ASSIGNMENT- 01

EXCERCISE 1

In [1]:

```
import torch
import torchvision  #To be able to access standard datasets more easily
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt
import numpy as np  # To plot and display stuff
import torch.optim as optim # Where the optimization modules are
import urllib.request
from random import randint
from mlxtend.data import loadlocal_mnist
import platform
from sklearn.metrics import accuracy_score
```

In [2]:

```
# Using torchvision we can conveniently load some datasets
trainset = torchvision.datasets.MNIST(root='./data', train=True, download=True, transform
=ToTensor())
testset = torchvision.datasets.MNIST(root='./data', train=False, download=True, transform
=ToTensor())
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to ./data/MNIST/r
aw/train-images-idx3-ubyte.gz


Extracting ./data/MNIST/raw/train-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to ./data/MNIST/r
aw/train-labels-idx1-ubyte.gz


Extracting ./data/MNIST/raw/train-labels-idx1-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to ./data/MNIST/ra
w/t10k-images-idx3-ubyte.gz


Extracting ./data/MNIST/raw/t10k-images-idx3-ubyte.gz to ./data/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to ./data/MNIST/ra
w/t10k-labels-idx1-ubyte.gz


Extracting ./data/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./data/MNIST/raw
```

In [3]:

```
# Extract tensor of data and labels for both the training and the test set
x, y = trainset.data.float(), trainset.targets
x_test, y_test = testset.data.float(), testset.targets
```

Q1

- **Try to load the same data directly from the "MINST database" website http://yann.lecun.com/exdb/mnist/**
- **Be careful that the images can have a different normalization and encoding**

In [4]:

```
!pip install wget
```

```
Looking in indexes: https://pypi.org/simple, https://us-python.pkg.dev/colab-wheels/publi
c/simple/
Collecting wget
  Downloading wget-3.2.zip (10 kB)
Building wheels for collected packages: wget
  Building wheel for wget (setup.py) ... done
  Created wheel for wget: filename=wget-3.2-py3-none-any.whl size=9675 sha256=381b9eb9102
5630cfd01c3afd1b6b73e81480b08f2299d98a4ecc55ab9cdca9d
  Stored in directory: /root/.cache/pip/wheels/a1/b6/7c/0e63e34eb06634181c63adacca38b79ff
8f35c37e3c13e3c02
Successfully built wget
Installing collected packages: wget
Successfully installed wget-3.2
```

In [5]:

```
import gzip
import shutil
import wget


trainset_images = wget.download("http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte
.gz", "train-images-idx3-ubyte.gz")

trainset_labels = wget.download("http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte
.gz", "train-labels-idx1-ubyte.gz")

testset_images = wget.download("http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.g
z", "t10k-images-idx3-ubyte.gz")

testset_labels = wget.download("http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.g
z", "t10k-labels-idx1-ubyte.gz")

filenames = ["train-images-idx3-ubyte", "train-labels-idx1-ubyte", "t10k-images-idx3-uby
te", "t10k-labels-idx1-ubyte"]

for f in filenames:
    with gzip.open(f+'.gz', 'rb') as f_in:
        with open(f, 'wb') as f_out:
            shutil.copyfileobj(f_in, f_out)

x_train_download, y_train_download = loadlocal_mnist(
            images_path='train-images-idx3-ubyte',
            labels_path='train-labels-idx1-ubyte')

x_test_download, y_test_download = loadlocal_mnist(
            images_path='t10k-images-idx3-ubyte',
            labels_path='t10k-labels-idx1-ubyte')
```

In [6]:

```
print(x_train_download.shape, y_train_download.shape)
print(x.shape, y.shape)
print(x_test_download.shape, y_test_download.shape)
print(x_test.shape, y_test.shape)
```

```
(60000, 784) (60000,)
torch.Size([60000, 28, 28]) torch.Size([60000])
(10000, 784) (10000,)
torch.Size([10000, 28, 28]) torch.Size([10000])
```

The difference between the downloaded data and the one loaded from torch vision is that the last two dimensions are flattened in the downloaded data, which is acually the correct shape of the data that we need to feed to the model.

Code to flatten the last two dimensions:

```
[Python]
torch.flatten(x, start_dim=1, end_dim=2)
```

In [7]:

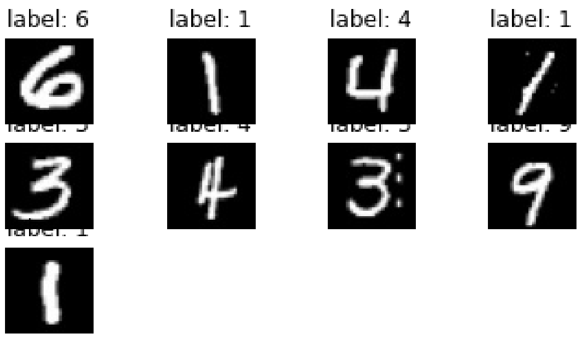
```
# Transform labels to one_hot encoding
y_one_hot = torch.nn.functional.one_hot(y.to(torch.int64), num_classes=10).float()
y_test_one_hot = torch.nn.functional.one_hot(y_test.to(torch.int64), num_classes=10).flo
at()
```

