Exam Number: Y3873801

1.1 Train and evaluate a least squares linear regression model predicting the value of variable D from variables A, B and C.

In [15]:

```
import numpy as np
from sklearn import linear_model
from sklearn.metrics import mean_squared_error, r2_score
from sklearn.model_selection import train_test_split
```

In [16]:

```
#loads data from csv and splits it into input data (variables a,b,c) and output data
(d)
file_data = np.loadtxt('data.csv',delimiter=',',skiprows=1)
input_data = file_data[:,:3]
output_data = file_data[:,3]
print ("input data:",np.shape(input_data),"output data:",np.shape(output_data))

#takes a section of the input and label data for training
input_train, input_test, output_train, output_test = train_test_split(input_data, outpu
t_data, test_size=0.2)

#checking sizes of training data and test data
print ("input training data",np.shape(input_train),"output training data",np.shape(outp
ut_train))
print ("input testing data",np.shape(input_test),"output testing data",np.shape(output_
test))
```

```
input data: (95, 3) output data: (95,)
input training data (76, 3) output training data (76,)
input testing data (19, 3) output testing data (19,)
```

In [17]:

```
from sklearn.linear_model import LinearRegression

#Uses ordinary least squares regression
reg = LinearRegression()
#fits to the training data
reg.fit(input_train,output_train)

#predicts variable output from test data input
test_predict = reg.predict(input_test)

print('Mean squared error: %.2f' % mean_squared_error(output_test, test_predict))

print('Coefficient of determination: %.2f'% r2_score(output_test, test_predict))
```

```
Mean squared error: 0.32
Coefficient of determination: 0.87
```

1.2 Repeat the above task after carrying out in turn data normalisation, data scaling and their combination, and evaluate the benefits of each of these 3 types of data preprocessing.

In [18]:

```
from sklearn.preprocessing import scale, normalize
import sklearn.preprocessing as preprocessing
from sklearn.pipeline import make_pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import Normalizer
```

In [19]:

```
#Makes a pipeline that uses a linear regression model after scaling the input
scale_pipe = make_pipeline(StandardScaler(), linear_model.LinearRegression())
scale_pipe.fit(input_train,output_train)

#predicts the output value from the inputs
scaled_test_predict = scale_pipe.predict(input_test)

print('Coefficients: \n', "A:",scale_pipe[-1].coef_[0],"B:",scale_pipe[-1].coef_
[1],"C:",scale_pipe[-1].coef_[2])
print('Mean squared error: %.2f' % mean_squared_error(output_test, scaled_test_predic
t))
print('Coefficient of determination: %.3f' % r2_score(output_test, scaled_test_predic
t))
```

```
Coefficients: 
 A: 0.47065850486754707 B: 0.4706585048675469 C: 0.470658504867547
Mean squared error: 0.32
Coefficient of determination: 0.874
```

In [20]:

```
#Makes a pipeline that uses a linear regression model after normalizing the input
norm_pipe = make_pipeline(Normalizer(), linear_model.LinearRegression())
norm_pipe.fit(input_train,output_train)

#predicts the output value from the inputs
norm_test_predict = norm_pipe.predict(input_test)

print('Coefficients: \n', "A:",norm_pipe[-1].coef_[0],"B:",norm_pipe[-1].coef_[1],"C:",
norm_pipe[-1].coef_[2])
print('Mean squared error: %.2f'% mean_squared_error(output_test, norm_test_predict))
print('Coefficient of determination: %.3f'% r2_score(output_test, norm_test_predict))
```

```
Coefficients: 
 A: 203139471325501.75 B: 903335441806.1875 C: -180265180646191.72
Mean squared error: 0.21
Coefficient of determination: 0.918
```

In [21]:

```
#Makes a pipeline that uses a linear regression model after normalizing then scaling th
e input
norm_scale_pipe = make_pipeline(Normalizer(),StandardScaler(), linear_model.LinearRegre
ssion())
norm_scale_pipe.fit(input_train,output_train)

#predicts the output value from the inputs
norm_scale_test_predict = norm_scale_pipe.predict(input_test)

print('Coefficients: \n', "A:",norm_scale_pipe[-1].coef_[0],"B:",norm_scale_pipe[-1].co
ef_[1],"C:",norm_scale_pipe[-1].coef_[2])

print('Mean squared error: %.2f'
      % mean_squared_error(output_test, norm_scale_test_predict))

print('Coefficient of determination: %.3f'
      % r2_score(output_test, norm_scale_test_predict))
```

```
Coefficients: 
 A: 0.42851015499358064 B: -0.5901648835549285 C: 0.42812070585820855
Mean squared error: 0.21
Coefficient of determination: 0.918
```

1.2A:

Scaling standardises each column so that each column has a mean of 0 and unit variance. This means that you can compare coefficients easier as the columns are all of similar magnitudes.

