lab-4-perceptron-adaline-torchDiff

March 4, 2023

[182]:
```python
import numpy as np
import pandas as pd
import torch
import matplotlib.pyplot as plt
```

0.1 Implementing numpy version of perceptron

[263]:
```python
data = np.genfromtxt('data/perceptron_toydata.txt', delimiter='\t')
X, y = data[:, :2], data[:, 2]
y = y.astype(int)

print('Class label counts:', np.bincount(y))
print('X.shape:', X.shape)
print('y.shape:', y.shape)

# Shuffling & train/test split
shuffle_idx = np.arange(y.shape[0])
shuffle_rng = np.random.RandomState(123)
shuffle_rng.shuffle(shuffle_idx)
X, y = X[shuffle_idx], y[shuffle_idx]

X_train, X_test = X[shuffle_idx[:70]], X[shuffle_idx[70:]]
y_train, y_test = y[shuffle_idx[:70]], y[shuffle_idx[70:]]

# Normalize (mean zero, unit variance)
mu, sigma = X_train.mean(axis=0), X_train.std(axis=0)
X_train = (X_train - mu) / sigma
X_test = (X_test - mu) / sigma
```

```
Class label counts: [50 50]
X.shape: (100, 2)
y.shape: (100,)
```

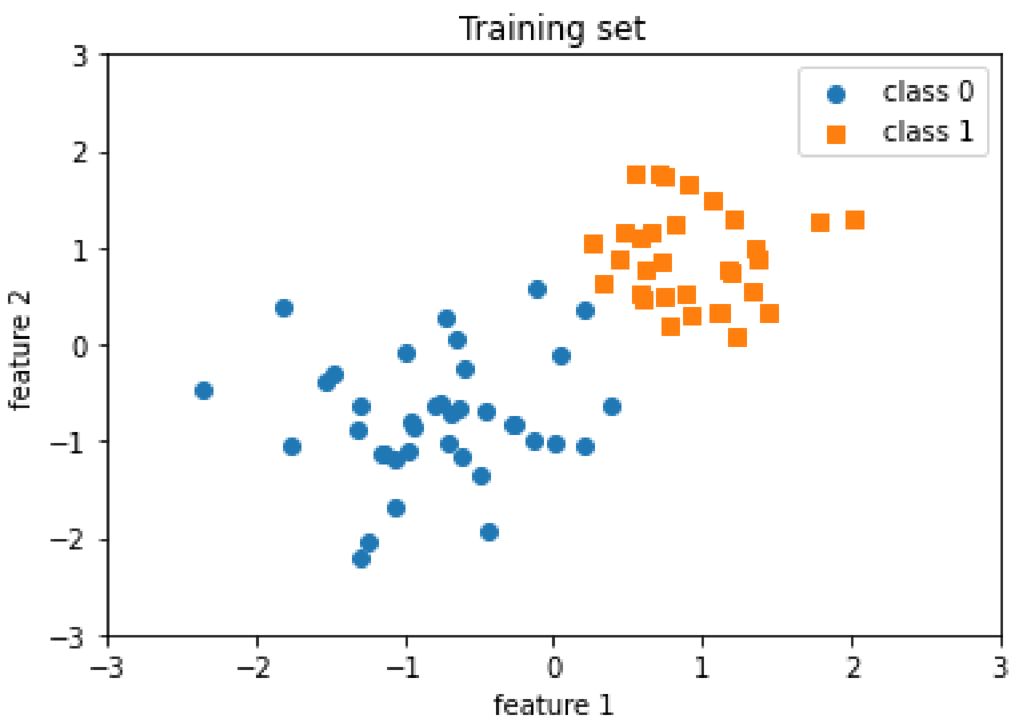
[19]:
```python
# After stadardisation
print("Mean = {} ".format(np.mean(X_train, axis=0)))
print("STD = {}".format(np.std(X_train, axis=0)))
```

```
Mean = [2.06184276e-17 7.93016446e-18]
STD = [1. 1.]
```

[24]:
```python
X_train.shape
```

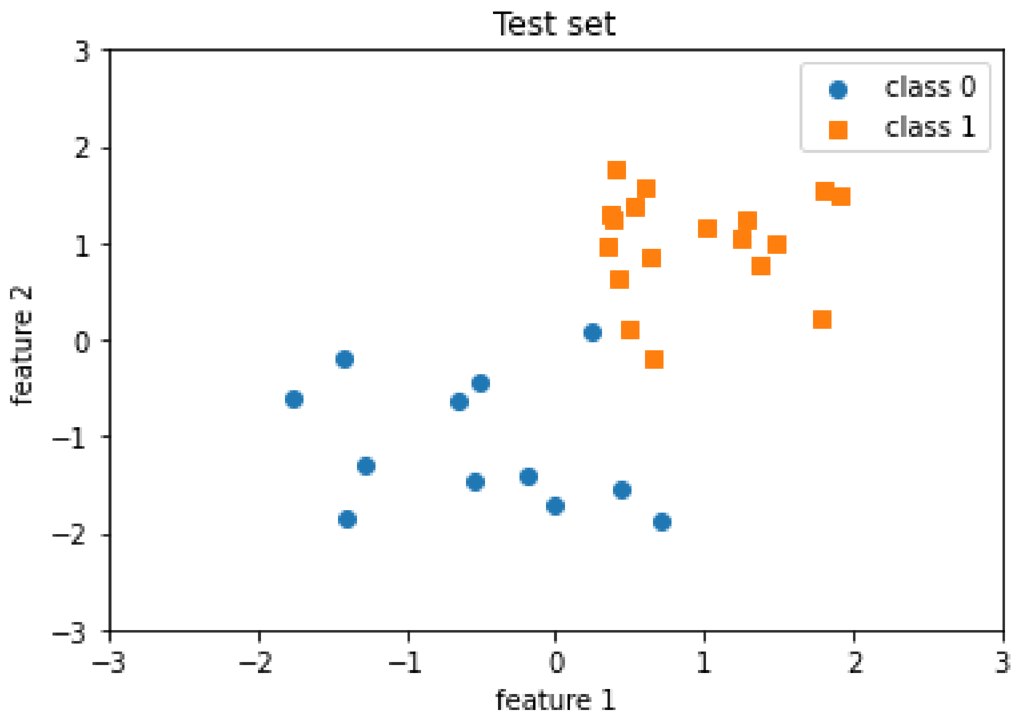
[24]: (70, 2)