Q2

Using the utilities in plt and numpy display some images and check that the corresponding labels are consistent

In [8]:

```
max_image_index = trainset.data.shape[0] - 1
n_images_to_show = 9
for i in range(0, n_images_to_show):
    image_idx = randint(0, max_image_index) # pick a random image from our dataset
    image, label = trainset[image_idx]
    plt.subplot(int(n_images_to_show/2),int(n_images_to_show/2), i+1)
    plt.imshow(image.numpy()[0], cmap='gray')
    plt.axis("off")
    plt.title("label: " + str(label))
plt.show()
```



Q3

- **Complete the code below so to have a MLP with one hidden layer with 300 neurons**
- **Remember that we want one-hot outputs**

In [9]:

```
# Now let us define the neural network we are using
image_size = trainset.data.shape[1] * trainset.data.shape[2]
net = torch.nn.Sequential(
    torch.nn.Linear(image_size, 300),
    torch.nn.Sigmoid(),
    torch.nn.Linear(300, 10),
)
```

In [10]:

```
# Now we define the optimizer and the loss function
loss = torch.nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.1)
```

In [11]:

```
# GPU
dev = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
# For Accuracy
def accuracy(out, yb):
    preds = torch.argmax(out, dim=1)
    return (preds == yb).float().mean() * 100
```

Q4

- **Complete the code below to perform a GD based optimization**

In [12]:

```
# CPU TO GPU

x = x.to(dev)
y = y.to(dev)
x_test = x_test.to(dev)
y_test = y_test.to(dev)
net = net.to(dev)

for k in range(100):
    optimizer.zero_grad()

    inputs = torch.flatten(x, start_dim=1, end_dim=2).to(dev)
    outputs = net(inputs)
    labels = y_one_hot.to(dev)

    #Define the empirical risk
    Risk = loss(outputs, labels)

    #Make the backward step (1 line instruction)
    Risk.backward()

    #Upadte the parameters (1 line instruction)
    optimizer.step()

    with torch.no_grad():
        print("k=", k, "    Risk = ", Risk.item())
```

```
k= 0     Risk =  2.426805019378662
k= 1     Risk =  2.059744119644165
k= 2     Risk =  1.8141142129898071
k= 3     Risk =  1.6330887079238892
k= 4     Risk =  1.4865326881408691
k= 5     Risk =  1.3666075468063354
k= 6     Risk =  1.2655328512191772
k= 7     Risk =  1.179999589920044
k= 8     Risk =  1.1057738065719604
k= 9     Risk =  1.040987253189087
k= 10     Risk =  0.984626054763794
k= 11     Risk =  0.9357080459594727
k= 12     Risk =  0.8923757076263428
k= 13     Risk =  0.8543803095817566
k= 14     Risk =  0.8201102018356323
k= 15     Risk =  0.7890397310256958
k= 16     Risk =  0.7604593634605408
k= 17     Risk =  0.7347379326820374
k= 18     Risk =  0.7110958099365234
k= 19     Risk =  0.6894578337669373
k= 20     Risk =  0.6695007681846619
k= 21     Risk =  0.650956392288208
k= 22     Risk =  0.6339602470397949
k= 23     Risk =  0.6178784966468811
k= 24     Risk =  0.6031579375267029
k= 25     Risk =  0.589174211025238
k= 26     Risk =  0.5763946175575256
k= 27     Risk =  0.5640968680381775
k= 28     Risk =  0.5526596307754517
k= 29     Risk =  0.5418834686279297
k= 30     Risk =  0.5318153500556946
k= 31     Risk =  0.5222426652908325
```

```
k= 32    Risk =  0.5131532549858093
k= 33    Risk =  0.5047115087509155
k= 34    Risk =  0.49653372168540955
k= 35    Risk =  0.488938570022583
k= 36    Risk =  0.4814798831939697
k= 37    Risk =  0.47461816668510437
k= 38    Risk =  0.4677993953227997
k= 39    Risk =  0.4615384638309479
k= 40    Risk =  0.4552369713783264
k= 41    Risk =  0.4495449364185333
k= 42    Risk =  0.4438147246837616
k= 43    Risk =  0.4386565089225769
k= 44    Risk =  0.4333876967430115
k= 45    Risk =  0.42859765887260437
k= 46    Risk =  0.4237360954284668
k= 47    Risk =  0.41927942633628845
k= 48    Risk =  0.41471827030181885
k= 49    Risk =  0.4104352593421936
k= 50    Risk =  0.40599918365478516
k= 51    Risk =  0.4018440544605255
k= 52    Risk =  0.39748871326446533
k= 53    Risk =  0.39345207810401917
k= 54    Risk =  0.389379620552063
k= 55    Risk =  0.3856368362903595
k= 56    Risk =  0.3820176124572754
k= 57    Risk =  0.37871691584587097
k= 58    Risk =  0.3756451904773712
k= 59    Risk =  0.3729083836078644
k= 60    Risk =  0.3703365921974182
k= 61    Risk =  0.36827659606933594
k= 62    Risk =  0.36628881096839905
k= 63    Risk =  0.3648653030395508
k= 64    Risk =  0.3632921278476715
k= 65    Risk =  0.36169764399528503
k= 66    Risk =  0.35941368341445923
k= 67    Risk =  0.3568192720413208
k= 68    Risk =  0.353937029838562
k= 69    Risk =  0.3504285514354706
k= 70    Risk =  0.34664905071258545
k= 71    Risk =  0.3429558575153351
k= 72    Risk =  0.3392547070980072
k= 73    Risk =  0.33610233664512634
k= 74    Risk =  0.333213746547699
k= 75    Risk =  0.330920934677124
k= 76    Risk =  0.32883813977241516
k= 77    Risk =  0.3272038996219635
k= 78    Risk =  0.32573580741882324
k= 79    Risk =  0.32494673132896423
k= 80    Risk =  0.32425329089164734
k= 81    Risk =  0.3242957890033722
k= 82    Risk =  0.32376882433891296
k= 83    Risk =  0.3232015073299408
k= 84    Risk =  0.3217678964138031
k= 85    Risk =  0.3199893534183502
k= 86    Risk =  0.31765809655189514
k= 87    Risk =  0.3152184784412384
k= 88    Risk =  0.3124867081642151
k= 89    Risk =  0.3095186948776245
k= 90    Risk =  0.3068850338459015
k= 91    Risk =  0.30432865023612976
k= 92    Risk =  0.30246058106422424
k= 93    Risk =  0.3004642128944397
k= 94    Risk =  0.2989695370197296
k= 95    Risk =  0.2973100244998932
k= 96    Risk =  0.2958942651748657
k= 97    Risk =  0.29477912187576294
k= 98    Risk =  0.2938283085823059
k= 99    Risk =  0.29318544268608093
```

