Regularization in Neural Network

BIS557 Final Project

Siqiang SU

1 Introduction

In traditional machine learning, the problem of over-fitting is a critical consideration when we build our model. We have the bias-variance trade off:

$$\begin{aligned} \text{Err}(x) &= (E[\hat{f}(x)] - f(x))^2 + E\left[(\hat{f}(x) - E[\hat{f}(x)])^2\right] + \sigma_e^2 \\ \text{Err}(x) &= \text{Bias}^2 + \text{Variance} \ + \ \text{Irreducible Error} \end{aligned} \tag{1}$$

Intuitively, it conveys that if our model is too complex, we may obtain an extremely well fit for the training data (nearly 100% accuracy), but the testing accuracy is poor. This means our model fails to generalize since we even fit the noise in the training data perfectly. When it comes to deep learning field, we still need to pay attention to the over-fitting problem.

There are several ways to handle the over-fitting problem: getting more training data, reduce the model complexity, and regularization. In deep learning, we do not want to reduce our model complexity since it will negative the power. Hence, in this project, different regularization methods will be implemented in the fully connected neural network, and their performance will be measured.

2 Background

2.1 Datasets

The benchmark dataset MNIST will be used.The MNIST database of handwritten digits has a training set of 60,000 examples, and a test set of 10,000 examples while each example is a 28x28 black-white (1 color channel) image. Below are some selected images from the dataset.

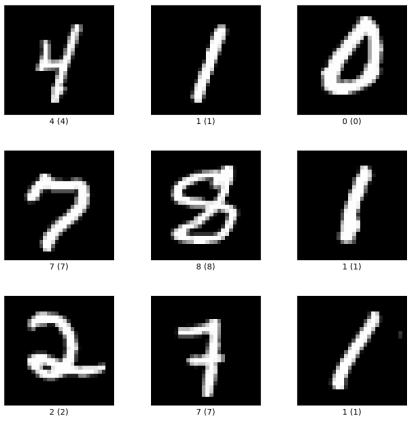


Figure 1: Samples of MNIST dataset

2.2 Method Description

In this project, we will mainly discuss 3 regularization methods: L-1 regularization, L-2 regularization and dropout. Suppose we have the loss function C_0, then the L-2 regularization has the following form:

$$C = C_0 + \frac{\lambda}{2n} \sum_w w^2 \tag{2}$$

We can see that regularization is a compromise between finding small weights and minimize cost functions. After taking the derivative, we have the update rule:

$$w \to w - \eta \frac{\partial C_0}{\partial w} - \frac{\eta \lambda}{n} w \tag{3}$$

We can see that our weight is decayed by factor $1 - \frac{\eta \lambda}{n}$. Hence, L-2 regularization also called weight decay.

For L-1 regularization, we have:

$$C = C_0 + \frac{\lambda}{n} \sum_w |w| \tag{4}$$

If we take the derivative, we have the update rule:

$$w \to w - \frac{\lambda \eta}{n} \operatorname{sgn}(w) - \eta \frac{\partial C_0}{\partial w} \tag{5}$$

We can see that L-1 regularization shifts the weight by a constant amount. Hence, if $|w|$ is large, L-1 shrinks the weights much less than L-2. If $|w|$ is small, L-1 shrinks the weights much more than L-2. We say that L-1 can generate sparse solutions if λ is big.

Unlike L-1 and L-2 regularization which directly modify the loss function, with dropout, we modify the neural network instead. We start by randomly and temporarily deleting a proportion of neurons (controlled by the probability p), then we forward propagate x and back-propagate to get the gradient. Then we update the weights and bias based on this gradient on a batch. By repeating above procedures, we will get weights and bias based on different subset of neurons of our full model. This is equivalent to averaging over multiple neuron networks which can reduce the over-fitting.

2.3 Metrics for Measuring the Performance

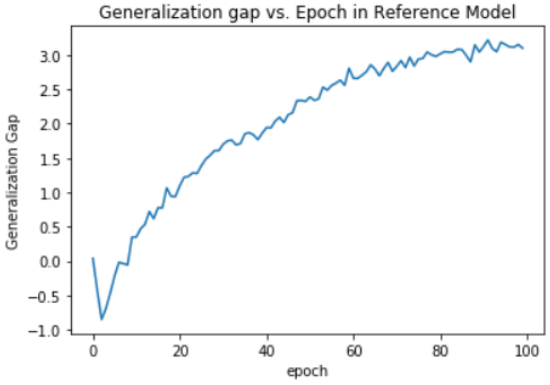
The plot for training and testing accuracy against epochs can reveal whether there is an over-fitting problem. If the training accuracy keeps increasing but the testing accuracy becomes flat, we can say that there is an over-fitting problem. Besides the training and testing accuracy, we also compute the generalization gap which is defined as the difference between a model's performance on training data and its performance on unseen data drawn from the same distribution. Our aim is to reduce the generalization gap to make our model generalize better.