Normalizing standardises each sample individually so that they all have unit norm which makes it easier for regression for each sample as the spread of the data is more consistent over the whole dataset.

The combination of both of these when first Normalizing then Scaling is you get coefficients of similar magnitude which comes with the scaling, as well as the additional benefit in regression accuracy that comes with normalizing the data first.

1.3 Try to outperform the best result of the previous step by using regularisation (e.g. L1,L2 or Elastic Net). Show how any parameter values are tuned and evaluate the benefits of regularisation

1.3A:

Below I have iterated through each of the main 3 regularisation methods, LassoCV, RidgeCV and ElasticNetCV. For each of these the 'alphas' array which contains a large range of alphas, and the regulariser picked whichever alpha value gave the highest r2 score for the training data. For each of these regularisation regression models, cross validation is used during training. The alpha value chosen by each regulariser is printed below which helped me pick and appropriate range of alpha values to choose from after running through this code multiple times. All regularisers produce similar accuracies which are often higher than the best value seen in the previous question.

In [22]:

```
from sklearn.linear_model import LassoCV,RidgeCV,ElasticNetCV

#Range of different alpha values to try using
alphas = np.logspace(-1,8,num=1000)

#Using regularisation after normalization and scaling
regs = LassoCV, RidgeCV, ElasticNetCV
regsNames = "LassoCV","RidgeCV","ElasticNetCV"
for i, regularizer in enumerate(regs):
  norm_scale_pipe = make_pipeline(Normalizer(),StandardScaler(), regularizer(alphas=alp
has))
  norm_scale_pipe.fit(input_train,output_train)

  norm_scale_test_predict = norm_scale_pipe.predict(input_test)
  print (f"{regsNames[i]} r2 score: %.3f"%r2_score(output_test, norm_scale_test_predic
t))
  print (f"{regsNames[i]} mean squared error: %.3f"%mean_squared_error(output_test, nor
m_scale_test_predict))
  print ("alpha used:",norm_scale_pipe[-1].alpha_)
  print('Coefficients: \n', "A:",norm_scale_pipe[-1].coef_[0],"B:",norm_scale_pipe[-1].
coef_[1],"C:",norm_scale_pipe[-1].coef_[2])
  print ("\n")
```

```
LassoCV r2 score: 0.907
LassoCV mean squared error: 0.234
alpha used: 0.1
Coefficients: 
 A: 0.8079544579861945 B: -0.5372508674300353 C: 0.0


RidgeCV r2 score: 0.917
RidgeCV mean squared error: 0.210
alpha used: 3.330600343624589
Coefficients: 
 A: 0.4495979564921759 B: -0.5260822603621242 C: 0.4493963902101017


ElasticNetCV r2 score: 0.911
ElasticNetCV mean squared error: 0.223
alpha used: 0.1
Coefficients: 
 A: 0.4352015246824772 B: -0.5011419850706843 C: 0.4358468764061655
```

Add a complete set of second-order polynomial basis functions to the original data and train a linear regression with an appropriate type of regularisation to find out whether the new basis functions bring any benefits. Explain briefly (in 1-2 sentences) your reasoning.

1.4A:

Adding polynomial basis functions gave a new best r2 score and lowest mean squared error. Adding polynomial basis functions helps regression as it allows it to catch non linear relationships in the data.

In [23]:

```
from sklearn.preprocessing import PolynomialFeatures
polyFeatures = PolynomialFeatures(degree=2, include_bias=False)
inp_train_poly = polyFeatures.fit_transform(input_train)
inp_test_poly = polyFeatures.transform(input_test)
```

In [24]:

```
from sklearn.linear_model import LassoCV,RidgeCV,ElasticNetCV

alphas = np.logspace(-2,5,num=1000)

#Using regularisation after normalization and scaling
norm_scale_pipe = make_pipeline(Normalizer(),StandardScaler(), RidgeCV(alphas=alphas))
norm_scale_pipe.fit(inp_train_poly,output_train)

norm_scale_test_predict = norm_scale_pipe.predict(inp_test_poly)
print (f"r2 score: %.2f"%r2_score(output_test, norm_scale_test_predict))
print (f"mean squared error: %.2f"%mean_squared_error(output_test, norm_scale_test_pred
ict))
print ("alpha used:",norm_scale_pipe[-1].alpha_)
print('Coefficients: \n', "A:",norm_scale_pipe[-1].coef_[0],"B:",norm_scale_pipe[-1].co
ef_[1],"C:",norm_scale_pipe[-1].coef_[2])
```

```
r2 score: 0.92
mean squared error: 0.19
alpha used: 0.01
Coefficients: 
 A: 0.47263355079972946 B: -0.9993149041256568 C: 0.4542392361965053
```

1.5 Implement an appropriate automated procedure that will train all of the above models and select the model expected to perform best on unseen data with the same distribution as your training data. You need to include a code tile at the end of this section of your Jupyter notebook that attempts to test your final choice of model on a data set stored in a file unseendata.csv and compute R2 for it. The file will have exactly the same format as file data.csv, including the header, but possibly a different overall number of rows. This means you can use a renamed copy of data.csv to debug that part of your code, and to produce the corresponding content for your PDF file (in order to demonstrate that this part of the code is in working order).