Q5

Compute the final accuracy on test set

In [13]:

```
predict_test = net(torch.flatten(x_test, start_dim=1, end_dim=2))
acc = accuracy(predict_test, y_test).cpu()
print(f"Final Accuracy on test: {acc} %")
```

```
Final Accuracy on test: 91.97000122070312 %
```

Exercise 2

Q1.

On line 49 of the code it is commented that we want to perform a GD based optimization. However on line 45 we invoked optim.SGD as the optimizer. Explain why in this case we are still performing a gradient descent on the whole dataset even if it seems that we are invoking a stochastic method.

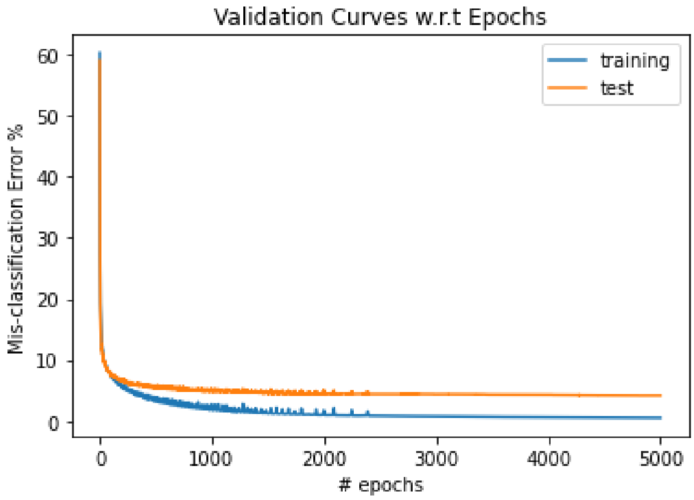
Yes, we are indeed performing Gradient Descent (GD) because in Q4, we invoke the SGD optimizer but we pass the entire training dataset to our optimizer (no batches are involved here).

Q2.

Discuss over-fitting issues by monitoring the train and test error curves.

From the Validation curve shown below it can be inferred that both training and testing errors are very high for low value of Epochs and rapidly increase with further Epochs. That is model went from under-fitting to over-fitting.

Both test and train errors keeps on decreasing even for very high values of Epochs and test error converges quicker than train error. This shows that our model exhibits little-overfitting but not very severe over-fitting.



Q3.

Discuss what role does the choice of the network (i.e. number of layers and number of neurons per layer) have on the bias-variance trade-off. First describe your expectations based on theoretical analysis (arguing on the different capacity of the models) then test this expectations with a small experimental campaign. Is the expected behavior confirmed by experimental results? Briefly discuss your findings.

