sum(torch.abs(self.bias))
        return l1_regularization
        ###########################################################################
        #                          ** END OF YOUR CODE **
        ###########################################################################

    def l2_weight_penalty(self):
        """
        Computes and returns the L2 weight penalty (i.e.
        sum of squared values of all model parameters).
        """
        ###########################################################################
        #                          ** START OF YOUR CODE **
        ###########################################################################
        l2_regularization = 0
        l2_regularization +=  torch.sum(self.theta * self.theta)
        l2_regularization +=  torch.sum(self.bias * self.bias)
        return torch.sqrt(l2_regularization)
        ###########################################################################
        #                          ** END OF YOUR CODE **
        ###########################################################################
```

In [3]:

```python
# *CODE FOR PART 3.2 IN THIS CELL - EXAMPLE WITH DEFAULT PARAMETERS PROVIDED *

model = MultinomialLogisticRegressionClassifier(weight_init_sd=0.03)
res = run_experiment(
                    model,
                    optimizer=optim.Adam(model.parameters(), lr = 1e-3),
                    train_loader=train_loader_0,
                    val_loader=val_loader_0,
                    test_loader=test_loader_0,
                    n_epochs=10,
                    l1_penalty_coef=0.00003,
                    l2_penalty_coef=0.00003,
                    suppress_output=False
                    )
```

```
Epoch 0: training...
Train set:      Average loss: 0.5726, Accuracy: 0.8610
Validation set: Average loss: 0.3614, Accuracy: 0.9018

Epoch 1: training...
Train set:      Average loss: 0.3347, Accuracy: 0.9084
Validation set: Average loss: 0.3137, Accuracy: 0.9128

Epoch 2: training...
Train set:      Average loss: 0.3047, Accuracy: 0.9162
Validation set: Average loss: 0.2959, Accuracy: 0.9165

Epoch 3: training...
Train set:      Average loss: 0.2913, Accuracy: 0.9196
Validation set: Average loss: 0.2875, Accuracy: 0.9193

Epoch 4: training...
Train set:      Average loss: 0.2826, Accuracy: 0.9216
Validation set: Average loss: 0.2840, Accuracy: 0.9200

Epoch 5: training...
Train set:      Average loss: 0.2769, Accuracy: 0.9232
Validation set: Average loss: 0.2781, Accuracy: 0.9202

Epoch 6: training...
Train set:      Average loss: 0.2730, Accuracy: 0.9237
Validation set: Average loss: 0.2745, Accuracy: 0.9218

Epoch 7: training...
Train set:      Average loss: 0.2699, Accuracy: 0.9256
Validation set: Average loss: 0.2719, Accuracy: 0.9222

Epoch 8: training...
Train set:      Average loss: 0.2669, Accuracy: 0.9261
Validation set: Average loss: 0.2742, Accuracy: 0.9237

Epoch 9: training...
Train set:      Average loss: 0.2649, Accuracy: 0.9272
Validation set: Average loss: 0.2713, Accuracy: 0.9248
```

```
Test set:       Average loss: 0.2672, Accuracy: 0.9249
```

Part 4

In this part, you will use PyTorch to implement and train a multi-layer fully-connected neural network to classify MNIST digits.

Your network must have three hidden layers with 128, 64, and 32 hidden units respectively.

The same restrictions as in Part 3 apply.

Please insert your solutions to the following tasks in the cells below:

1. Complete the `MultilayerClassifier` class below by filling in the missing parts of the following methods (expected behaviour is prescribed in the documentation):
   - The constructor
   - `forward`
   - `parameters`
   - `l1_weight_penalty`
   - `l2_weight_penalty`
2. The default hyperparameters for `MultilayerClassifier` and `run_experiment` have been deliberately chosen to produce poor results. Experiment with different hyperparameters until you are able to get a test set accuracy above 97% after a maximum of 10 epochs of training. However, DO NOT use the test set accuracy to tune your hyperparameters; use the validation loss / accuracy. You can use any optimizer in `torch.optim`.
3. Describe an alternative strategy for initializing weights that may perform better than the strategy we have used here.

In [4]:

```
# *CODE FOR PART 4.1 IN THIS CELL*

class MultilayerClassifier:
    def __init__(self, activation_fun="sigmoid", weight_init_sd=1.0):
        """
        Initializes model parameters to values drawn from the Normal
        distribution with mean 0 and standard deviation `weight_init_sd`.
        """
        super().__init__()
        self.activation_fun = activation_fun
        self.weight_init_sd = weight_init_sd

        if self.activation_fun == "relu":
            self.activation = F.relu
        elif self.activation_fun == "sigmoid":
            self.activation = torch.sigmoid
        elif self.activation_fun == "tanh":
            self.activation = torch.tanh
        else:
            raise NotImplementedError()

        #########################################################################
        #                        ** START OF YOUR CODE **
        #########################################################################

        self.weights1 = torch.normal(mean=0.0, std = torch.zeros(784,128) + self.we
        self.weights1 = self.weights1.requires_grad_(True)
        self.weights2 = torch.normal(mean=0.0, std = torch.zeros(128,64) + self.wei
        self.weights2 = self.weights2.requires_grad_(True)
        self.weights3 = torch.normal(mean=0.0, std = torch.zeros(64,32) + self.weig
        self.weights3 = self.weights3.requires_grad_(True)
        self.weights4 = torch.normal(mean=0.0, std = torch.zeros(32,10) + self.weig
        self.weights4 = self.weights4.requires_grad_(True)

        self.bias1 = torch.normal(mean=0.0, std = torch.zeros(128) + self.weight_in
        self.bias1 = self.bias1.requires_grad_(True)
        self.bias2 = torch.normal(mean=0.0, std = torch.zeros(64) + self.weight_init
        self.bias2 = self.bias2.requires_grad_(True)
        self.bias3 = torch.normal(mean=0.0, std = torch.zeros(32) + self.weight_init
        self.bias3 = self.bias3.requires_grad_(True)
        self.bias4 = torch.normal(mean=0.0, std = torch.zeros(10) + self.weight_init
        self.bias4 = self.bias4.requires_grad_(True)
        #########################################################################
        #                        ** END OF YOUR CODE **
        #########################################################################

    def __call__(self, *args, **kwargs):
        return self.forward(*args, **kwargs)

    def forward(self, inputs):
        """
        Performs the forward pass through the model.

        Expects `inputs` to be Tensor of shape (batch_size, 1, 28, 28) containing
        minibatch of MNIST images.

        Inputs should be flattened into a Tensor of shape (batch_size, 784),
        before being fed into the model.
```

```
        Should return a Tensor of logits of shape (batch_size, 10).
        """
        ######################################################################
        #                        ** START OF YOUR CODE **
        ######################################################################
        tmp = inputs.view(-1,784)@(self.weights1) + self.bias1

        tmp = self.activation(tmp)
#         print(tmp.shape,self.bias2.shape)
        tmp = tmp.view(-1,128)@(self.weights2) + self.bias2
#         print(tmp.shape,self.bias2.shape)
        tmp = self.activation(tmp)
#         print(tmp.shape,self.bias3.shape)
        tmp = tmp.view(-1,64)@(self.weights3) + self.bias3
#         print(tmp.shape,self.bias3.shape)
        tmp = self.activation(tmp)
#         print(tmp.shape,self.bias4.shape)
        tmp = tmp.view(-1,32)@(self.weights4)


        tmp = self.activation(tmp)
        return F.log_softmax(tmp, dim=1)

        ######################################################################
        #                        ** END OF YOUR CODE **
        ######################################################################

    def parameters(self):
        """
        Should return an iterable of all the model parameter Tensors.
        """
        ######################################################################
        #                        ** START OF YOUR CODE **
        ######################################################################
        yield self.weights1
        yield self.bias1
        yield self.weights2
        yield self.bias2
        yield self.weights3
        yield self.bias3
        yield self.weights4
        yield self.bias4
        ######################################################################
        #                        ** END OF YOUR CODE **
        ######################################################################


    def l1_weight_penalty(self):
        """
        Computes and returns the L1 norm of the model's weight vector (i.e. sum
        of absolute values of all model parameters).
        """
        ######################################################################
        #                        ** START OF YOUR CODE **
        ######################################################################
        l1_regularization =  0
        l1_regularization +=  torch.sum(torch.abs(self.weights1))
        l1_regularization +=  torch.sum(torch.abs(self.weights2))
        l1_regularization +=  torch.sum(torch.abs(self.weights3))
```