1.5A

The first cell below is the train_all function, which takes the training and test data as well as an array called 'combinations' which contains all the different combinations of models used so far and save the output coefficients of the best model to be used on the unseen data later. The second cell below gets all the different combinations and sends them to the function in the first cell.

In [25]:

```
def train_all(combinations,input_train,output_train,input_test,output_test):
  import pickle

  polyFeatures = PolynomialFeatures(degree=2, include_bias=False)
  #creates dataset with polynomial features added
  poly_input_train = polyFeatures.fit_transform(input_train)
  poly_input_test = polyFeatures.transform(input_test)
  bestScore = 0
  scores = []

  #goes through each combination of factors input and finds the factors which give the
best r2 score
  for combination in combinations:
    polynomial_features_included = combination[0]
    preprocessing = combination[1]
    model = combination[2]


    pipeline_list = []
    if polynomial_features_included:
      pipeline_list+=[('polynomial', PolynomialFeatures(degree=2))]

    if preprocessing:
      if isinstance(preprocessing,list):
        pipeline_list+=preprocessing
      else:
        pipeline_list+=[preprocessing]

    pipeline_list += [model]

    pipeline = Pipeline(pipeline_list)

    pipeline.fit(input_train,output_train)
    pipe_predict = pipeline.predict(input_test)

    score = r2_score(output_test, pipe_predict)
    scores.append(score)


    if score > bestScore:
      bestScore = score
      with open('pipeline_stored.pkl', 'wb') as f:
        #stores the coefficients of the best regression pipeline
        pickle.dump(pipeline, f)
      best_pipeline = pipeline

  print ("Pipeline "+str(best_pipeline)+"\nScore "+str(bestScore))
  return best_pipeline
```

In [26]:

```
import itertools
import pickle

from sklearn.pipeline import Pipeline

#The different models to test
models = [("lin_reg",LinearRegression()),
          ("lasso",LassoCV(alphas=alphas)),
          ("ridge",RidgeCV(alphas=alphas)),
          ("elastic_net",ElasticNetCV(alphas=alphas))]

#The possible data preprocessing (polynomial features not included)
preprocessing = [None,
                ("Normalizer",Normalizer()),
                ("Scaler",StandardScaler()),
                [("Normalizer",Normalizer()),("Scaler",StandardScaler())]
                ]

#Whether polynomial features are added to the input data
polynomial_features_included = [True,False]

#creates a list of all the different combinations of the 3 arrays
combinations = list(itertools.product(polynomial_features_included,preprocessing,model
s))


#trains each combination of factors to get the combination with the best r2 score
train_all(combinations,input_train,output_train,input_test,output_test)
```

In [51]:

```
import numpy as np

#loads the unseendata.csv file and extracts the input data and the output data
file_data = np.loadtxt('unseendata.csv',delimiter=',',skiprows=1)
input_data = file_data[:,:3]
output_data = file_data[:,3]

#reads the best model coefficients file created in the previous cells
with open('pipeline_stored.pkl', 'rb') as f:
    best_pipeline = pickle.load(f)

#scores this unseen data using this model
best_pipeline.score(input_data,output_data)
```

Out[51]:

```
0.9276813245548649
```

1.6 Starting with the data in data.csv, find the median value of variable D. Replace all values up to and including the median value with 0, and all values greater than that with 1. Treat the resulting values of D as class labels to train and evaluate a classifier based on logistic regression that takes variables A, B and C as input.

In [49]:

```
import numpy as np
from sklearn.linear_model import LogisticRegression

median = np.median(output_data)
logistic_output = np.empty(0)

#replaces all labels below or equal to the median with 0 and all above the median with
1
for n in output_data:
  if n<=median:
    logistic_output = np.append(logistic_output,0)
  else:
    logistic_output = np.append(logistic_output,1)

logReg = LogisticRegression()

#splits this new dataset into train and test inputs and outputs
log_input_train, log_input_test, log_output_train, log_output_test = train_test_split(i
nput_data, logistic_output, test_size=0.2)

#trains on the training data
logReg.fit(log_input_train,log_output_train)

#predicts on the fitted logistic regression
logPredict = logReg.predict(log_input_test)

#scores this prediction based on the actual output
logReg.score(log_input_test,log_output_test)
```

Out[49]:

```
0.9473684210526315
```

2. Starting with the same data.csv file from Q1, extend the table with 6 additional columns consisting of the product of each pair of the original 4 variables A, B, C and D. Apply principal component analysis (PCA) with a number of principal components (PCs) equal to the number of original variables, i.e. p = 4. Label the resulting principal components in decreasing order of variance as PC1. . .PC4 and list the linear equations showing how each of them is calculated from the 10 input variables. Describe which variables affect most strongly each of the 4 principal components, highlighting any notable findings and providing plausible explanations for them.