Theoretical Analysis

The bias and variance tradeoff:

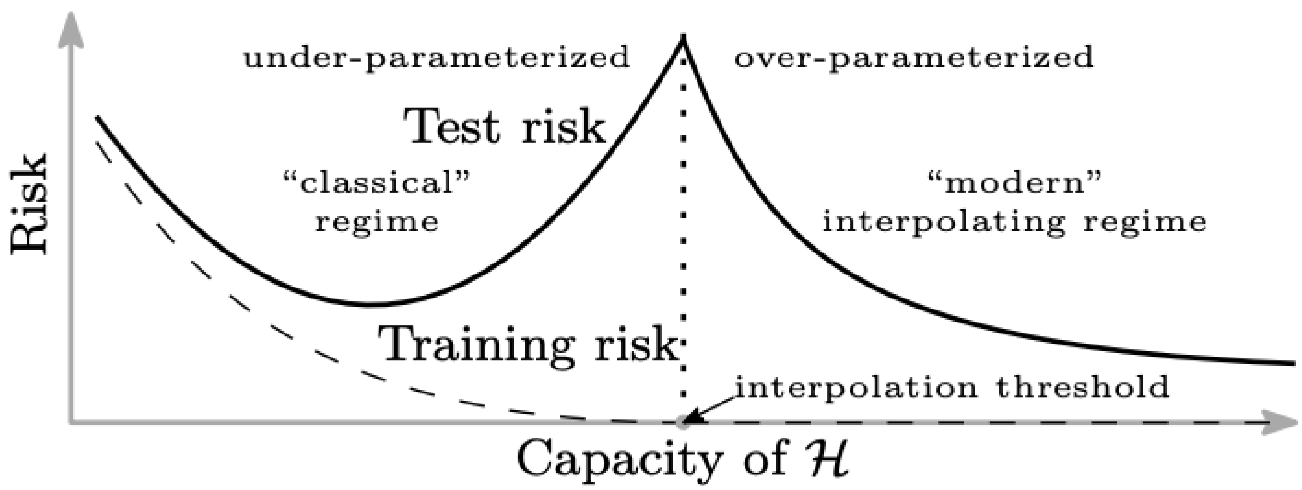
- under-fitting => high training and testing risk
- over-fitting => test risk is very high while the training risk is very low.

Aim: Find the sweet spot between under-fitting and over-fitting by varying the capacity (complexity) of the model.

As already seen in the lecture, in theory:

- **Classical regime:** Very high bias and small variance (under-fitting) for a low capacity model and a small bias and large variance (over-fitting) when the model capacity is very high. But it is possible to find the sweet spot by finding an optimal capacity.
- **Modern interpolating regime:** the model behaves similarly as in the classical regime, but after a certain high capacity (deep networks) it seems that it is possible to have a model with perfect fitting on the training data and test data at the same time, thereby achieving better model generalization.

Refer:

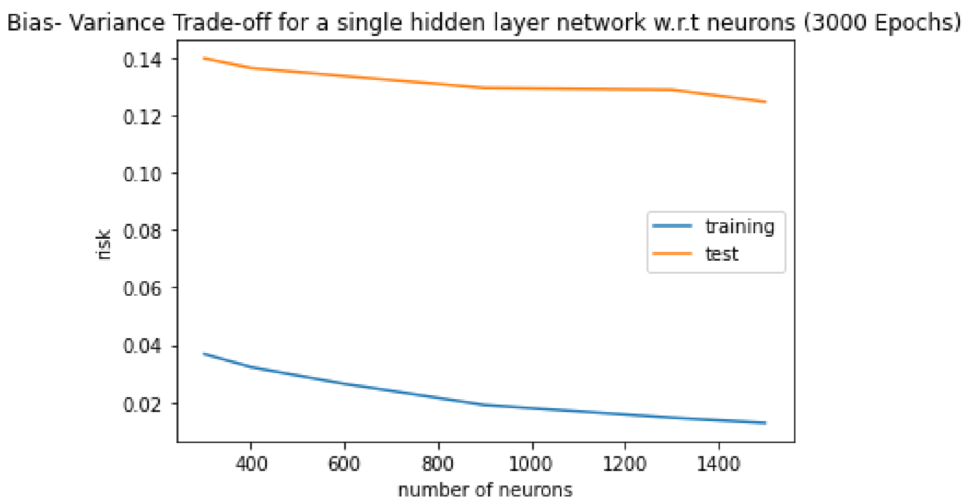


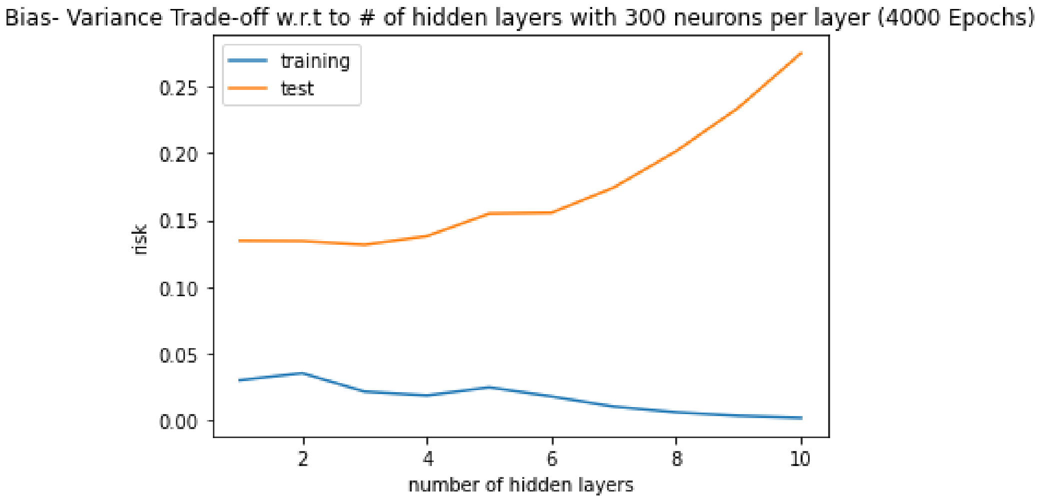
source https://arxiv.org/pdf/1812.11118.pdf

