questions

January 31, 2019

1 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).

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

1.1 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. There are k-fold cross-validation and hold-out validation

- K-fold Cross-Validation: K-fold Cross-Validation assesses how well the result of statistic analysis will generalize a independent dataset. Cross-Validation contain multiple rounds, and each round involves partitioning the samples of data into several complimentary subsets, left one subset alone and perform training on the rest subsets, and validating the training result by predicting test result on the left-alone subset. Each round we choose one subset toleft alone, repeat rounds until every subset has been used as test set.

- Hold-Out Method(Validation): It is a kind of simplified Cross Validation. Randomly split dataset into a training and a testing partitions. Use training set to train and tset set to test the performance This will result a very quick estimate of performance.

- K-fold Cross validation is the most commonly used, because hold-out method is to simple to give a good estimate for the neural network, and some other method like left-one-out method and bootstrapping are only work well on small dataset. Besides, k-fold cross validation gives a very good valitation and work well in hyper-parameters choosing.

2.
- add drop out layers within the network
- add L2-norm weight penalty
- reduce the depth or width of the network (thus reduce # parameters, and complexity of the model)
- add more data to training

1.2 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. As networks' weight received an update according to the partial derivative of the error function, weight will be update to a portion to the derivative of the activation function. Using sigmoid or hyperbolic tangent as activation functions cause a problem that when the back-propagated error function value is close to 0 or 1, the gradient will be vanishingly small, preventing weight from changing its value. This is called gradient vanishing problem.
And this effect will be really a problem of the hidden unit, the gradients of the network's output with respect to the parameters in the early layers become extremely small, because at each hidden layer, the activation functions are trying to map large input into small region, thus large change in the input will result rather small change of the output. For earlier layer, zoom-in may happen several times. This make training difficult.
2. As descussed in previous question, when we only consider the output layer, the zoom-in effect only take place once, which cause less of an issue.
3.

* In gradient descent, if zero is set to all weights, the training may tend to stuck on local minima, and all neuron tend to follow same gradient and end up with doing the same things.
* In logistic regression, the globle minima may always be reached as it is convex function. And it is not a issue as logistic regression is simply y = Wx + b.

1.3 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]:
```
from torch.distributions import normal
import torch.autograd as autograd

# *CODE FOR PART 3.1 IN THIS CELL*
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 **
        ###########################################################################
        m = normal.Normal(0, self.weight_init_sd)
        self.param = autograd.Variable(m.sample((784, 10)), requires_grad=True)
        self.bias = autograd.Variable(m.sample((1, 10)), requires_grad=True)

        self.params = [self.param, self.bias]
        ###########################################################################
        #                         ** 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 **
    ###########################################################################
    batch_size, _, _, _ = inputs.size()
    inputs = inputs.view(batch_size, 784)
    z = torch.mm(inputs, self.param) + self.bias
    # z = xTw
    z -= torch.max(z, dim=1)[0].view(batch_size, -1)
    # z = z - max(z) for each line
    exp = torch.exp(z)
    # e^z
    sums = torch.sum(exp, dim=1).view(batch_size, -1)
    # sum(e^zi)
    y = z - torch.log(sums)
    # y = z - log(sum(e^zi))
    return y
    ###########################################################################
    #                         ** END OF YOUR CODE **
    ###########################################################################

def parameters(self):
    """
    Should return an iterable of all the model parameter Tensors.
    """
    ###########################################################################
    #                         ** START OF YOUR CODE **
    ###########################################################################
    for p in self.params:
        yield self.param
    ###########################################################################
    #                         ** 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 **
            ###########################################################################
            return torch.norm(self.param, p=1) + torch.norm(self.bias, p=1)
            ###########################################################################
            #                          ** 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 **
            ###########################################################################
            return torch.norm(self.param, p=2) + torch.norm(self.bias, p=2)
            ###########################################################################
            #                          ** END OF YOUR CODE **
            ###########################################################################
```

In [3]:
```
# *CODE FOR PART 3.2 IN THIS CELL - EXAMPLE WITH DEFAULT PARAMETERS PROVIDED *
model = MultinomialLogisticRegressionClassifier(weight_init_sd=0.35)
res = run_experiment(
    model,
    optimizer=optim.SGD(model.parameters(), 0.075, momentum=0.65),
    train_loader=train_loader_0,
    val_loader=val_loader_0,
    test_loader=test_loader_0,
    n_epochs=8,
    l1_penalty_coef=0.0,
    l2_penalty_coef=0.0035,
    suppress_output=False
)
```

```
Epoch 0: training...
Train set:        Average loss: 0.4914, Accuracy: 0.8575
Validation set:         Average loss: 0.3654, Accuracy: 0.8938

Epoch 1: training...
Train set:        Average loss: 0.3380, Accuracy: 0.9029
Validation set:         Average loss: 0.3223, Accuracy: 0.9070
```

```
Epoch 2: training...
Train set:        Average loss: 0.3152, Accuracy: 0.9112
Validation set:        Average loss: 0.3116, Accuracy: 0.9127

Epoch 3: training...
Train set:        Average loss: 0.3038, Accuracy: 0.9141
Validation set:        Average loss: 0.3128, Accuracy: 0.9095

Epoch 4: training...
Train set:        Average loss: 0.2966, Accuracy: 0.9167
Validation set:        Average loss: 0.3006, Accuracy: 0.9145

Epoch 5: training...
Train set:        Average loss: 0.2915, Accuracy: 0.9177
Validation set:        Average loss: 0.2950, Accuracy: 0.9162

Epoch 6: training...
Train set:        Average loss: 0.2882, Accuracy: 0.9195
Validation set:        Average loss: 0.2923, Accuracy: 0.9180

Epoch 7: training...
Train set:        Average loss: 0.2848, Accuracy: 0.9200
Validation set:        Average loss: 0.2859, Accuracy: 0.9210

Test set:        Average loss: 0.2814, Accuracy: 0.9216
```