In [29]:

```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.decomposition import PCA
polyFeatures = PolynomialFeatures(degree=2, include_bias=False,interaction_only=True)
new_file_data = polyFeatures.fit_transform(file_data)

pca = PCA(n_components=4)
pca.fit(new_file_data)
newData = pca.fit_transform(new_file_data)

PC1 = pca.components_[0]
PC2 = pca.components_[1]
PC3 = pca.components_[2]
PC4 = pca.components_[3]

for i, dim in enumerate(('PC1','PC2','PC3','PC4')):
  dimVar = np.var(newData[:,i])
  print ("%s variance %.3f"%(dim,dimVar))
```

```
PC1 variance 20892.780
PC2 variance 80.399
PC3 variance 10.508
PC4 variance 0.849
```

Question 3: Hand drawn tree symbol classification

In [30]:

```
"""
Assuming symbols_dataset.zip is in current directory /content
The below code creates a directory symbols_dataset_unzipped and then unzips the zip f
ile
into this directory
"""
!mkdir symbols_dataset_unzipped
!unzip symbols_dataset.zip -d symbols_dataset_unzipped
```

The cell below reads all the file names from each folder and adds the appropriate label to the labels array. Then a new dataset class TreeDrawingDataset is created, and an instance of this class is created using the image file names and labels data.

Then subsets of this dataset are made for training and testing and dataloaders for these datasets are created. A series of image augmentation transforms are also added here to avoid overfitting to the training data, better generalisation and creates different samples for each epoch.

The augmentations chosen include random rotation, random horizontal flip and random crop to help the CNN become more translation invarient and detect patterns in the data more generally. This should also have the other important impact of providing additional variations of the training examples for the classes which do not have many training images. This should help the network identify the patterns and shapes which represent these classes and identify them better. After experimentation I realized that a larger batch size also helped with this, as smaller batches often led to generalization where the minority classes were all predicted as one of the majority classes. I also normalized the input tensors to standardize the data to make the training easier for the network.

```
from torch.utils.data import Dataset, DataLoader, random_split
import torch.nn as nn
import math
import torch
import os
import numpy as np
from PIL import Image

classes = [0,1,2,3,4]
file_list = []
labels = []

#extracts all the image file paths
for class_num in classes:
  files_path = f'/content/symbols_dataset_unzipped/class_{class_num}'
  class_num_files = os.listdir(files_path)
  files_with_path = [os.path.join(files_path, file) for file in class_num_files if '.ti
f' in file]
  file_list = file_list + files_with_path
  labels = labels+[class_num]*len(class_num_files)


file_list = np.array(file_list)
labels = np.array(labels)

import torchvision.transforms as transforms


#data augmentation to help generalize the network training
transform = transforms.Compose([transforms.ToTensor(),
                                transforms.RandomRotation((10,30)),
                                transforms.RandomHorizontalFlip(0.5),
                                transforms.RandomCrop(40),
                                transforms.Normalize(0.5,0.5)])


class TreeDrawingDataset(Dataset):
  def __init__(self,data,labels,transform=None):
    self.data = data
    self.labels = labels
    self.transform = transform

  def __getitem__(self,idx):
    label = self.labels[idx]
    image = Image.open(self.data[idx]).convert('L')
    if self.transform:
      image = self.transform(image)
    return image,label

  def __len__(self):
    return len(self.data)

#dataset object of the whole image list
tree_dataset = TreeDrawingDataset(file_list,labels,transform)

#creates subsets of the dataset randomly with an 80/20 split for training and testing
training_dataset, testing_dataset = random_split(tree_dataset,[0.8,0.2])

#dataloader for training purposes, using a large batch size of 200
training_dataloader = torch.utils.data.DataLoader(training_dataset, batch_size=200, shu
ffle=True)
```

```
testing_dataloader = torch.utils.data.DataLoader(testing_dataset, batch_size=50, shuffl
e=True)
```

For the classifying convolutional neural network, I implented convolutions to capture patterns in the data, with smaller sized convolutions used to capture more detailed features after a larger sized convolution to capture larger features. After each convolution layer I added a batch normalization layer to normalize the data after each output followed by the ReLU activation function. I then chose a MaxPool to follow this as this firstly decreases the resolution of the input so simplifies input to layers with higher numbers of channels and avoids computation speeds suffering too heavily, and secondly because MaxPool is good at detecting major features in the data which should be clearer than if using average pooling. Once I have increased the output channels to 48 I then put the data through fully connected layers. After each linear layer I also normalized and passed it through a ReLU layer. In addition to these, I added two dropout layers to reduce overfitting due to the large number of parameters being used.