```python
        l1_regularization +=  torch.sum(torch.abs(self.weights4))
        l1_regularization +=  torch.sum(torch.abs(self.bias1))
        l1_regularization +=  torch.sum(torch.abs(self.bias2))
        l1_regularization +=  torch.sum(torch.abs(self.bias3))
        l1_regularization +=  torch.sum(torch.abs(self.bias4))
        return l1_regularization
        ########################################################################
        #                         ** END OF YOUR CODE **
        ########################################################################

    def l2_weight_penalty(self):
        """
        Computes and returns the L2 weight penalty (i.e.
        sum of squared values of all model parameters).
        """
        ########################################################################
        #                         ** START OF YOUR CODE **
        ########################################################################
        l2_regularization =  0
        l2_regularization +=  torch.sum(self.weights1*self.weights1)
        l2_regularization +=  torch.sum(self.weights2*self.weights2)
        l2_regularization +=  torch.sum(self.weights3*self.weights3)
        l2_regularization +=  torch.sum(self.weights4*self.weights4)
        l2_regularization +=  torch.sum(self.bias1*self.bias1)
        l2_regularization +=  torch.sum(self.bias2*self.bias2)
        l2_regularization +=  torch.sum(self.bias3*self.bias3)
        l2_regularization +=  torch.sum(self.bias4*self.bias4)
        return l2_regularization
        ########################################################################
        #                         ** END OF YOUR CODE **
        ########################################################################
```

In [6]:

```
# *CODE FOR PART 4.2 IN THIS CELL - EXAMPLE WITH DEFAULT PARAMETERS PROVIDED *

model = MultilayerClassifier(activation_fun='relu', weight_init_sd=0.05)
res = run_experiment(
    model,
    optimizer=optim.Adam(model.parameters(), lr=1e-3),
    train_loader=train_loader_0,
    val_loader=val_loader_0,
    test_loader=test_loader_0,
    n_epochs=10,
    l1_penalty_coef=0.00003,
    l2_penalty_coef=0.00003,
    suppress_output=False
)
```

```
Epoch 0: training...
Train set:      Average loss: 0.6943, Accuracy: 0.7800
Validation set: Average loss: 0.4278, Accuracy: 0.8535

Epoch 1: training...
Train set:      Average loss: 0.3638, Accuracy: 0.8769
Validation set: Average loss: 0.1677, Accuracy: 0.9522

Epoch 2: training...
Train set:      Average loss: 0.1455, Accuracy: 0.9575
Validation set: Average loss: 0.1262, Accuracy: 0.9620

Epoch 3: training...
Train set:      Average loss: 0.1137, Accuracy: 0.9665
Validation set: Average loss: 0.1172, Accuracy: 0.9643

Epoch 4: training...
Train set:      Average loss: 0.0950, Accuracy: 0.9717
Validation set: Average loss: 0.1083, Accuracy: 0.9687

Epoch 5: training...
Train set:      Average loss: 0.0807, Accuracy: 0.9761
Validation set: Average loss: 0.0894, Accuracy: 0.9747

Epoch 6: training...
Train set:      Average loss: 0.0735, Accuracy: 0.9777
Validation set: Average loss: 0.1170, Accuracy: 0.9647

Epoch 7: training...
Train set:      Average loss: 0.0649, Accuracy: 0.9804
Validation set: Average loss: 0.0972, Accuracy: 0.9693

Epoch 8: training...
Train set:      Average loss: 0.0605, Accuracy: 0.9812
Validation set: Average loss: 0.1007, Accuracy: 0.9703

Epoch 9: training...
Train set:      Average loss: 0.0530, Accuracy: 0.9843
Validation set: Average loss: 0.0935, Accuracy: 0.9728

Test set:       Average loss: 0.0877, Accuracy: 0.9745
```

ANSWERS FOR PART 4.3 IN THIS CELL

Instead of using a constant standard deviation for all the weights in different layers, we may choose the standard deviation according to the number of input features. $w = U([0, n]) * sqrt(2.0/n)$ where n is the number of inputs of the neural network[1].

[1] He, K., Zhang, X., Ren, S. and Sun, J., 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In Proceedings of the IEEE international conference on computer vision (pp. 1026-1034).