3 Models and Results

3.1 Reference Model Without Regularization

We build a neuron network with 2 hidden layers: Input layers has 784 neurons, first hidden layer has 256 neurons, second hidden layer has 256 neurons, then the output layer has 10 neurons. The activation function is sigmoid function, and we apply log-softmax function on the output layer. We train the model for 100 epochs using stochastic gradient descent (SGD) and we plot the training accuracy and testing accuracy in Figure 2. We can see that after around 30 epochs, the testing accuracy becomes flat (around 94%) while the training accuracy still goes up. The generalization gap is also large (about 3%).This indicates that this model is over-fitted and regularization is needed to make the generalization gap smaller.

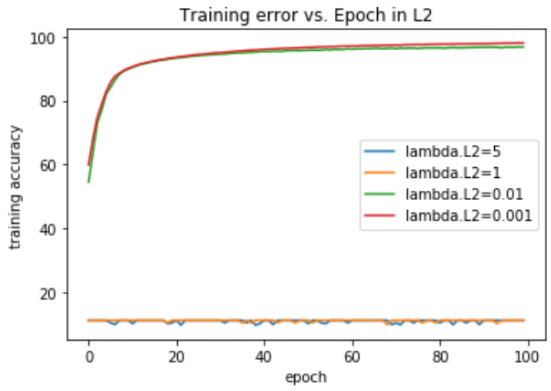
(a) Accuracy vs. Epoch for Reference Model



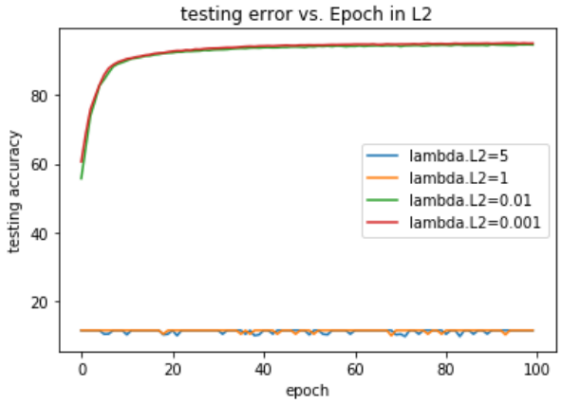
(b) Generalization Gap

3.2 L-2 Regularization

We apply L-2 regularization based on the reference model. By the derivation of the derivative of L-2 regularization, we know that it is equivalent to the weight decay where the magnitude of decay is controlled by λ. To implement L-2 regularization, we set the weight decay parameter as λ in the SGD optimizer. Here we test 4 different λ values: $\lambda \in \{5, 1, 0.01, 0.001\}$. The performance is summarized for each λ in the following 3 plots and 1 table. In the table, the training accuracy, testing accuracy and generalization gap at the 100 epoch are recorded.



(a) Training Accuracy vs. Epoch for L-2 Model



(b) Testing Accuracy vs. Epoch for L-2 Model

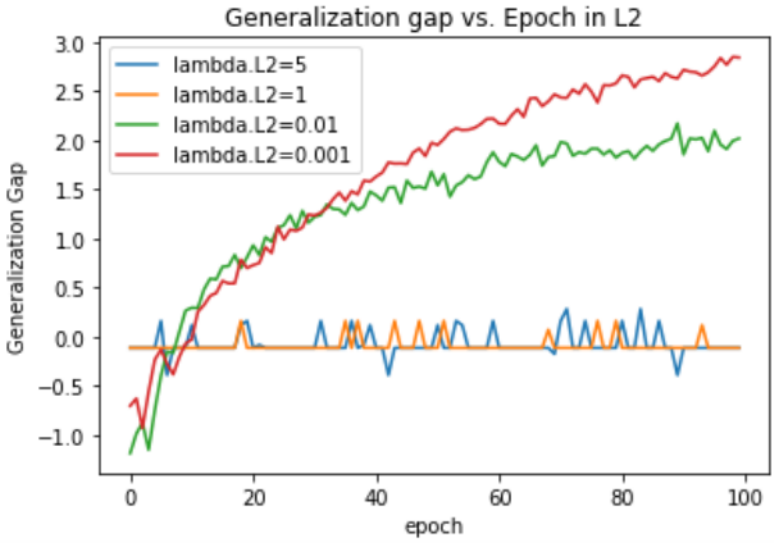


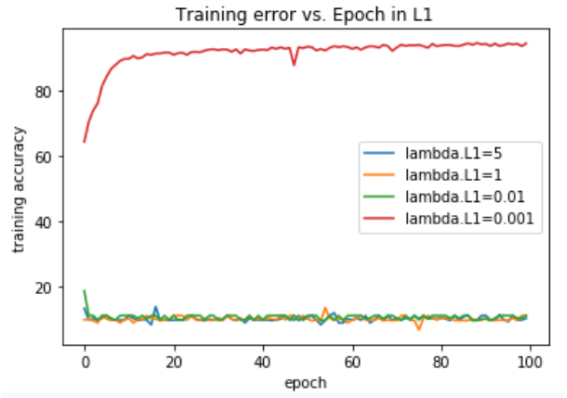
Figure 4: Generalization Gap for L-2 Regularization