[28]:
```python
plt.scatter(X_train[y_train==0, 0], X_train[y_train==0, 1], label='class 0',␣
 ↪marker='o')
plt.scatter(X_train[y_train==1, 0], X_train[y_train==1, 1], label='class 1',␣
 ↪marker='s')
plt.title('Training set')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.legend()
plt.show()
```



[32]:
```python
plt.scatter(X_test[y_test==0, 0], X_test[y_test==0, 1], label='class 0',␣
 ↪marker='o')
plt.scatter(X_test[y_test==1, 0], X_test[y_test==1, 1], label='class 1',␣
 ↪marker='s')
```

```
plt.title('Test set')
plt.xlabel('feature 1')
plt.ylabel('feature 2')
plt.xlim([-3, 3])
plt.ylim([-3, 3])
plt.legend()
plt.show()
```



[176]:
```
class Perceptron():
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = np.zeros((num_features,1), dtype=float)
        self.bias = np.zeros(1, dtype=float)

    def forward(self, x):
        linear = np.dot(x, self.weights) + self.bias
        linear = linear.flatten()
        predictions = np.where(linear>0,1,0)
        return predictions

    def backward(self, x,y):
        predictions = self.forward(x)
        errors = y - predictions
```

```
        return errors

    def newTrain(self, x,y,epochs):
        for e in range(epochs):
            error = self.backward(x,y).reshape(x.shape[0],1)
            error = error*x
            error = np.sum(error, axis=0).reshape(-1,1)
            self.weights+=error
            self.bias+=error[0,0]

    def train(self, x,y,epochs):
        for e in range(epochs):
            for i in range(y.shape[0]):
                error = self.backward(x[i].reshape(1, self.num_features),y[i]).
↪reshape(-1)
                self.weights += (error*x[i]).reshape(self.num_features,1)
                self.bias += error
    def evaluate(self,x,y):
        predictions = self.forward(x).reshape(-1)
        accuracy = np.sum(predictions == y) / y.shape[0]
        return accuracy
```

[177]:
```
ppn = Perceptron(num_features=2)

ppn.newTrain(X_train, y_train, epochs=1)

print('Model parameters:\n\n')
print('  Weights: %s\n' % ppn.weights)
print('  Bias: %s\n' % ppn.bias)
```

```
(2, 1)
Model parameters:



  Weights: [[29.83124073]
 [28.73135694]]

  Bias: [29.83124073]
```

[179]:
```
ppn.evaluate(X_test, y_test)*100
```

[179]: 93.33333333333333

0.2 Fully Connected Linear layer

[61]:
```
X = torch.arange(50, dtype=torch.float32).view(10,5)
```

[65]:
```
fc_layer = torch.nn.Linear(in_features=5, out_features = 3)
```

[66]:
```
fc_layer.weight.shape
```

[66]:
```
torch.Size([3, 5])
```

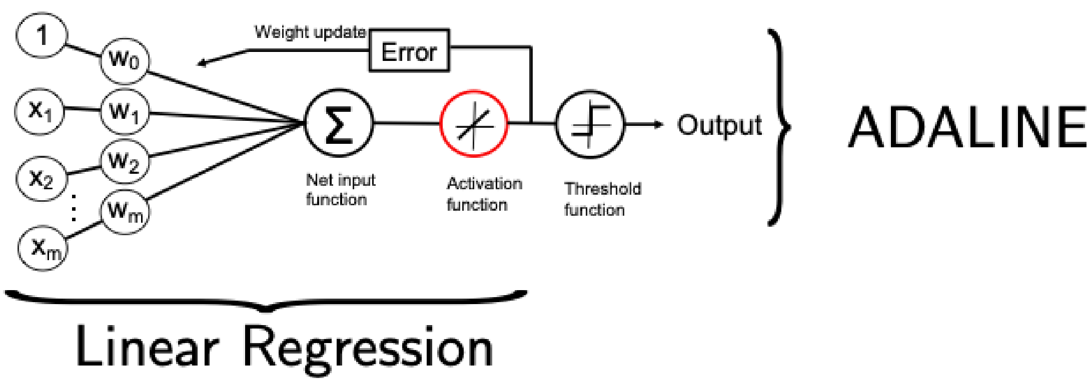
[67]:
```
fc_layer.bias.shape
```

[67]:
```
torch.Size([3])
```

[68]:
```
fc_layer(X).shape
```

[68]:
```
torch.Size([10, 3])
```

0.3 Adaline



[269]:
```
# Load the data
df = pd.read_csv('data/linreg-data.csv', index_col=0)
df.tail()
```

[269]:

	x1	x2	y
995	-0.942094	-0.835856	-22.324428
996	1.222445	-0.403177	-52.121493
997	-0.112466	-1.688230	-57.043196
998	-0.403459	-0.412272	-27.701833
999	0.021351	-0.499017	-9.804714