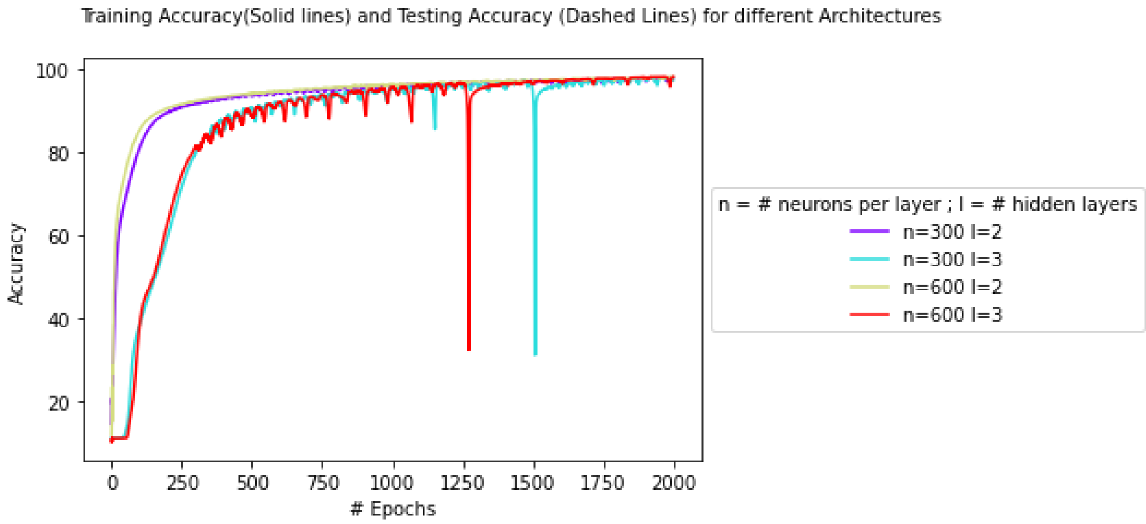
Experimental Analysis:

The following Bias-Variance Trade-off are obtained after the analysis where model capacity is increased by:

- the no. of neurons (for a single hidden layer): This is found to be part of modern interpolating regime since both risks continue to decrease.
- Fixing the no. of neurons per layer to 300 and increasing the no. of layers: This is found to be part of classical regime, since there is over-fitting.
- Based upon the the two previous graphs, the final analysis is done and it found to be that a Neural Network with Architecture of 600 neurons per hidden layer, with 2 hidden layers performs better among the other compared models (Even the 300 neurons perform good but higher neurons are usually preferred from the first graph). (Note: This combination of Layers and Neurons is not unique, but this is what I chose). This Architecture is also belongs to Modern interpolating regime.







Q4.

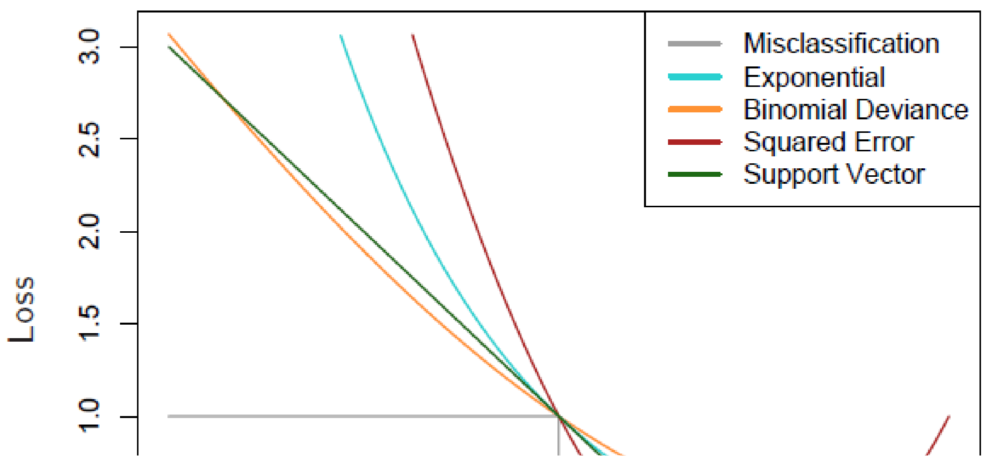
Discuss the benefits of using a cross entropy loss with respect to a quadratic loss.

This is can be explained by an anology to Binary classsification problem. The below figure taken from the book, "Elements of Statistical Learning, Second Edition" shows various Loss functions in case of Binary classification problems. Here, yf is positive in case of a correct classification and negative in case of wrong classification.