λ	epoch	training accuracy (%)	testing accuracy (%)	generalization gap (%)
5	100	11.24	11.35	-0.11
1	100	11.24	11.35	-0.11
0.01	100	96.81	94.79	**2.02**
0.001	100	98.04	95.20	2.84

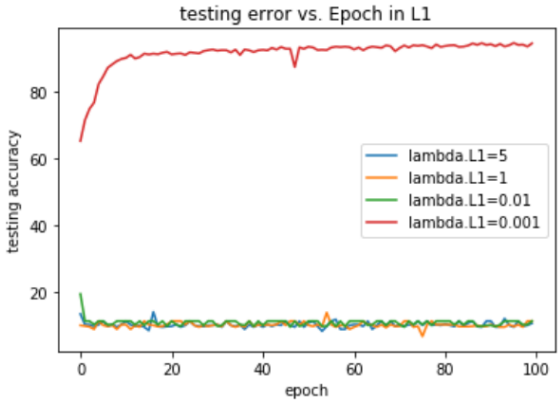
From the plot of testing and training accuracy, we can see that when $\lambda = 1$ and $\lambda = 5$, the accuracy is very poor and it is close to 10-class random guessing. This is because we penalize the weight too much and the model now is under-fitting. When λ becomes small, the accuracy increases significantly. We can see that $\lambda = 0.01$ gives the smallest generalization gap while preserves high accuracy in both training and testing set.

3.3 L-1 Regularization

We apply L-1 regularization to the reference model. To implement L-1 regularization, we add a new penalty component $\lambda \sum_w |w|$ to the original loss function and then implement back-propagation to get gradients based on the new loss function. Here we test 4 different λ values:$\lambda \in \{5, 1, 0.01, 0.001\}$. The performance is summarized for each λ in the following 3 plots and 1 table. In the table, the training accuracy, testing accuracy and generalization gap at the 100 epoch are recorded.



(a) Training Accuracy vs. Epoch for L-1 Model



(b) Testing Accuracy vs. Epoch for L-1 Model

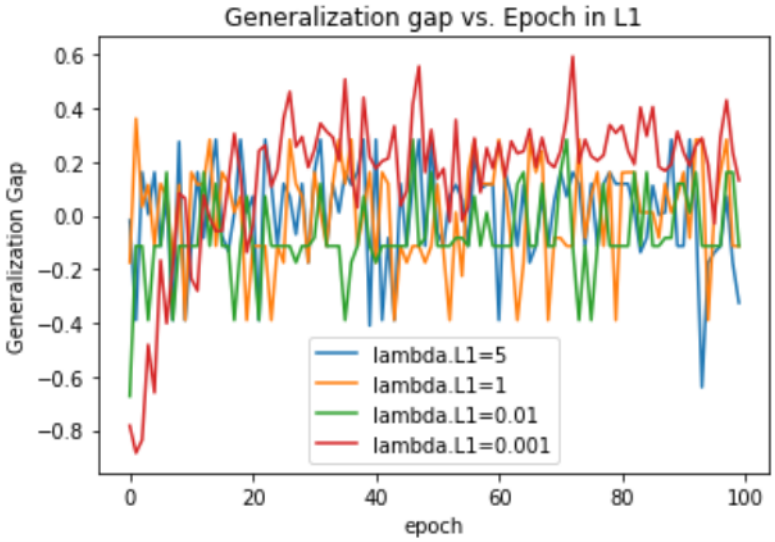


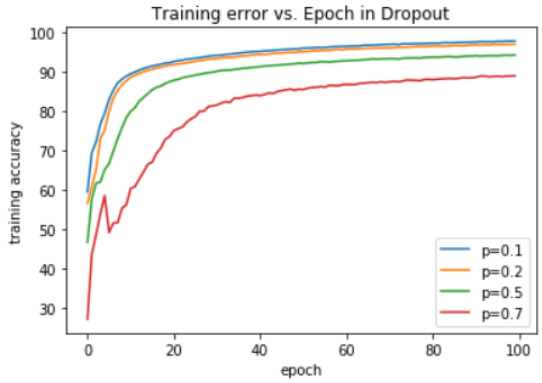
Figure 6: Generalization Gap for L-1 Regularization

λ	epoch	training accuracy (%)	testing accuracy (%)	generalization gap (%)
5	100	10.39	10.71	-0.32
1	100	11.24	11.35	-0.11
0.01	100	11.24	11.35	-0.11
0.001	100	94.48	94.35	0.13