[270]:
```
X = torch.tensor(df[['x1', 'x2']].values, dtype=torch.float)
y = torch.tensor(df['y'].values, dtype=torch.float)

shuffle_idx = torch.randperm(y.size(0), dtype=torch.long)
```

```
X, y = X[shuffle_idx], y[shuffle_idx]

percent70 = int(shuffle_idx.size(0)*0.7)

X_train, X_test = X[shuffle_idx[:percent70]], X[shuffle_idx[percent70:]]
y_train, y_test = y[shuffle_idx[:percent70]], y[shuffle_idx[percent70:]]

# Normalize (mean zero, unit variance)

mu, sigma = X_train.mean(dim=0), X_train.std(dim=0)
X_train = (X_train - mu) / sigma
X_test = (X_test - mu) / sigma
```

[193]:
```
class LinearRegression1():
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = torch.zeros(num_features, 1, dtype=torch.float)
        self.bias = torch.zeros(1, dtype=torch.float)

    def forward(self, x):
        netinputs = torch.add(torch.mm(x, self.weights), self.bias)
        activations = netinputs
        return activations.view(-1)

    def backward(self, x, yhat, y):

        grad_loss_yhat = 2*(yhat - y) #derivative of (yhat-y)**2 wrt yhat.␣
 ↪Verified

        grad_yhat_weights = x
        grad_yhat_bias = 1.

        # Chain rule: inner times outer
        grad_loss_weights =  torch.mm(grad_yhat_weights.t(),grad_loss_yhat.
 ↪view(-1, 1)) / y.size(0)

        grad_loss_bias = torch.sum(grad_yhat_bias*grad_loss_yhat) / y.size(0)

        # return negative gradient
        return (-1)*grad_loss_weights, (-1)*grad_loss_bias
```

[194]:
```
#Define Training and Evaluation Functions

# mean squared error
def loss(yhat, y):
    return torch.mean((yhat - y)**2)
```

```python
# batch gradient descent
def train(model, x, y, num_epochs, learning_rate=0.01):
    cost = []
    for e in range(num_epochs):

        #### Compute outputs ####
        yhat = model.forward(x)

        #### Compute gradients ####
        negative_grad_w, negative_grad_b = model.backward(x, yhat, y)

        #### Update weights ####
        model.weights += learning_rate * negative_grad_w
        model.bias += learning_rate * negative_grad_b

        #### Logging ####
        yhat = model.forward(x) # not that this is a bit wasteful here
        curr_loss = loss(yhat, y)
        print('Epoch: %03d' % (e+1), end="")
        print(' | MSE: %.5f' % curr_loss)
        cost.append(curr_loss)

    return cost
```

[195]:
```python
model = LinearRegression1(num_features=X_train.size(1))
cost = train(model, X_train, y_train, num_epochs=100, learning_rate=0.05)
```