As you can see that Cross-Entropy(Binomial Deviance) is a monotonously decreasing function, and it penalizes misclassified data more heavily than correctly classified data.

Quadratic loss (Squared Error) is not monotonically decreasing and starts to increase after $yf > 1$, thereby penalizing the correctly classified points. This turns into a non-convex optimization problem, which is undesirable.

Hence it is beneficial to use Cross-Entropy.



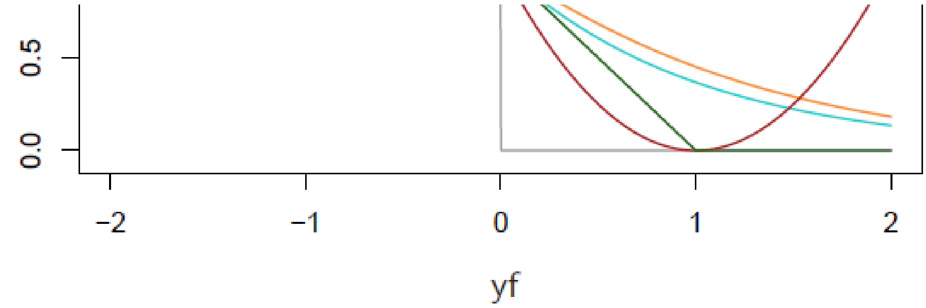


FIGURE 10.4. *Loss functions for two-class classification. The response is* $y = \pm 1$*; the prediction is* f*, with class prediction* $\text{sign}(f)$*. The losses are misclassification:* $I(\text{sign}(f) \neq y)$*; exponential:* $\exp(-yf)$*; binomial deviance:* $\log(1 + \exp(-2yf))$*; squared error:* $(y - f)^2$*; and support vector:* $(1 - yf)_+$ *(see Section 12.3). Each function has been scaled so that it passes through the point* $(0, 1)$.

Q5.

Why using a one-hot encoding? Wouldn't be simpler to use a single output? Hint: The answer has to do with the interplay between the loss and the sigmoidal activation functions.

In classification problems, involving categorical variables. It is beneficial to use One- hot Encoding if there is no natural relationship between the categories.

If these categorical variables are instead treated with ordinal Encoding, then we are giving a wrong sense of notion to our model. This means if we define a Quadratic loss function (again this results in non-convex optimization) then the loss of misclassification of one class w.r.t another is not same anymore. That means we are inducing inherent biases into our data analysis.

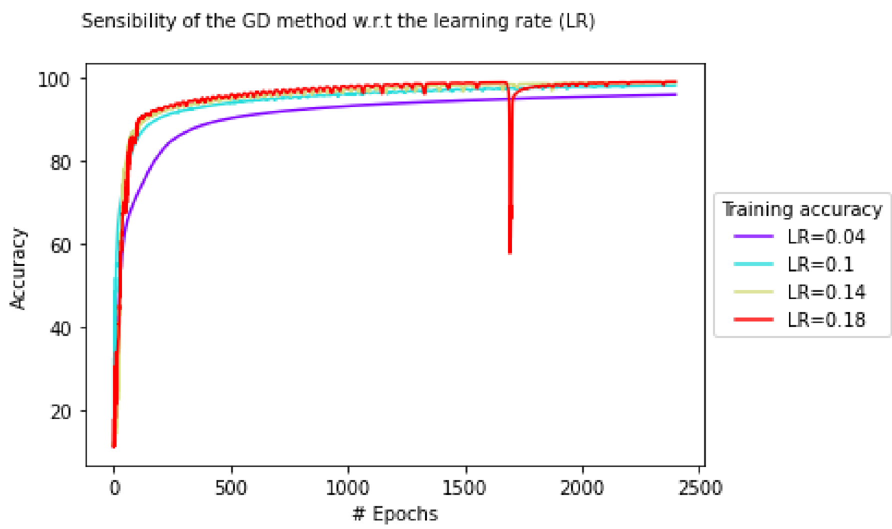
With Ordinal Encoding, we cannot use Sigmoid or Softmax activation function which implies that we cannot evaluate Cross-Entropy loss (a more robust loss function)

To avoid this, we can use one-hot encoding, after which we can define Cross-Entropy loss (which has symmetrical misclassification loss about any classes).

Q6.

Test the sensibility of the gradient descent method with respect to the learning rate.

The learning rates 0.1 and 0.14 are found to be optimal with faster convergence as well as smooth curves with less variability. Learning rate 0.14 behaves slightly better than 0.1. Other combinations like 0.2 has faster convergence but erratic curve and 0.04 is converging too slowly.



Q7.

With the network architecture that is described in Q3. of Exercise 1 (line 32–33 of the code) do you experience any problem related to the vanishing of the gradient? Why?