In [32]:

```
class TreeClassifier(nn.Module):
    def __init__(self):
        super(TreeClassifier, self).__init__()
        self.conv = nn.Sequential(
            #5x5 Convolution to detect larger features
            nn.Conv2d(in_channels=1, out_channels=12, kernel_size=5, stride=1, padding=2),
            #Normalizes the batch which helps training
            nn.BatchNorm2d(12),
            #ReLU Activation function
            nn.ReLU(),
            #Max pooling to simplify resolution and identify key features
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=12, out_channels=24, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(24),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),
            nn.Conv2d(in_channels=24, out_channels=48, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(48),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2)
        )

        self.fc = nn.Sequential(
            #linear fully connected layers
            nn.Linear(in_features=5*5*48,out_features=150),
            nn.BatchNorm1d(150),
            nn.ReLU(),
            #Dropout layer to prevent overfitting
            nn.Dropout(p=0.5),
            nn.Linear(in_features=150,out_features=75),
            nn.BatchNorm1d(75),
            nn.ReLU(),
            nn.Dropout(p=0.5),
            nn.Linear(in_features=75,out_features=5)
        )

    def forward(self, x):
        x = self.conv(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x

model = TreeClassifier()
```

After some experimentation, I found that the Adam optimiser was the best for this task, and a small learning rate of 0.0001 also yielded the best results. A weight decay of 0.1 was also added to this optimiser to further prevent against overfitting and help the network to identify unseen data.

In [33]:

```
#15 epochs was found to be enough to converge to a low loss value
epochs = 15
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = TreeClassifier()
model = model.to(device)
model.train()
optim = torch.optim.Adam(model.parameters(), lr = 0.0001,weight_decay=0.1)

# Using cross entropy loss loss function
lossF = nn.CrossEntropyLoss()

for epoch in range(epochs):
  totalLoss = 0
  for i, (images, labels) in enumerate(training_dataloader):
    images, labels = images.cuda(), labels.cuda()
    output = model(images)
    loss = lossF(output,labels)
    optim.zero_grad()
    loss.backward()
    optim.step()
    totalLoss += loss
  print('Total loss for epoch %d: %.2f'%(epoch+1,totalLoss))
```

```
Total loss for epoch 1: 77.17
Total loss for epoch 2: 52.11
Total loss for epoch 3: 40.15
Total loss for epoch 4: 32.03
Total loss for epoch 5: 26.76
Total loss for epoch 6: 22.50
Total loss for epoch 7: 19.32
Total loss for epoch 8: 16.77
Total loss for epoch 9: 14.80
Total loss for epoch 10: 13.64
Total loss for epoch 11: 12.42
Total loss for epoch 12: 11.33
Total loss for epoch 13: 10.36
Total loss for epoch 14: 10.07
Total loss for epoch 15: 9.52
```

In [34]:

```
correct = 0
total = 0

model.eval()
for images, labels in testing_dataloader:
    images, labels = images.cuda(), labels.cuda()
    output = model(images)
    test_prediction = torch.argmax(output, 1)
    correct += (test_prediction == labels).sum()
    total += float(labels.size(0))
accuracy = correct/total

print('Test Accuracy of the model on the test images: %.2f' % accuracy)
```

```
Test Accuracy of the model on the test images: 0.99
```

In [42]:

```
torch.save(model.state_dict(), 'weights.pkl')
```

Q4 Generating tree symbols

In the cell below, I setup a new dataset using the TreeDrawingDataset class, however this time without the 0 class which contained non-tree images. For the batch size, I settled on a larger batch size of 150 as this seemed to produce more consistently good outputs and fast computation. For the latent input size I used 100, this was a good balance between not enough detail and requiring a lot of computation. Similarly I found values of 60 for the size of the feature maps to be a good balance, it allowed me to scale up the number of channels without getting to too huge numbers of parameters well and provided a good level of depth and quality output. 20 epochs was the number at which I did not see a lot of improvement in output image quality and I found a learning rate of 0.0001 to converge at a decent pace.

For image augmentation transformations, I normalized the input to help the network training, however other transformations I attempted affected the quality of the output generated images too much.

The beta1 parameter for the Adam optimizer I kept at a low value as I found this value being kept higher led to a slower convergence as the moving average from previous gradients was affecting it too much.