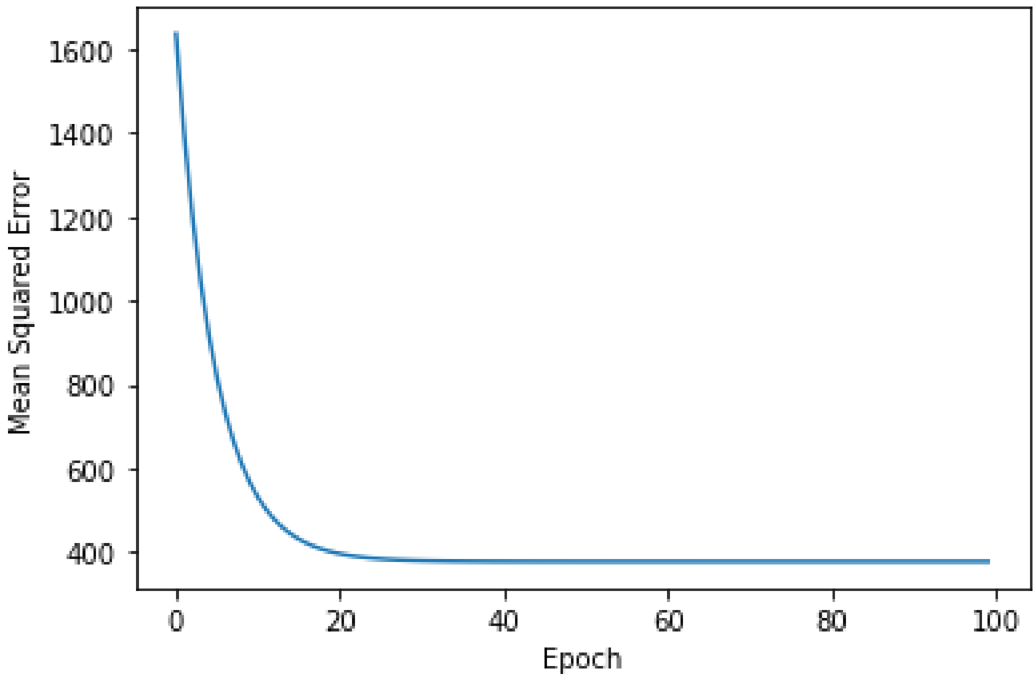
```
Epoch: 001 | MSE: 1636.67786
Epoch: 002 | MSE: 1397.68274
Epoch: 003 | MSE: 1204.04407
Epoch: 004 | MSE: 1047.15381
Epoch: 005 | MSE: 920.03729
Epoch: 006 | MSE: 817.04437
Epoch: 007 | MSE: 733.59680
Epoch: 008 | MSE: 665.98505
Epoch: 009 | MSE: 611.20398
Epoch: 010 | MSE: 566.81860
Epoch: 011 | MSE: 530.85614
Epoch: 012 | MSE: 501.71808
Epoch: 013 | MSE: 478.10925
Epoch: 014 | MSE: 458.98059
Epoch: 015 | MSE: 443.48166
Epoch: 016 | MSE: 430.92383
Epoch: 017 | MSE: 420.74896
Epoch: 018 | MSE: 412.50482
```

```
Epoch: 019 | MSE: 405.82495
Epoch: 020 | MSE: 400.41263
Epoch: 021 | MSE: 396.02731
Epoch: 022 | MSE: 392.47412
Epoch: 023 | MSE: 389.59509
Epoch: 024 | MSE: 387.26233
Epoch: 025 | MSE: 385.37228
Epoch: 026 | MSE: 383.84076
Epoch: 027 | MSE: 382.59982
Epoch: 028 | MSE: 381.59442
Epoch: 029 | MSE: 380.77972
Epoch: 030 | MSE: 380.11963
Epoch: 031 | MSE: 379.58475
Epoch: 032 | MSE: 379.15140
Epoch: 033 | MSE: 378.80023
Epoch: 034 | MSE: 378.51572
Epoch: 035 | MSE: 378.28519
Epoch: 036 | MSE: 378.09836
Epoch: 037 | MSE: 377.94696
Epoch: 038 | MSE: 377.82428
Epoch: 039 | MSE: 377.72501
Epoch: 040 | MSE: 377.64441
Epoch: 041 | MSE: 377.57919
Epoch: 042 | MSE: 377.52631
Epoch: 043 | MSE: 377.48343
Epoch: 044 | MSE: 377.44876
Epoch: 045 | MSE: 377.42062
Epoch: 046 | MSE: 377.39786
Epoch: 047 | MSE: 377.37936
Epoch: 048 | MSE: 377.36438
Epoch: 049 | MSE: 377.35226
Epoch: 050 | MSE: 377.34247
Epoch: 051 | MSE: 377.33441
Epoch: 052 | MSE: 377.32803
Epoch: 053 | MSE: 377.32281
Epoch: 054 | MSE: 377.31857
Epoch: 055 | MSE: 377.31519
Epoch: 056 | MSE: 377.31238
Epoch: 057 | MSE: 377.31009
Epoch: 058 | MSE: 377.30829
Epoch: 059 | MSE: 377.30679
Epoch: 060 | MSE: 377.30557
Epoch: 061 | MSE: 377.30466
Epoch: 062 | MSE: 377.30383
Epoch: 063 | MSE: 377.30322
Epoch: 064 | MSE: 377.30273
Epoch: 065 | MSE: 377.30228
Epoch: 066 | MSE: 377.30191
```

```
Epoch: 067 | MSE: 377.30167
Epoch: 068 | MSE: 377.30142
Epoch: 069 | MSE: 377.30124
Epoch: 070 | MSE: 377.30112
Epoch: 071 | MSE: 377.30099
Epoch: 072 | MSE: 377.30090
Epoch: 073 | MSE: 377.30081
Epoch: 074 | MSE: 377.30072
Epoch: 075 | MSE: 377.30072
Epoch: 076 | MSE: 377.30066
Epoch: 077 | MSE: 377.30063
Epoch: 078 | MSE: 377.30063
Epoch: 079 | MSE: 377.30057
Epoch: 080 | MSE: 377.30057
Epoch: 081 | MSE: 377.30048
Epoch: 082 | MSE: 377.30054
Epoch: 083 | MSE: 377.30054
Epoch: 084 | MSE: 377.30054
Epoch: 085 | MSE: 377.30054
Epoch: 086 | MSE: 377.30048
Epoch: 087 | MSE: 377.30048
Epoch: 088 | MSE: 377.30048
Epoch: 089 | MSE: 377.30045
Epoch: 090 | MSE: 377.30048
Epoch: 091 | MSE: 377.30045
Epoch: 092 | MSE: 377.30048
Epoch: 093 | MSE: 377.30045
Epoch: 094 | MSE: 377.30048
Epoch: 095 | MSE: 377.30045
Epoch: 096 | MSE: 377.30045
Epoch: 097 | MSE: 377.30048
Epoch: 098 | MSE: 377.30045
Epoch: 099 | MSE: 377.30048
Epoch: 100 | MSE: 377.30048
```

[196]:

```
# plot MSE
plt.plot(range(len(cost)), cost)
plt.ylabel('Mean Squared Error')
plt.xlabel('Epoch')
plt.show()
```



[197]:
```
train_pred = model.forward(X_train)
test_pred = model.forward(X_test)

print('Train MSE: %.5f' % loss(train_pred, y_train))
print('Test MSE: %.5f' % loss(test_pred, y_test))
```

```
Train MSE: 377.30048
Test MSE: 394.55591
```

Try Modify the train() function such that the dataset is shuffled prior to each epoch. Do you see a difference – Yes/No? Try to come up with an explanation for your observation.

0.4 Adaline With SGD

[275]:
```
df = pd.read_csv('data/iris.data', index_col=None, header=None)
df.columns = ['x1', 'x2', 'x3', 'x4', 'y']
df = df.iloc[50:150]
df['y'] = df['y'].apply(lambda x: 0 if x == 'Iris-versicolor' else 1)
df.tail()
```

[275]:

	x1	x2	x3	x4	y
145	6.7	3.0	5.2	2.3	1
146	6.3	2.5	5.0	1.9	1
147	6.5	3.0	5.2	2.0	1
148	6.2	3.4	5.4	2.3	1