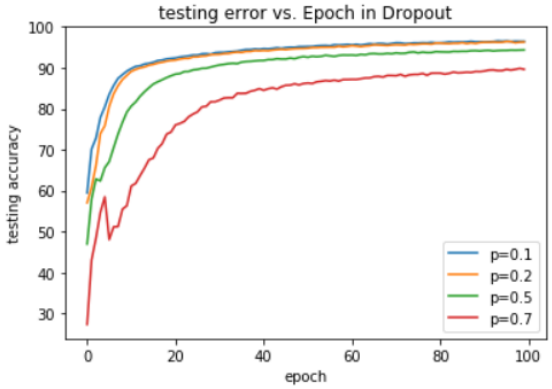
From the plot of testing and training accuracy, we can see that when $\lambda = 5$,$\lambda = 1$ and $\lambda = 0.01$, the accuracy is very poor and it is close to 10-class random guessing. Compared to L-2, we can see that L-1 shrinks the weights more in this model. This is probably because the $|w|$ in this model is small. When $\lambda = 0.001$, the accuracy increases significantly. We can see that $\lambda = 0.001$ gives the small generalization gap while preserves high accuracy in both training and testing set.

3.4 Dropout

Here we implement dropout method to the reference model. We add a dropout layer nn.Dropout(p) with a probability p. We drop the weights at each hidden layer probability p. We test different probabilities: $p \in \{0.1, 0.2, 0.5, 0.7\}$. The performance is summarized for each p in the following 3 plots and 1 table. In the table, the training accuracy, testing accuracy and generalization gap at the 100 epoch are recorded.



(a) Training Accuracy vs. Epoch for Dropout Model



(b) Testing Accuracy vs. Epoch for Dropout Model

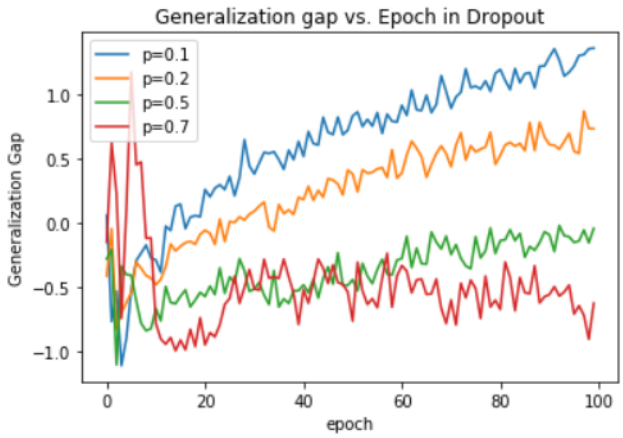


Figure 8: Generalization Gap for Dropout Regularization

p	epoch	training accuracy (%)	testing accuracy (%)	generalization gap (%)
0.1	100	97.84	96.48	1.36
0.2	100	97.04	96.31	0.73
0.5	100	94.33	94.37	-0.04
0.7	100	89.02	89.64	-0.62

Unlike the L-1 and L-2 regularization where the accuracy differs drastically between different λ, the accuracy in dropout method is gradually increasing when the p goes to 0. When p increases, more weights are dropped hence the training accuracy also drops. In the summary table, we can observe that the dropout method gives the highest testing accuracy among these regularization methods.

4 Conclusion

In this section, we combine the best results from above 3 regularization methods and compared their performance to the reference model. We can observe that the training accuracy of reference model is highest. This makes sense since the reference model is the most complicated one and thus it over-fits the data. The L-1 and L-2 regularization give the lower training accuracy and lower testing accuracy due to the simplified model. We should run these 2 models for more

Methods	penalty parameter	training accuracy (%)	testing accuracy (%)	generalization gap (%)
Reference	NA	98.11	95.00	3.11
L-2	$\lambda = 0.01$	96.81	94.79	2.02
L-1	$\lambda = 0.001$	94.48	94.35	0.13
Dropout	$p = 0.2$	97.04	**96.31**	0.73

epochs to get compatible results. Nevertheless, they give smaller generalization gaps compared to reference model. This means L-1 and L-2 regularization can alleviate the over-fitting problem. The dropout method is the best in this setting. It achieves both highest testing accuracy (96.31%) among all models and achieves a smaller generalization gap compared to reference model.

5 Code Appendix

All the results are obtained from PyTorch code. The PyTorch code is on the following pages.