1.4 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 **
                ########################################################################
                m = normal.Normal(0, self.weight_init_sd)
                input_layer = 784
                layer1 = 128
                layer2 = 64
                layer3 = 32
                layer4 = 10
                self.param1 = autograd.Variable(m.sample((input_layer, layer1)),
                                                requires_grad=True)
                self.param2 = autograd.Variable(m.sample((layer1, layer2)),
                                                requires_grad=True)
                self.param3 = autograd.Variable(m.sample((layer2, layer3)),
                                                requires_grad=True)
                self.param4 = autograd.Variable(m.sample((layer3, layer4)),
                                                requires_grad=True)
                self.bias1 = autograd.Variable(m.sample((1, layer1)),
                                               requires_grad=True)
                self.bias2 = autograd.Variable(m.sample((1, layer2)),
                                               requires_grad=True)
                self.bias3 = autograd.Variable(m.sample((1, layer3)),
                                               requires_grad=True)
                self.bias4 = autograd.Variable(m.sample((1, layer4)),
```

```
                                    requires_grad=True)
        self.params = [self.param1, self.param2, self.param3, self.param4,
                       self.bias1, self.bias2, self.bias3, self.bias4]
        ############################################################################
        #                          ** 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 **
        ############################################################################
        batch_size, _, _, _ = inputs.size()
        inputs = inputs.view(batch_size, 784)
        z0 = torch.mm(inputs, self.param1) + self.bias1
        y0 = self.activation(z0)    #128

        z1 = torch.mm(y0, self.param2) + self.bias2
        y1 = self.activation(z1)    #64

        z2 = torch.mm(y1, self.param3) + self.bias3
        y2 = self.activation(z2)    #32

        z3 = torch.mm(y2, self.param4) + self.bias4
        z3 -= torch.max(z3, dim=1)[0].view(batch_size, -1)
        # z = z - max(z) for each line
        exp = torch.exp(z3)
        # e^z
        sums = torch.sum(exp, dim=1).view(batch_size, -1)
        # sum(e^zi)
        y3 = z3 - torch.log(sums)
        # y = z - log(sum(e^zi))
        # 10
        return y3
        ############################################################################
```

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

    def parameters(self):
        """
        Should return an iterable of all the model parameter Tensors.
        """
        ############################################################################
        #                           ** START OF YOUR CODE **
        ############################################################################
        for p in self.params:
            yield p
        ############################################################################
        #                           ** 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 **
        ############################################################################
        res = 0
        for x in self.params:
            res += torch.norm(x, p=1)
        return res
        ############################################################################
        #                           ** 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 **
        ############################################################################
        res = 0
        for x in self.params:
            res += torch.norm(x, p=2)
        return res
        ############################################################################
        #                           ** END OF YOUR CODE **
        ############################################################################
```

In [5]:
```
# *CODE FOR PART 4.2 IN THIS CELL - EXAMPLE WITH DEFAULT PARAMETERS PROVIDED *

        model = MultilayerClassifier(activation_fun='relu', weight_init_sd=0.20)
        res = run_experiment(
            model,
            optimizer=optim.SGD(model.parameters(), 0.073, momentum=0.7),
            train_loader=train_loader_0,
            val_loader=val_loader_0,
            test_loader=test_loader_0,
            n_epochs=9,
            l1_penalty_coef=0.0,
            l2_penalty_coef=0.014,
            suppress_output=False
        )
```

```
Epoch 0: training...
Train set:        Average loss: 0.3336, Accuracy: 0.8964
Validation set:        Average loss: 0.1752, Accuracy: 0.9468

Epoch 1: training...
Train set:        Average loss: 0.1589, Accuracy: 0.9522
Validation set:        Average loss: 0.1902, Accuracy: 0.9410

Epoch 2: training...
Train set:        Average loss: 0.1262, Accuracy: 0.9629
Validation set:        Average loss: 0.1285, Accuracy: 0.9590

Epoch 3: training...
Train set:        Average loss: 0.1077, Accuracy: 0.9681
Validation set:        Average loss: 0.1149, Accuracy: 0.9653

Epoch 4: training...
Train set:        Average loss: 0.0986, Accuracy: 0.9711
Validation set:        Average loss: 0.1124, Accuracy: 0.9645

Epoch 5: training...
Train set:        Average loss: 0.0900, Accuracy: 0.9737
Validation set:        Average loss: 0.1197, Accuracy: 0.9635

Epoch 6: training...
Train set:        Average loss: 0.0845, Accuracy: 0.9755
Validation set:        Average loss: 0.1084, Accuracy: 0.9663

Epoch 7: training...
Train set:        Average loss: 0.0829, Accuracy: 0.9753
Validation set:        Average loss: 0.0963, Accuracy: 0.9698

Epoch 8: training...
```

```
Train set:        Average loss: 0.0776, Accuracy: 0.9777
Validation set:        Average loss: 0.0996, Accuracy: 0.9715

Test set:        Average loss: 0.0903, Accuracy: 0.9720
```

ANSWERS FOR PART 4.3 IN THIS CELL

3. He initialization Simply multiply random initialization for layer l with sqrt(2 / size_of_layer_l-1) This work well with sigmoid activation function