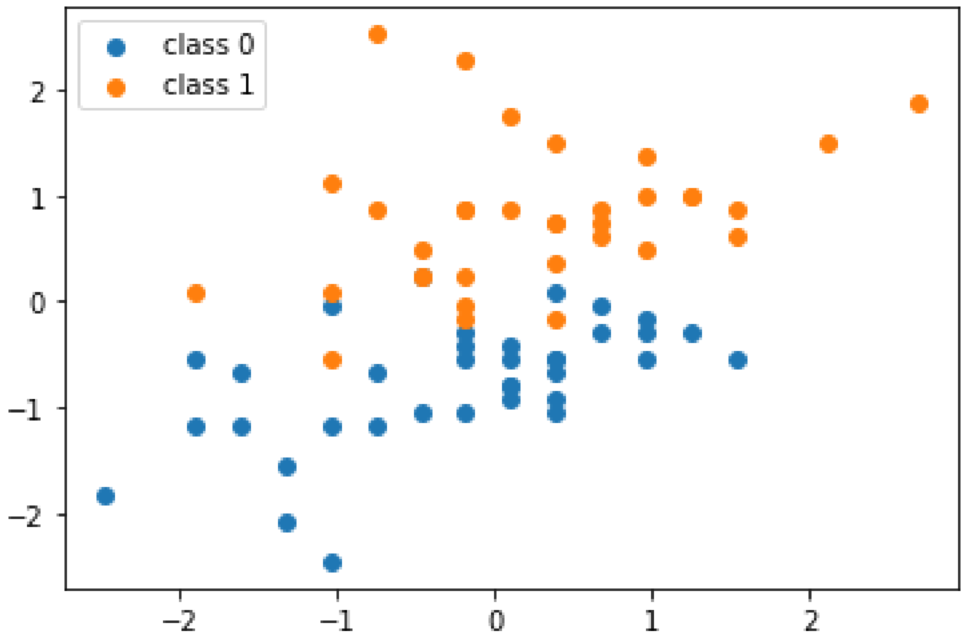
```
149  5.9  3.0  5.1  1.8  1
```

[200]:

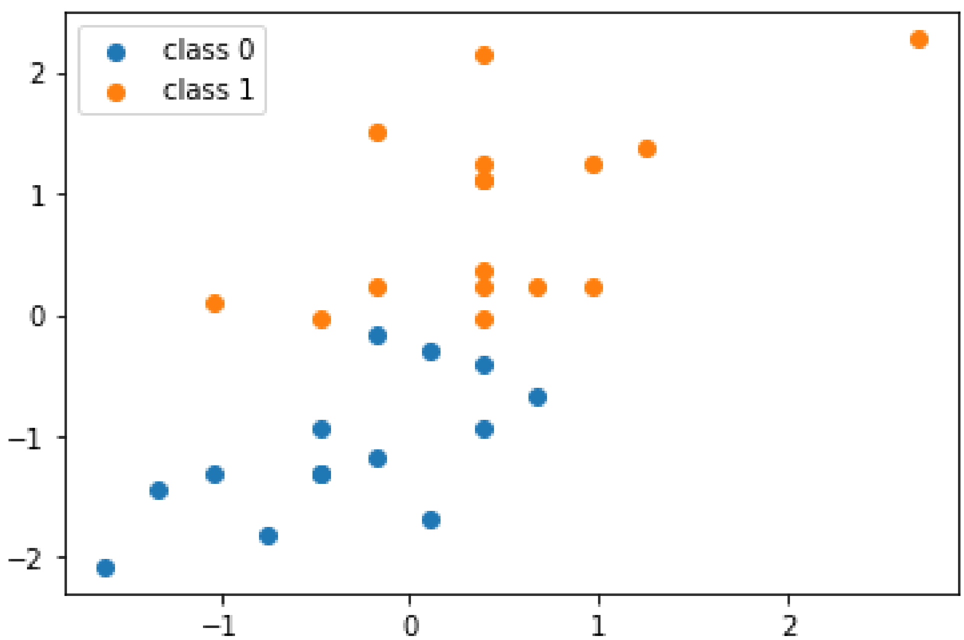
```
# Assign features and target

X = torch.tensor(df[['x2', 'x3']].values, dtype=torch.float)
y = torch.tensor(df['y'].values, dtype=torch.int)

# Shuffling & train/test split

torch.manual_seed(123)
shuffle_idx = torch.randperm(y.size(0), dtype=torch.long)

X, y = X[shuffle_idx], y[shuffle_idx]

percent70 = int(shuffle_idx.size(0)*0.7)

X_train, X_test = X[shuffle_idx[:percent70]], X[shuffle_idx[percent70:]]
y_train, y_test = y[shuffle_idx[:percent70]], y[shuffle_idx[percent70:]]

# Normalize (mean zero, unit variance)

mu, sigma = X_train.mean(dim=0), X_train.std(dim=0)
X_train = (X_train - mu) / sigma
X_test = (X_test - mu) / sigma
```

[202]:

```
plt.scatter(X_train[y_train == 0, 0], X_train[y_train == 0, 1], label='class 0')
plt.scatter(X_train[y_train == 1, 0], X_train[y_train == 1, 1], label='class 1')
plt.legend()
plt.show()
```



[203]:
```
plt.scatter(X_test[y_test == 0, 0], X_test[y_test == 0, 1], label='class 0')
plt.scatter(X_test[y_test == 1, 0], X_test[y_test == 1, 1], label='class 1')
plt.legend()
plt.show()
```



[204]:
```python
# implement ADALINE
class Adaline1():
    def __init__(self, num_features):
        self.num_features = num_features
        self.weights = torch.zeros(num_features, 1, dtype=torch.float)
        self.bias = torch.zeros(1, dtype=torch.float)

    def forward(self, x):
        netinputs = torch.add(torch.mm(x, self.weights), self.bias)
        activations = netinputs
        return activations.view(-1)

    def backward(self, x, yhat, y):

        grad_loss_yhat = 2*(yhat - y)

        grad_yhat_weights = x
        grad_yhat_bias = 1.

        # Chain rule: inner times outer
        grad_loss_weights = torch.mm(grad_yhat_weights.t(),grad_loss_yhat.
 ↪view(-1, 1)) / y.size(0)

        grad_loss_bias = torch.sum(grad_yhat_bias*grad_loss_yhat) / y.size(0)

        # return negative gradient
        return (-1)*grad_loss_weights, (-1)*grad_loss_bias
```