Reference Model Part

December 16, 2020

[2]:
```
import torch

import torch.nn as nn  # neural network modules
import torch.nn.functional as F  # activation functions
import torch.optim as optim  # optimizer
from torch.autograd import Variable # add gradients to tensors
from torch.nn import Parameter # model parameter functionality

import torchvision
import torchvision.datasets as datasets

import numpy as np
import matplotlib.pyplot as plt

torch.manual_seed(42)

# ###################################
# import data
# ###################################
# download the MNIST dataset
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=None)
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=None)


# separate into data and labels
# training data
train_data = mnist_trainset.data.to(dtype=torch.float32)
train_data = train_data.reshape(-1, 784)
train_labels = mnist_trainset.targets.to(dtype=torch.long)

print("train data shape: {}".format(train_data.size()))
print("train label shape: {}".format(train_labels.size()))

# testing data
test_data = mnist_testset.data.to(dtype=torch.float32)
```

```
test_data = test_data.reshape(-1, 784)
test_labels = mnist_testset.targets.to(dtype=torch.long)

print("test data shape: {}".format(test_data.size()))
print("test label shape: {}".format(test_labels.size()))

# load into torch datasets
train_dataset = torch.utils.data.TensorDataset(train_data, train_labels)
test_dataset = torch.utils.data.TensorDataset(test_data, test_labels)

# ###################################
# set hyperparameters
# ###################################
# Parameters
learning_rate = 0.005
num_epochs = 100
batch_size = 128

# Network Parameters
n_hidden_1 = 256  # 1st layer number of neurons
n_hidden_2 = 256  # 2nd layer number of neurons
num_input = 784  # MNIST data input (img shape: 28*28)
num_classes = 10  # MNIST total classes (0-9 digits)

# ###################################
# defining the model
# ###################################

# reference model
class FCNN(nn.Module):
    def __init__(self, input_dim, output_dim):
        super(FCNN, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim

        self.layer1 = nn.Linear(input_dim, n_hidden_1)
        self.layer2 = nn.Linear(n_hidden_1, n_hidden_2)
        self.layer3 = nn.Linear(n_hidden_2, num_classes)

        self.nonlin1 = nn.Sigmoid()
        self.nonlin2 = nn.Sigmoid()


    def forward(self, x):
        h1 = self.nonlin1(self.layer1(x))
        h2 = self.nonlin2(self.layer2(h1))
```

```
        output = self.layer3(h2)

        return F.log_softmax(output,dim=1)


# ###################################
# helper functions
# ###################################
def get_accuracy(output, targets):
    """calculates accuracy from model output and targets
    """
    output = output.detach()
    predicted = output.argmax(-1)
    correct = (predicted == targets).sum().item()

    accuracy = correct / output.size(0) * 100

    return accuracy


def to_one_hot(y, c_dims=10):
    """converts a N-dimensional input to a NxC dimnensional one-hot encoding
    """
    y_tensor = y.data if isinstance(y, Variable) else y
    y_tensor = y_tensor.type(torch.LongTensor).view(-1, 1)
    c_dims = c_dims if c_dims is not None else int(torch.max(y_tensor)) + 1
    y_one_hot = torch.zeros(y_tensor.size()[0], c_dims).scatter_(1, y_tensor, 1)
    y_one_hot = y_one_hot.view(*y.shape, -1)
    return Variable(y_one_hot) if isinstance(y, Variable) else y_one_hot

# ###################################
# creata dataloader
# ###################################
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
                                          shuffle=True)


# ###################################
# main training function
# ###################################
def train(lossflag):

    model = FCNN(num_input, num_classes)
    optimizer = optim.SGD(model.parameters(), lr=learning_rate)

    # initialize loss list
    metrics = [[0, 0]]
```

```
    # iterate over epochs
    for ep in range(num_epochs):
        model.train()

        # iterate over batches
        for batch_indx, batch in enumerate(trainloader):

            # unpack batch
            data, labels = batch


            # get predictions from model
            pred = model(data)

            if lossflag == 1:
                labels = to_one_hot(labels,c_dims=10) #change labels to 10␣
↪vector
                loss_fn=nn.MSELoss()
            elif lossflag == 2:
                loss_fn=nn.CrossEntropyLoss()
            elif lossflag==3:
                loss_fn=nn.NLLLoss()

            loss = loss_fn(pred,labels)


            # print loss every 100 batches
            #if batch_indx % 100 == 0:
                #print("loss {:.3f} at batch {} epoch {}".format(loss.item(),␣
↪batch_indx, ep))

            # backpropagate the loss
            loss.backward()

            # update parameters
            optimizer.step()

            # reset gradients
            optimizer.zero_grad()

        # compute full train and test accuracies
        # every epoch
        model.eval() # model will not calculate gradients for this pass
        train_ep_pred = model(train_data)
        test_ep_pred = model(test_data)
```

```
        train_accuracy = get_accuracy(train_ep_pred, train_labels)
        test_accuracy = get_accuracy(test_ep_pred, test_labels)

        print("train acc: {}\t test acc: {}\t at epoch: {}".
 ↪format(train_accuracy,
                                                                test_accuracy,
                                                                ep))
        metrics.append([train_accuracy, test_accuracy])

    return np.array(metrics), model



def plot_accuracies_v_epoch(metric_array):

    pass
```

```
train data shape: torch.Size([60000, 784])
train label shape: torch.Size([60000])
test data shape: torch.Size([10000, 784])
test label shape: torch.Size([10000])
```