In [36]:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets
import torchvision.transforms as transforms
import numpy as np

%matplotlib inline
import matplotlib.pyplot as plt

#not including 0 class
classes = [1,2,3,4]
file_list = []
labels = []
for class_num in classes:
  files_path = f'/content/symbols_dataset_unzipped/class_{class_num}'
  class_num_files = os.listdir(files_path)
  files_with_path = [os.path.join(files_path, file) for file in class_num_files if '.ti
f' in file]
  file_list = file_list + files_with_path
  labels = labels+[class_num]*len(class_num_files)


file_list = np.array(file_list)
labels = np.array(labels)


batch_size = 150

#latent input vector size
nz = 100

#sizes for the features maps in the generator and discriminator
ngf = 60
ndf = 60

#number of epochs used for training
epochs = 20

#optimisers learning rate
lr = 0.0001
beta1 = 0.1 # Beta1 Adam param

# Set up the dataset and dataloader


transform = transforms.Compose([transforms.ToTensor(),
                                transforms.Normalize(0.5,0.5)])
tree_dataset_no_zero_class = TreeDrawingDataset(file_list,labels,transform)
dataloader = torch.utils.data.DataLoader(tree_dataset_no_zero_class,batch_size=150,shuf
fle=True,num_workers=1)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

For the generator, I used a sequence of transposed convolutions to increase the size of the input to a 48x48 image output. Firstly, I included a large 5 size kernel convolution while increasing the number of channels to 60*6 = 360. From this point with each convolution transposition I decreased the number of channels until the output was a singular channel with output size 48x48. After each convolution I normalized the batch to help with training and then passed it through a LeakyReLU layer. This should increase the stability of the network training over a normal ReLU.

In [37]:

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.main = nn.Sequential(
            #increases the image size x3 = 3x3
            nn.ConvTranspose2d(in_channels=nz, out_channels=ngf * 6, kernel_size=5, str
ide=2, padding = 1, bias=False),
            nn.BatchNorm2d(ngf * 6),
            nn.LeakyReLU(0.1, inplace=True),
            #increases the image size x 2 = 6x6
            nn.ConvTranspose2d( in_channels=ngf *6, out_channels=ngf*4, kernel_size=4,
stride=2, padding = 1, bias=False),
            nn.BatchNorm2d(ngf*4),
            nn.LeakyReLU(0.1, inplace=True),
            #increases the image size x 2 = 12x12
            nn.ConvTranspose2d( in_channels=ngf *4, out_channels=ngf*2, kernel_size=4,
stride=2, padding = 1, bias=False),
            nn.BatchNorm2d(ngf*2),
            nn.LeakyReLU(0.1, inplace=True),
            #increases the image size x 2 = 24x24
            nn.ConvTranspose2d(in_channels=ngf*2, out_channels=ngf, kernel_size=4, stri
de=2, padding = 1, bias=False),
            nn.BatchNorm2d(ngf),
            nn.LeakyReLU(0.1, inplace=True),
            #increases the image size x 2 = 48x48
            nn.ConvTranspose2d( in_channels=ngf, out_channels=1, kernel_size=4, stride=
2, padding = 1, bias=False),
            nn.Tanh()
        )

    def forward(self, x):
        return self.main(x)

netG = Generator()
netG = netG.to(device)
```

For the discriminator, I initially increased the number of channels through convolution layers to add depth to the network and the convolution filters, before decreasing the number of channels before the output where it outputs a singular channel and an output of size 1, i.e. whether the input image is real or fake. I used a MaxPool layer to help decrease the resolution without requiring another convolution layer, this helped decrease complexity of the network and the number of parameters required as well as helping highlight key features of the real images. The main sequence also ends in a sigmoid function to give the output of real or fake as a probability that it is real.

In [38]:

```
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.main = nn.Sequential(
            #increases the channel size to ndf
            nn.Conv2d(in_channels=1, out_channels=ndf, kernel_size=3, stride=1, padding
= 1, bias=False),
            nn.BatchNorm2d(ndf),
            nn.LeakyReLU(0.1, inplace=True),

            #decreases image size by 2 to 24x24
            nn.Conv2d(in_channels=ndf, out_channels=ndf * 2, kernel_size=4, stride=2, p
adding = 1, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.1, inplace=True),

            #decreases image size by 2 to 12x12
            nn.Conv2d(in_channels=ndf * 2, out_channels=ndf * 4, kernel_size=4, stride=
2, padding = 1, bias=False),

            #decreases image size by 2 to 6x6
            nn.MaxPool2d(kernel_size=2),
            nn.BatchNorm2d(ndf * 4),
            nn.LeakyReLU(0.1, inplace=True),

            #decreases image size by 2 to 3x3
            nn.Conv2d(in_channels=ndf * 4, out_channels=ndf*2, kernel_size=4, stride=1,
padding = 0, bias=False),
            nn.BatchNorm2d(ndf * 2),
            nn.LeakyReLU(0.1, inplace=True),

            #decreases image size by 3 to 1x1
            nn.Conv2d(in_channels=ndf * 2, out_channels=1, kernel_size=3, stride=1, pad
ding = 0, bias=False),
            nn.Sigmoid()

        )

    def forward(self, x):
        return self.main(x)

netD = Discriminator()
netD = netD.to(device)
```

Below is the training loop to train the discriminator and generator networks

In [39]:

```
real_label = 1.
fake_label = 0.