[205]:
```python
def loss(yhat, y):
    return torch.mean((yhat - y)**2)


def train(model, x, y, num_epochs,
          learning_rate=0.01, seed=123, minibatch_size=10):
    cost = []

    torch.manual_seed(seed)
    for e in range(num_epochs):

        #### Shuffle epoch
        shuffle_idx = torch.randperm(y.size(0), dtype=torch.long)
        minibatches = torch.split(shuffle_idx, minibatch_size)

        for minibatch_idx in minibatches:

            #### Compute outputs ####
            yhat = model.forward(x[minibatch_idx])
```

```
            #### Compute gradients ####
            negative_grad_w, negative_grad_b = \
                model.backward(x[minibatch_idx], yhat, y[minibatch_idx])

            #### Update weights ####
            model.weights += learning_rate * negative_grad_w
            model.bias += learning_rate * negative_grad_b

            #### Logging ####
            minibatch_loss = loss(yhat, y[minibatch_idx])
            print('    Minibatch MSE: %.3f' % minibatch_loss)

        #### Logging ####
        yhat = model.forward(x)
        curr_loss = loss(yhat, y)
        print('Epoch: %03d' % (e+1), end="")
        print(' | MSE: %.5f' % curr_loss)
        cost.append(curr_loss)

    return cost
```

[206]:
```
model = Adaline1(num_features=X_train.size(1))
cost = train(model, X_train, y_train.float(),num_epochs=20,learning_rate=0.
 ↪1,seed=123,minibatch_size=10)
```

```
    Minibatch MSE: 0.500
    Minibatch MSE: 0.341
    Minibatch MSE: 0.220
    Minibatch MSE: 0.245
    Minibatch MSE: 0.157
    Minibatch MSE: 0.133
    Minibatch MSE: 0.144
Epoch: 001 | MSE: 0.12142
    Minibatch MSE: 0.107
    Minibatch MSE: 0.147
    Minibatch MSE: 0.064
    Minibatch MSE: 0.079
    Minibatch MSE: 0.185
    Minibatch MSE: 0.063
    Minibatch MSE: 0.135
Epoch: 002 | MSE: 0.09932
    Minibatch MSE: 0.093
    Minibatch MSE: 0.064
    Minibatch MSE: 0.128
    Minibatch MSE: 0.099
    Minibatch MSE: 0.079
```