[ ]:
```
metric_array2, trained_model = train(lossflag=2)
```

[ ]:
```
#We plot test accuracy and training accuracy vs. epoch
epoch = np.arange(0,num_epochs)
plt.figure()
LL=plt.scatter(epoch,metric_array2[1:,1],label='testing accuracy')
plt.xlabel('epoch')
plt.ylabel('accuracy')
CE=plt.scatter(epoch,metric_array2[1:,0],label='training accuracy')
plt.title('Test and Training accuracy vs. epoch')
plt.legend(handles=[LL, CE])
```

[ ]:
```
epoch = np.arange(0,num_epochs)
train_err21=metric_array2[1:,0]
test_err21=metric_array2[1:,1]
gap21=train_err21-test_err21

plt.figure()
plt.xlabel('epoch')
plt.ylabel('Generalization Gap')
plt.title('Generalization gap vs. Epoch in Reference Model')
md2=plt.plot(epoch,gap21)
```

[ ]:
```

```

Regularization Part

December 16, 2020

[4]:
```
import torch
import torch.nn as nn  # neural network modules
import torch.nn.functional as F  # activation functions
import torch.optim as optim  # optimizer
from torch.autograd import Variable # add gradients to tensors
from torch.nn import Parameter # model parameter functionality

import torchvision
import torchvision.datasets as datasets

import numpy as np
import matplotlib.pyplot as plt

torch.manual_seed(42)

# ###################################
# import data
# ###################################
# download the MNIST dataset
mnist_trainset = datasets.MNIST(root='./data', train=True, download=True,␣
 ↪transform=None)
mnist_testset = datasets.MNIST(root='./data', train=False, download=True,␣
 ↪transform=None)


# separate into data and labels
# training data
train_data = mnist_trainset.data.to(dtype=torch.float32)
train_data = train_data.reshape(-1, 784)
train_labels = mnist_trainset.targets.to(dtype=torch.long)

print("train data shape: {}".format(train_data.size()))
print("train label shape: {}".format(train_labels.size()))

# testing data
test_data = mnist_testset.data.to(dtype=torch.float32)
```

```
test_data = test_data.reshape(-1, 784)
test_labels = mnist_testset.targets.to(dtype=torch.long)

print("test data shape: {}".format(test_data.size()))
print("test label shape: {}".format(test_labels.size()))

# load into torch datasets
train_dataset = torch.utils.data.TensorDataset(train_data, train_labels)
test_dataset = torch.utils.data.TensorDataset(test_data, test_labels)

# ###################################
# set hyperparameters
# ###################################
# Parameters
learning_rate = 0.005
num_epochs = 100
batch_size = 128

# Network Parameters
n_hidden_1 = 256  # 1st layer number of neurons
n_hidden_2 = 256  # 2nd layer number of neurons
num_input = 784  # MNIST data input (img shape: 28*28)
num_classes = 10  # MNIST total classes (0-9 digits)

# ###################################
# defining the model
# ###################################

# method 1
class FCNN(nn.Module):
    def __init__(self, input_dim, output_dim,p_drop):
        super(FCNN, self).__init__()

        self.input_dim = input_dim
        self.output_dim = output_dim
        self.layer1 = nn.Linear(input_dim, n_hidden_1)
        self.layer2 = nn.Linear(n_hidden_1, n_hidden_2)
        self.layer3 = nn.Linear(n_hidden_2, num_classes)
        self.dropout = nn.Dropout(p=p_drop)


        self.nonlin1 = nn.Sigmoid()
        self.nonlin2 = nn.Sigmoid()

    def forward(self, x):
        x=self.dropout(x)
        h1 = self.dropout(self.nonlin1(self.layer1(x)))
```

```
        h2 = self.dropout(self.nonlin2(self.layer2(h1)))
        output = self.layer3(h2)

        return output


# ###################################
# helper functions
# ###################################
def get_accuracy(output, targets):
    """calculates accuracy from model output and targets
    """
    output = output.detach()
    predicted = output.argmax(-1)
    correct = (predicted == targets).sum().item()

    accuracy = correct / output.size(0) * 100

    return accuracy


def to_one_hot(y, c_dims=10):
    """converts a N-dimensional input to a NxC dimnensional one-hot encoding
    """
    y_tensor = y.data if isinstance(y, Variable) else y
    y_tensor = y_tensor.type(torch.LongTensor).view(-1, 1)
    c_dims = c_dims if c_dims is not None else int(torch.max(y_tensor)) + 1
    y_one_hot = torch.zeros(y_tensor.size()[0], c_dims).scatter_(1, y_tensor, 1)
    y_one_hot = y_one_hot.view(*y.shape, -1)
    return Variable(y_one_hot) if isinstance(y, Variable) else y_one_hot

# ###################################
# creata dataloader
# ###################################
trainloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size,
                                          shuffle=True)


# ###################################
# main training function
# ###################################

def train(regu,p_drop,regu1):

    model = FCNN(num_input, num_classes,p_drop)
    optimizer = optim.SGD(model.parameters(),␣
 ↪lr=learning_rate,weight_decay=regu)
```

```
    # initialize loss list
    metrics = [[0, 0]]

    # iterate over epochs
    for ep in range(num_epochs):
        model.train()

        # iterate over batches
        for batch_indx, batch in enumerate(trainloader):

            # unpack batch
            data, labels = batch

            # get predictions from model
            pred = model(data)


            def l1_loss(x):
                return torch.abs(x).sum()

            to_regularise = []

            for param in model.parameters():
                to_regularise.append(param.view(-1))

            l1 = regu1*l1_loss(torch.cat(to_regularise))

            loss_fn=nn.CrossEntropyLoss()
            loss = loss_fn(pred, labels)+l1

            # print loss every 100 batches
            #if batch_indx % 100 == 0:
               # print("loss {:.3f} at batch {} epoch {}".format(loss.item(),␣
↪batch_indx, ep))

            # backpropagate the loss
            loss.backward()

            # update parameters
            optimizer.step()

            # reset gradients
            optimizer.zero_grad()

        # compute full train and test accuracies
        # every epoch
```

```
        model.eval() # model will not calculate gradients for this pass
        train_ep_pred = model(train_data)
        test_ep_pred = model(test_data)

        train_accuracy = get_accuracy(train_ep_pred, train_labels)
        test_accuracy = get_accuracy(test_ep_pred, test_labels)

        print("train acc: {}\t test acc: {}\t at epoch: {}".
↪format(train_accuracy,
                                                                test_accuracy,
                                                                ep))
        metrics.append([train_accuracy, test_accuracy])

    return np.array(metrics), model


def plot_accuracies_v_epoch(metric_array):
    pass
```

```
train data shape: torch.Size([60000, 784])
train label shape: torch.Size([60000])
test data shape: torch.Size([10000, 784])
test label shape: torch.Size([10000])
```