#Using binary cross entropy loss
criterion = nn.BCELoss()

#set up optimizers
DiscOptim = optim.Adam(netD.parameters(), lr=lr, betas=(beta1, 0.999))
GenOptim = optim.Adam(netG.parameters(), lr=lr, betas=(beta1, 0.999))


for epoch in range(epochs):
    for i, (images,labels) in enumerate(dataloader, 0):
        """
        Loops through real images to train discriminator
        """
        netD.zero_grad()
        real_images = images.to(device)
        label = torch.full((len(images),), real_label, dtype=torch.float, device=devic
e)
        output = netD(real_images)
        output = output.view(-1)
        errD_real = criterion(output, label)
        errD_real.backward()
        D_x = output.mean().item()

        """
        Loops through fake images generated by generator to train discriminator
        """

        z = torch.randn(len(images), nz, 1, 1, device=device)
        fake = netG(z)
        label.fill_(fake_label)
        output = netD(fake.detach())
        output = output.view(-1)
        errD_fake = criterion(output, label)
        errD_fake.backward()
        D_G_z1 = output.mean().item()
        errD = errD_real + errD_fake
        DiscOptim.step()

        """
        Inputs generated images into the discriminator to get loss
        and update generator
        """
        netG.zero_grad()
        label.fill_(real_label)
        output = netD(fake).view(-1)
        errG = criterion(output, label)
        errG.backward()
        D_G_z2 = output.mean().item()
        GenOptim.step()

        if i % 50 == 0:
            print('[%d/%d][%d/%d]\tLoss_D: %.4f\tLoss_G: %.4f\tD(x): %.4f\tD(G(z)): %.4
f / %.4f'
                  % (epoch+1, epochs, i, len(dataloader),
                     errD.item(), errG.item(), D_x, D_G_z1, D_G_z2))
```

```
[1/20][0/73]    Loss_D: 1.5055  Loss_G: 1.0038  D(x): 0.4455    D(G(z)):
0.4810 / 0.3742
[1/20][50/73]   Loss_D: 0.0397  Loss_G: 4.4583  D(x): 0.9865    D(G(z)):
0.0256 / 0.0120
[2/20][0/73]    Loss_D: 0.0218  Loss_G: 5.0710  D(x): 0.9919    D(G(z)):
0.0134 / 0.0064
[2/20][50/73]   Loss_D: 0.0117  Loss_G: 5.7273  D(x): 0.9962    D(G(z)):
0.0078 / 0.0033
[3/20][0/73]    Loss_D: 0.0104  Loss_G: 5.8751  D(x): 0.9963    D(G(z)):
0.0066 / 0.0028
[3/20][50/73]   Loss_D: 0.0069  Loss_G: 6.0636  D(x): 0.9972    D(G(z)):
0.0041 / 0.0023
[4/20][0/73]    Loss_D: 0.0055  Loss_G: 6.2188  D(x): 0.9982    D(G(z)):
0.0037 / 0.0020
[4/20][50/73]   Loss_D: 0.0038  Loss_G: 6.5569  D(x): 0.9985    D(G(z)):
0.0023 / 0.0014
[5/20][0/73]    Loss_D: 0.0048  Loss_G: 6.4978  D(x): 0.9976    D(G(z)):
0.0024 / 0.0015
[5/20][50/73]   Loss_D: 0.0362  Loss_G: 7.0663  D(x): 0.9959    D(G(z)):
0.0314 / 0.0009
[6/20][0/73]    Loss_D: 0.0142  Loss_G: 6.0005  D(x): 0.9975    D(G(z)):
0.0117 / 0.0025
[6/20][50/73]   Loss_D: 0.0145  Loss_G: 6.8070  D(x): 0.9964    D(G(z)):
0.0108 / 0.0011
[7/20][0/73]    Loss_D: 0.0451  Loss_G: 5.6396  D(x): 0.9926    D(G(z)):
0.0368 / 0.0045
[7/20][50/73]   Loss_D: 0.2104  Loss_G: 3.1139  D(x): 0.8867    D(G(z)):
0.0764 / 0.0470
[8/20][0/73]    Loss_D: 0.2264  Loss_G: 4.2998  D(x): 0.9731    D(G(z)):
0.1747 / 0.0156
[8/20][50/73]   Loss_D: 0.0447  Loss_G: 5.0816  D(x): 0.9694    D(G(z)):
0.0107 / 0.0072
[9/20][0/73]    Loss_D: 0.0823  Loss_G: 4.5033  D(x): 0.9702    D(G(z)):
0.0474 / 0.0120
[9/20][50/73]   Loss_D: 0.2376  Loss_G: 5.1369  D(x): 0.9168    D(G(z)):
0.1251 / 0.0066
[10/20][0/73]   Loss_D: 0.2919  Loss_G: 2.8418  D(x): 0.8393    D(G(z)):
0.0840 / 0.0710
[10/20][50/73]  Loss_D: 0.3693  Loss_G: 2.5412  D(x): 0.8019    D(G(z)):
0.1124 / 0.0883
[11/20][0/73]   Loss_D: 0.6214  Loss_G: 0.7673  D(x): 0.5972    D(G(z)):
0.0391 / 0.4923
[11/20][50/73]  Loss_D: 0.3640  Loss_G: 2.1375  D(x): 0.7602    D(G(z)):
0.0617 / 0.1405
[12/20][0/73]   Loss_D: 0.3970  Loss_G: 2.1100  D(x): 0.7824    D(G(z)):
0.1143 / 0.1443
[12/20][50/73]  Loss_D: 0.3199  Loss_G: 3.9073  D(x): 0.9041    D(G(z)):
0.1809 / 0.0268
[13/20][0/73]   Loss_D: 0.1795  Loss_G: 3.4775  D(x): 0.9615    D(G(z)):
0.1255 / 0.0369
[13/20][50/73]  Loss_D: 0.2483  Loss_G: 2.9457  D(x): 0.9059    D(G(z)):
0.1275 / 0.0625
[14/20][0/73]   Loss_D: 0.3137  Loss_G: 3.5435  D(x): 0.8899    D(G(z)):
0.1652 / 0.0373
[14/20][50/73]  Loss_D: 0.3450  Loss_G: 2.1210  D(x): 0.7605    D(G(z)):
0.0150 / 0.1572
[15/20][0/73]   Loss_D: 0.1970  Loss_G: 3.5043  D(x): 0.9437    D(G(z)):
0.1222 / 0.0383
[15/20][50/73]  Loss_D: 0.2589  Loss_G: 2.5414  D(x): 0.8180    D(G(z)):
0.0278 / 0.1030
[16/20][0/73]   Loss_D: 0.3068  Loss_G: 4.7313  D(x): 0.9574    D(G(z)):
```