In [14]:

```python
# CPU TO GPU

x = x.to(dev)
y = y.to(dev)
x_test = x_test.to(dev)
y_test = y_test.to(dev)
net = net.to(dev)

for k in range(200):
    optimizer.zero_grad()

    inputs = torch.flatten(x, start_dim=1, end_dim=2).to(dev)
    outputs = net(inputs)
    labels = y_one_hot.to(dev)

    #Define the empirical risk
    Risk = loss(outputs, labels)

    #Make the backward step (1 line instruction)
    Risk.backward()

    #Upadte the parameters (1 line instruction)
    optimizer.step()

    with torch.no_grad():
        print("k=", k, "   Risk = ", Risk.item())
```

```
k= 0    Risk =  0.2926501929759979
k= 1    Risk =  0.2927183508872986
k= 2    Risk =  0.29261791706085205
k= 3    Risk =  0.29291239380836487
k= 4    Risk =  0.29318758845329285
k= 5    Risk =  0.2921077013015747
k= 6    Risk =  0.29083219170570374
k= 7    Risk =  0.28884947299957275
k= 8    Risk =  0.28688308596611023
k= 9    Risk =  0.2842438519001007
k= 10    Risk =  0.28164106607437134
k= 11    Risk =  0.27926185727119446
k= 12    Risk =  0.27746737003326416
k= 13    Risk =  0.2760606110095978
k= 14    Risk =  0.2752714157104492
k= 15    Risk =  0.27495288848876953
k= 16    Risk =  0.2743118405342102
k= 17    Risk =  0.2744567096233368
k= 18    Risk =  0.2742465138435364
k= 19    Risk =  0.2744676470756531
k= 20    Risk =  0.2738681733608246
k= 21    Risk =  0.27383166551589966
k= 22    Risk =  0.27232125401496887
k= 23    Risk =  0.2711130976676941
k= 24    Risk =  0.26918521523475647
k= 25    Risk =  0.2672545313835144
k= 26    Risk =  0.26431146264076233
k= 27    Risk =  0.26180848479270935
k= 28    Risk =  0.25969094038009644
k= 29    Risk =  0.2585197687149048
k= 30    Risk =  0.25735798478126526
k= 31    Risk =  0.25665342807769775
k= 32    Risk =  0.2557223439216614
k= 33    Risk =  0.25571462512016296
k= 34    Risk =  0.2555045783519745
```

```
k= 35     Risk =  0.25625938177108765
k= 36     Risk =  0.2563525140285492
k= 37     Risk =  0.2574616074562073
k= 38     Risk =  0.25728359818458557
k= 39     Risk =  0.2581658959388733
k= 40     Risk =  0.2567943334579468
k= 41     Risk =  0.2563040554523468
k= 42     Risk =  0.2540383040904999
k= 43     Risk =  0.25263962149620056
k= 44     Risk =  0.2501818537712097
k= 45     Risk =  0.24813112616539001
k= 46     Risk =  0.24633224308490753
k= 47     Risk =  0.24470557272434235
k= 48     Risk =  0.24363331496715546
k= 49     Risk =  0.24262571334838867
k= 50     Risk =  0.2422822117805481
k= 51     Risk =  0.2420254945755005
k= 52     Risk =  0.2424962818622589
k= 53     Risk =  0.24313460290431976
k= 54     Risk =  0.2443612962961197
k= 55     Risk =  0.2452876716852188
k= 56     Risk =  0.24629680812358856
k= 57     Risk =  0.24638640880584717
k= 58     Risk =  0.24603036046028137
k= 59     Risk =  0.24444851279258728
k= 60     Risk =  0.24294348061084747
k= 61     Risk =  0.2407308667898178
k= 62     Risk =  0.2389145791530609
k= 63     Risk =  0.23619934916496277
k= 64     Risk =  0.2344420701265335
k= 65     Risk =  0.2328115850687027
k= 66     Risk =  0.23181286454200745
k= 67     Risk =  0.23054656386375427
k= 68     Risk =  0.22976073622703552
k= 69     Risk =  0.22895175218582153
k= 70     Risk =  0.2285642921924591
k= 71     Risk =  0.22760607302188873
k= 72     Risk =  0.22726745903491974
k= 73     Risk =  0.22669316828250885
k= 74     Risk =  0.22687220573425293
k= 75     Risk =  0.22642987966537476
k= 76     Risk =  0.22682012617588043
k= 77     Risk =  0.2267308086156845
k= 78     Risk =  0.22772616147994995
k= 79     Risk =  0.22814303636550903
k= 80     Risk =  0.22991083562374115
k= 81     Risk =  0.22986464202404022
k= 82     Risk =  0.23083741962909698
k= 83     Risk =  0.230153888463974
k= 84     Risk =  0.23020540177822113
k= 85     Risk =  0.22781890630722046
k= 86     Risk =  0.2261677086353302
k= 87     Risk =  0.22314539551734924
k= 88     Risk =  0.22016242146492004
k= 89     Risk =  0.2175503522157669
k= 90     Risk =  0.21550653874874115
k= 91     Risk =  0.2141910195350647
k= 92     Risk =  0.21327273547649384
k= 93     Risk =  0.21279235184192657
k= 94     Risk =  0.2125612199306488
k= 95     Risk =  0.2126246690750122
k= 96     Risk =  0.21312540769577026
k= 97     Risk =  0.2145322561264038
k= 98     Risk =  0.21559983491897583