[ ]:
```
####L1-regularized model##########
metric_array11, trained_model1 = train(regu=0,regu1=5,p_drop=0)
metric_array21, trained_model1 = train(regu=0,regu1=1,p_drop=0)
metric_array31, trained_model1 = train(regu=0,regu1=0.01,p_drop=0)
metric_array41, trained_model1 = train(regu=0,regu1=0.001,p_drop=0)

epoch = np.arange(0,num_epochs)
train_err11=metric_array11[1:,0]
test_err11=metric_array11[1:,1]
gap11=train_err11-test_err11

train_err21=metric_array21[1:,0]
test_err21=metric_array21[1:,1]
gap21=train_err21-test_err21

train_err31=metric_array31[1:,0]
test_err31=metric_array31[1:,1]
gap31=train_err31-test_err31

train_err41=metric_array41[1:,0]
test_err41=metric_array41[1:,1]
gap41=train_err41-test_err41
```

```
plt.figure()
plt.xlabel('epoch')
plt.ylabel('training accuracy')
plt.title('Training error vs. Epoch in L1')
md1=plt.plot(epoch,train_err11,label='lambda=5')
md2=plt.plot(epoch,train_err21,label='lambda=1')
md3=plt.plot(epoch,train_err31,label='lambda=0.01')
md4=plt.plot(epoch,train_err41,label='lambda=0.001')
plt.legend(('lambda.L1=5', 'lambda.L1=1', 'lambda.L1=0.01','lambda.L1=0.001'),
           )


plt.figure()
plt.xlabel('epoch')
plt.ylabel('testing accuracy')
plt.title('testing error vs. Epoch in L1')
md1=plt.plot(epoch,test_err11,label='lambda=5')
md2=plt.plot(epoch,test_err21,label='lambda=1')
md3=plt.plot(epoch,test_err31,label='lambda=0.01')
md4=plt.plot(epoch,test_err41,label='lambda=0.001')
plt.legend(('lambda.L1=5', 'lambda.L1=1', 'lambda.L1=0.01','lambda.L1=0.001'),
           )


plt.figure()
plt.xlabel('epoch')
plt.ylabel('Generalization Gap')
plt.title('Generalization gap vs. Epoch in L1')
md1=plt.plot(epoch,gap11,label='lambda=5')
md2=plt.plot(epoch,gap21,label='lambda=1')
md3=plt.plot(epoch,gap31,label='lambda=0.01')
md4=plt.plot(epoch,gap41,label='lambda=0.001')
plt.legend(('lambda.L1=5', 'lambda.L1=1', 'lambda.L1=0.01','lambda.L1=0.001'),
           )
```