```
0.2103 / 0.0141
[16/20][50/73]  Loss_D: 0.3563  Loss_G: 2.6862  D(x): 0.8486    D(G(z)):
0.1555 / 0.0857
[17/20][0/73]   Loss_D: 0.1487  Loss_G: 3.3406  D(x): 0.9301    D(G(z)):
0.0646 / 0.0481
[17/20][50/73]  Loss_D: 0.1619  Loss_G: 4.0110  D(x): 0.9437    D(G(z)):
0.0926 / 0.0230
[18/20][0/73]   Loss_D: 0.1764  Loss_G: 3.4895  D(x): 0.9391    D(G(z)):
0.0997 / 0.0378
[18/20][50/73]  Loss_D: 0.1430  Loss_G: 3.2219  D(x): 0.9078    D(G(z)):
0.0369 / 0.0521
[19/20][0/73]   Loss_D: 0.1408  Loss_G: 3.6027  D(x): 0.9845    D(G(z)):
0.1096 / 0.0410
[19/20][50/73]  Loss_D: 0.1249  Loss_G: 3.9562  D(x): 0.9469    D(G(z)):
0.0627 / 0.0259
[20/20][0/73]   Loss_D: 1.2883  Loss_G: 3.8892  D(x): 0.3651    D(G(z)):
0.0014 / 0.0382
[20/20][50/73]  Loss_D: 0.4965  Loss_G: 5.3631  D(x): 0.9942    D(G(z)):
0.3432 / 0.0106
```

Below here are 8 generated images using the generator network on a random latent sample

In [48]:

```python
z = torch.randn(8,nz,1,1,device=device)
images = netG(z)

figure = plt.figure(figsize=(15, 10))
cols, rows = 8, 1
for i in range(cols * rows):
    figure.add_subplot(rows, cols, i+1)
    plt.axis("off")
    plt.imshow(images[i,:].cpu().detach().squeeze(), cmap="gray")
plt.show()
```

Below are two images (far left and far right) linearly interpolated with 5 interpolated samples.

In [41]:

```
latentSpace1 = torch.randn(1, nz, 1, 1, device=device)
latentSpace2 = torch.randn(1, nz, 1, 1, device=device)

interpolatedImages = torch.zeros(7,nz,1,1,device=device)
for i in range(7):
  Image1Proportion = i/6
  Image2Proportion = 1-Image1Proportion
  interpolatedImages[i,:,:,:] = Image1Proportion*latentSpace1 + Image2Proportion*latent
Space2
images = netG(interpolatedImages)

figure = plt.figure(figsize=(20, 5))
for i in range(7):
    figure.add_subplot(1, 7, i+1)
    plt.axis("off")
    plt.imshow(images[i,:].squeeze().cpu().detach(), cmap="gray")
plt.show()
```