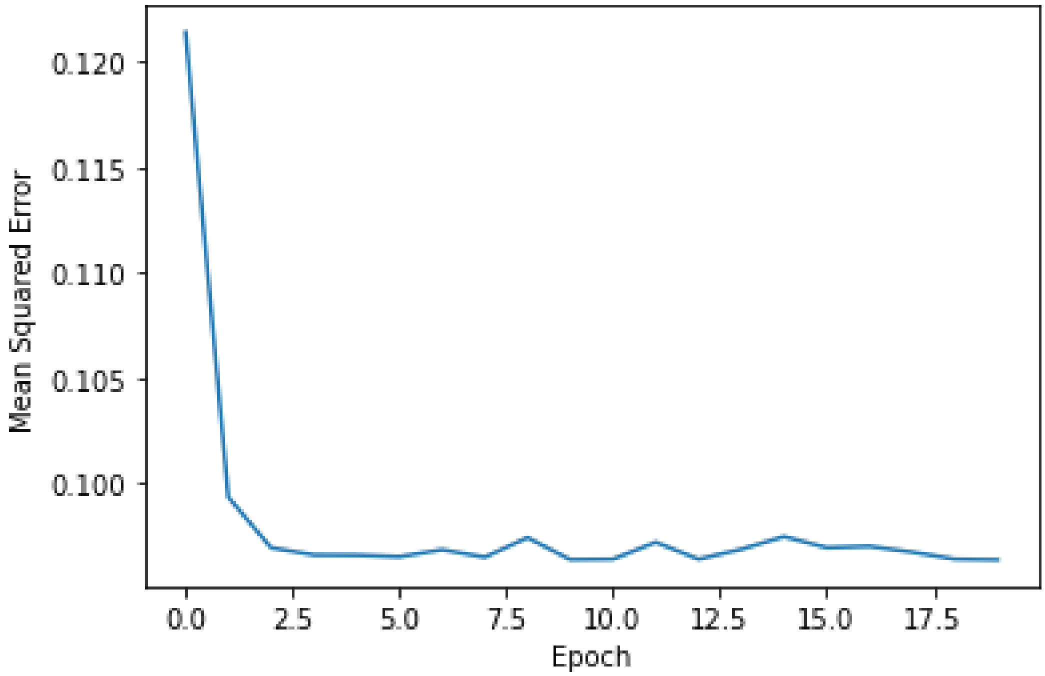
```
    Minibatch MSE: 0.157
    Minibatch MSE: 0.080
Epoch: 003 | MSE: 0.09693
    Minibatch MSE: 0.131
    Minibatch MSE: 0.146
    Minibatch MSE: 0.050
    Minibatch MSE: 0.095
    Minibatch MSE: 0.106
    Minibatch MSE: 0.072
    Minibatch MSE: 0.102
Epoch: 004 | MSE: 0.09658
    Minibatch MSE: 0.107
    Minibatch MSE: 0.204
    Minibatch MSE: 0.149
    Minibatch MSE: 0.054
    Minibatch MSE: 0.060
    Minibatch MSE: 0.056
    Minibatch MSE: 0.069
Epoch: 005 | MSE: 0.09657
    Minibatch MSE: 0.068
    Minibatch MSE: 0.111
    Minibatch MSE: 0.092
    Minibatch MSE: 0.115
    Minibatch MSE: 0.157
    Minibatch MSE: 0.074
    Minibatch MSE: 0.087
Epoch: 006 | MSE: 0.09650
    Minibatch MSE: 0.057
    Minibatch MSE: 0.070
    Minibatch MSE: 0.133
    Minibatch MSE: 0.127
    Minibatch MSE: 0.062
    Minibatch MSE: 0.153
    Minibatch MSE: 0.103
Epoch: 007 | MSE: 0.09683
    Minibatch MSE: 0.102
    Minibatch MSE: 0.110
    Minibatch MSE: 0.101
    Minibatch MSE: 0.065
    Minibatch MSE: 0.126
    Minibatch MSE: 0.124
    Minibatch MSE: 0.076
Epoch: 008 | MSE: 0.09648
    Minibatch MSE: 0.120
    Minibatch MSE: 0.056
    Minibatch MSE: 0.100
    Minibatch MSE: 0.102
    Minibatch MSE: 0.106
```

```
    Minibatch MSE: 0.075
    Minibatch MSE: 0.144
Epoch: 009 | MSE: 0.09740
    Minibatch MSE: 0.073
    Minibatch MSE: 0.071
    Minibatch MSE: 0.084
    Minibatch MSE: 0.152
    Minibatch MSE: 0.099
    Minibatch MSE: 0.108
    Minibatch MSE: 0.118
Epoch: 010 | MSE: 0.09636
    Minibatch MSE: 0.058
    Minibatch MSE: 0.070
    Minibatch MSE: 0.145
    Minibatch MSE: 0.081
    Minibatch MSE: 0.093
    Minibatch MSE: 0.127
    Minibatch MSE: 0.115
Epoch: 011 | MSE: 0.09638
    Minibatch MSE: 0.123
    Minibatch MSE: 0.091
    Minibatch MSE: 0.085
    Minibatch MSE: 0.093
    Minibatch MSE: 0.091
    Minibatch MSE: 0.143
    Minibatch MSE: 0.081
Epoch: 012 | MSE: 0.09718
    Minibatch MSE: 0.096
    Minibatch MSE: 0.076
    Minibatch MSE: 0.149
    Minibatch MSE: 0.092
    Minibatch MSE: 0.116
    Minibatch MSE: 0.093
    Minibatch MSE: 0.091
Epoch: 013 | MSE: 0.09638
    Minibatch MSE: 0.095
    Minibatch MSE: 0.104
    Minibatch MSE: 0.107
    Minibatch MSE: 0.120
    Minibatch MSE: 0.102
    Minibatch MSE: 0.045
    Minibatch MSE: 0.124
Epoch: 014 | MSE: 0.09685
    Minibatch MSE: 0.121
    Minibatch MSE: 0.051
    Minibatch MSE: 0.095
    Minibatch MSE: 0.122
    Minibatch MSE: 0.030
```

```
    Minibatch MSE: 0.158
    Minibatch MSE: 0.121
Epoch: 015 | MSE: 0.09745
    Minibatch MSE: 0.080
    Minibatch MSE: 0.119
    Minibatch MSE: 0.091
    Minibatch MSE: 0.095
    Minibatch MSE: 0.044
    Minibatch MSE: 0.092
    Minibatch MSE: 0.180
Epoch: 016 | MSE: 0.09693
    Minibatch MSE: 0.054
    Minibatch MSE: 0.075
    Minibatch MSE: 0.184
    Minibatch MSE: 0.105
    Minibatch MSE: 0.121
    Minibatch MSE: 0.066
    Minibatch MSE: 0.096
Epoch: 017 | MSE: 0.09699
    Minibatch MSE: 0.076
    Minibatch MSE: 0.050
    Minibatch MSE: 0.198
    Minibatch MSE: 0.105
    Minibatch MSE: 0.054
    Minibatch MSE: 0.136
    Minibatch MSE: 0.099
Epoch: 018 | MSE: 0.09672
    Minibatch MSE: 0.158
    Minibatch MSE: 0.099
    Minibatch MSE: 0.087
    Minibatch MSE: 0.070
    Minibatch MSE: 0.103
    Minibatch MSE: 0.112
    Minibatch MSE: 0.081
Epoch: 019 | MSE: 0.09638
    Minibatch MSE: 0.095
    Minibatch MSE: 0.093
    Minibatch MSE: 0.111
    Minibatch MSE: 0.147
    Minibatch MSE: 0.083
    Minibatch MSE: 0.102
    Minibatch MSE: 0.065
Epoch: 020 | MSE: 0.09635
```

[208]:
```
plt.plot(range(len(cost)), cost)
plt.ylabel('Mean Squared Error')
plt.xlabel('Epoch')
```

```
plt.show()
```



[209]:

```
ones = torch.ones(y_train.size())
zeros = torch.zeros(y_train.size())
train_pred = model.forward(X_train)
train_acc = torch.mean(
    (torch.where(train_pred > 0.5,
                 ones,
                 zeros).int() == y_train).float())

ones = torch.ones(y_test.size())
zeros = torch.zeros(y_test.size())
test_pred = model.forward(X_test)
test_acc = torch.mean(
    (torch.where(test_pred > 0.5,
                 ones,
                 zeros).int() == y_test).float())

print('Training Accuracy: %.2f' % (train_acc*100))
print('Test Accuracy: %.2f' % (test_acc*100))
```

```
Training Accuracy: 90.00
Test Accuracy: 96.67
```