[ ]:
```
####L2-regularized model########
metric_array1, trained_model1 = train(regu=5,regu1=0,p_drop=0)
metric_array2, trained_model1 = train(regu=1,regu1=0,p_drop=0)
metric_array3, trained_model1 = train(regu=0.01,regu1=0,p_drop=0)
metric_array4, trained_model1 = train(regu=0.001,regu1=0,p_drop=0)

epoch = np.arange(0,num_epochs)
train_err1=metric_array1[1:,0]
test_err1=metric_array1[1:,1]
gap1=train_err1-test_err1

train_err2=metric_array2[1:,0]
```

```
test_err2=metric_array2[1:,1]
gap2=train_err2-test_err2

train_err3=metric_array3[1:,0]
test_err3=metric_array3[1:,1]
gap3=train_err3-test_err3

train_err4=metric_array4[1:,0]
test_err4=metric_array4[1:,1]
gap4=train_err4-test_err4
plt.figure()
plt.xlabel('epoch')
plt.ylabel('training accuracy')
plt.title('Training error vs. Epoch in L2')
md1=plt.plot(epoch,train_err1,label='lambda=5')
md2=plt.plot(epoch,train_err2,label='lambda=1')
md3=plt.plot(epoch,train_err3,label='lambda=0.01')
md4=plt.plot(epoch,train_err4,label='lambda=0.001')
plt.legend(('lambda.L2=5', 'lambda.L2=1', 'lambda.L2=0.01','lambda.L2=0.001'),
           )



plt.figure()
plt.xlabel('epoch')
plt.ylabel('testing accuracy')
plt.title('testing error vs. Epoch in L2')
md1=plt.plot(epoch,test_err1,label='lambda=5')
md2=plt.plot(epoch,test_err2,label='lambda=1')
md3=plt.plot(epoch,test_err3,label='lambda=0.01')
md4=plt.plot(epoch,test_err4,label='lambda=0.001')
plt.legend(('lambda.L2=5', 'lambda.L2=1', 'lambda.L2=0.01','lambda.L2=0.001'),
           )



plt.figure()
plt.xlabel('epoch')
plt.ylabel('Generalization Gap')
plt.title('Generalization gap vs. Epoch in L2')
md1=plt.plot(epoch,gap1,label='lambda=5')
md2=plt.plot(epoch,gap2,label='lambda=1')
md3=plt.plot(epoch,gap3,label='lambda=0.01')
md4=plt.plot(epoch,gap4,label='lambda=0.001')
plt.legend(('lambda.L2=5', 'lambda.L2=1', 'lambda.L2=0.01','lambda.L2=0.001'),
           )
```

[ ]:
```
########Dropout model############
metric_array13, trained_model1 = train(regu=0,regu1=0,p_drop=0.1)
metric_array23, trained_model1 = train(regu=0,regu1=0,p_drop=0.2)
metric_array33, trained_model1 = train(regu=0,regu1=0,p_drop=0.5)
metric_array43, trained_model1 = train(regu=0,regu1=0,p_drop=0.7)

epoch = np.arange(0,num_epochs)
train_err13=metric_array13[1:,0]
test_err13=metric_array13[1:,1]
gap13=train_err13-test_err13

train_err23=metric_array23[1:,0]
test_err23=metric_array23[1:,1]
gap23=train_err23-test_err23

train_err33=metric_array33[1:,0]
test_err33=metric_array33[1:,1]
gap33=train_err33-test_err33

train_err43=metric_array43[1:,0]
test_err43=metric_array43[1:,1]
gap43=train_err43-test_err43
plt.figure()
plt.xlabel('epoch')
plt.ylabel('training accuracy')
plt.title('Training error vs. Epoch in Dropout')
md1=plt.plot(epoch,train_err13,label='lambda=5')
md2=plt.plot(epoch,train_err23,label='lambda=1')
md3=plt.plot(epoch,train_err33,label='lambda=0.01')
md4=plt.plot(epoch,train_err43,label='lambda=0.001')
plt.legend(('p=0.1', 'p=0.2', 'p=0.5','p=0.7'),
           )


plt.figure()
plt.xlabel('epoch')
plt.ylabel('testing accuracy')
plt.title('testing error vs. Epoch in Dropout')
md1=plt.plot(epoch,test_err13,label='lambda=5')
md2=plt.plot(epoch,test_err23,label='lambda=1')
md3=plt.plot(epoch,test_err33,label='lambda=0.01')
md4=plt.plot(epoch,test_err43,label='lambda=0.001')
plt.legend(('p=0.1', 'p=0.2', 'p=0.5','p=0.7'),
           )


plt.figure()
```

```
plt.xlabel('epoch')
plt.ylabel('Generalization Gap')
plt.title('Generalization gap vs. Epoch in Dropout')
md1=plt.plot(epoch,gap13,label='lambda=5')
md2=plt.plot(epoch,gap23,label='lambda=1')
md3=plt.plot(epoch,gap33,label='lambda=0.01')
md4=plt.plot(epoch,gap43,label='lambda=0.001')
plt.legend(('p=0.1', 'p=0.2', 'p=0.5','p=0.7'),
           )
```