k= 99     Risk =  0.21861563622951508
k= 100     Risk =  0.2197045087814331
k= 101     Risk =  0.2223576307296753
k= 102     Risk =  0.22173382341861725
k= 103     Risk =  0.22191974520683289
k= 104     Risk =  0.21947720646858215
k= 105     Risk =  0.21879537403583527
k= 106     Risk =  0.21628284454345703
```

```
k= 107    Risk =  0.21413388848304749
k= 108    Risk =  0.21093377470970154
k= 109    Risk =  0.20894813537597656
k= 110    Risk =  0.20756617188453674
k= 111    Risk =  0.20660939812660217
k= 112    Risk =  0.20529013872146606
k= 113    Risk =  0.20474238693714142
k= 114    Risk =  0.2045293152332306
k= 115    Risk =  0.20505426824092865
k= 116    Risk =  0.20497190952301025
k= 117    Risk =  0.20607945322990417
k= 118    Risk =  0.20693553984165192
k= 119    Risk =  0.2097356766462326
k= 120    Risk =  0.21056292951107025
k= 121    Risk =  0.21298687160015106
k= 122    Risk =  0.21197445690631866
k= 123    Risk =  0.21341504156589508
k= 124    Risk =  0.21101883053779602
k= 125    Risk =  0.2100611925125122
k= 126    Risk =  0.20667849481105804
k= 127    Risk =  0.20461009442806244
k= 128    Risk =  0.20136351883411407
k= 129    Risk =  0.1992969661951065
k= 130    Risk =  0.19717060029506683
k= 131    Risk =  0.1957741528749466
k= 132    Risk =  0.1951116919517517
k= 133    Risk =  0.1946978121995926
k= 134    Risk =  0.19440490007400513
k= 135    Risk =  0.19440752267837524
k= 136    Risk =  0.19476091861724854
k= 137    Risk =  0.19568632543087006
k= 138    Risk =  0.1965150684118271
k= 139    Risk =  0.19827355444431305
k= 140    Risk =  0.19923341274261475
k= 141    Risk =  0.20116674900054932
k= 142    Risk =  0.2009112536907196
k= 143    Risk =  0.20290818810462952
k= 144    Risk =  0.2012593001127243
k= 145    Risk =  0.20198950171470642
k= 146    Risk =  0.19940346479415894
k= 147    Risk =  0.19900792837142944
k= 148    Risk =  0.1957746148109436
k= 149    Risk =  0.1936778575181961
k= 150    Risk =  0.19116249680519104
k= 151    Risk =  0.18990430235862732
k= 152    Risk =  0.18894174695014954
k= 153    Risk =  0.18827059864997864
k= 154    Risk =  0.1878347545862198
k= 155    Risk =  0.1877845972776413
k= 156    Risk =  0.1882404386997223
k= 157    Risk =  0.18901365995407104
k= 158    Risk =  0.19073697924613953
k= 159    Risk =  0.19297581911087036
k= 160    Risk =  0.19540032744407654
k= 161    Risk =  0.1971975564956665
k= 162    Risk =  0.19725894927978516
k= 163    Risk =  0.19753897190093994
k= 164    Risk =  0.19630125164985657
k= 165    Risk =  0.19446136057376862
k= 166    Risk =  0.19239479303359985
k= 167    Risk =  0.19055277109146118
k= 168    Risk =  0.18853633105754852
k= 169    Risk =  0.18706756830215454
k= 170    Risk =  0.185482457280159
k= 171    Risk =  0.18436165153980255
k= 172    Risk =  0.1833115816116333
k= 173    Risk =  0.18281735479831696
k= 174    Risk =  0.1824057400226593
k= 175    Risk =  0.1819288432598114
k= 176    Risk =  0.18170152604579926
k= 177    Risk =  0.18148984014987946
k= 178    Risk =  0.18171274662017822
```

```
k= 179     Risk =  0.18130789697170258
k= 180     Risk =  0.1812773048877716
k= 181     Risk =  0.18132972717285156
k= 182     Risk =  0.18196803331375122
k= 183     Risk =  0.1819193959236145
k= 184     Risk =  0.18212340772151947
k= 185     Risk =  0.18292094767093658
k= 186     Risk =  0.18459704518318176
k= 187     Risk =  0.18615353107452393
k= 188     Risk =  0.18783020973205566
k= 189     Risk =  0.18829751014709473
k= 190     Risk =  0.18858569860458374
k= 191     Risk =  0.1872432976961136
k= 192     Risk =  0.18692493438720703
k= 193     Risk =  0.18431705236434937
k= 194     Risk =  0.1818595975637436
k= 195     Risk =  0.17833536863327026
k= 196     Risk =  0.17617174983024597
k= 197     Risk =  0.17439508438110352
k= 198     Risk =  0.17350739240646362
k= 199     Risk =  0.17306512594223022
```

No, as you can see from the two hundered epoch of the validation curves from Ex 2: $Q3$ we did not experience any Vanishing gradient because the training risk keeps on decreasing. This is because there is only one hidden layer and the distance between input and output layer is very small. So the gradients calculated during back propagation are not subjected to any residual factors that might result in Vanishing Gradient.