0.5 Automatic differentiation with PyTorch

[210]:
```python
import torch
from torch.autograd import grad
import torch.nn.functional as F
```

[212]:
```python
x = torch.tensor([3.])
w = torch.tensor([2.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)
a = F.relu(x*w + b)
```

[213]:
```python
a
```

[213]:
```
tensor([7.], grad_fn=<ReluBackward0>)
```

[214]:
```python
grad(a, w, retain_graph=True)
```

[214]:
```
(tensor([3.]),)
```

[215]:
```python
grad(a, b)
```

[215]:
```
(tensor([1.]),)
```

[216]:
```python
grad(a, b)
```

```
---------------------------------------------------------------------------
RuntimeError                              Traceback (most recent call last)
Input In [216], in <cell line: 1>()
----> 1 grad(a, b)

File /opt/anaconda3/lib/python3.9/site-packages/torch/autograd/__init__.py:300,␣
 ↪in grad(outputs, inputs, grad_outputs, retain_graph, create_graph,␣
 ↪only_inputs, allow_unused, is_grads_batched)
    298     return _vmap_internals._vmap(vjp, 0, 0,␣
 ↪allow_none_pass_through=True)(grad_outputs_)
    299 else:
--> 300     return␣
 ↪Variable._execution_engine.run_backward(  # Calls into the C++ engine to run the backward p
    301         t_outputs, grad_outputs_, retain_graph, create_graph, t_inputs,
    302         allow_unused, accumulate_grad=False)

RuntimeError: Trying to backward through the graph a second time (or directly␣
 ↪access saved tensors after they have already been freed). Saved intermediate␣
 ↪values of the graph are freed when you call .backward() or autograd.grad().␣
 ↪Specify retain_graph=True if you need to backward through the graph a second␣
 ↪time or if you need to access saved tensors after calling backward.
```

[217]:
```
x = torch.tensor([3.])
w = torch.tensor([2.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)

def my_relu(z):
    if z > 0.:
        return z
    else:
        z[:] = 0.
        return z

a = my_relu(x*w + b)
grad(a, w)
```

[217]:
```
(tensor([3.]),)
```

0.5.1 What happens to ReLU function at 0 ?

[218]:
```
x = torch.tensor([-1.])
w = torch.tensor([1.], requires_grad=True)
b = torch.tensor([1.], requires_grad=True)

def my_relu(z):
    if z > 0.:
        return z
    else:
        z[:] = 0.
        return z

a = F.relu(x*w + b)
grad(a, w, retain_graph=False)
```

[218]:
```
(tensor([-0.]),)
```

0.6 Adaline With Pytorch

[247]:
```
df = pd.read_csv('data/iris.data', index_col=None, header=None)
df.columns = ['x1', 'x2', 'x3', 'x4', 'y']
df = df.iloc[50:150]
df['y'] = df['y'].apply(lambda x: 0 if x == 'Iris-versicolor' else 1)



# Assign features and target

X = torch.tensor(df[['x2', 'x3']].values, dtype=torch.float)
```

```
y = torch.tensor(df['y'].values, dtype=torch.int)

# Shuffling & train/test split

torch.manual_seed(123)
shuffle_idx = torch.randperm(y.size(0), dtype=torch.long)

X, y = X[shuffle_idx], y[shuffle_idx]

percent70 = int(shuffle_idx.size(0)*0.7)

X_train, X_test = X[shuffle_idx[:percent70]], X[shuffle_idx[percent70:]]
y_train, y_test = y[shuffle_idx[:percent70]], y[shuffle_idx[percent70:]]

# Normalize (mean zero, unit variance)

mu, sigma = X_train.mean(dim=0), X_train.std(dim=0)
X_train = (X_train - mu) / sigma
X_test = (X_test - mu) / sigma
```

[221]:
```
class Adaline2():
    def __init__(self, num_features):
        self.num_features = num_features
        self.weight = torch.zeros(num_features, 1, dtype=torch.
 ↪float,requires_grad=True)
        self.bias = torch.zeros(1, dtype=torch.float, requires_grad=True)

    def forward(self, x):
        netinputs = torch.add(torch.mm(x, self.weight), self.bias)
        activations = netinputs
        return activations.view(-1)


def loss_func(yhat, y):
    return torch.mean((yhat - y)**2)


def train(model, x, y, num_epochs,learning_rate=0.01, seed=123,␣
 ↪minibatch_size=10):
    cost = []

    torch.manual_seed(seed)
    for e in range(num_epochs):

        #### Shuffle epoch
        shuffle_idx = torch.randperm(y.size(0), dtype=torch.long)
        minibatches = torch.split(shuffle_idx, minibatch_size)
```

```python
        for minibatch_idx in minibatches:

            #### Compute outputs ####
            yhat = model.forward(x[minibatch_idx])
            loss = loss_func(yhat, y[minibatch_idx])

            #### Compute gradients ####
            negative_grad_w = grad(loss, model.weight, retain_graph=True)[0] *␣
↪(-1)
            negative_grad_b = grad(loss, model.bias)[0] * (-1)


            #### Update weights ####
            model.weight = model.weight + learning_rate * negative_grad_w
            model.bias = model.bias + learning_rate * negative_grad_b

        #### Logging ####
        with torch.no_grad():
            # context manager to
            # avoid building graph during "inference"
            # to save memory
            yhat = model.forward(x)
            curr_loss = loss_func(yhat, y)
            print('Epoch: %03d' % (e+1), end="")
            print(' | MSE: %.5f' % curr_loss)
            cost.append(curr_loss)

    return cost
```

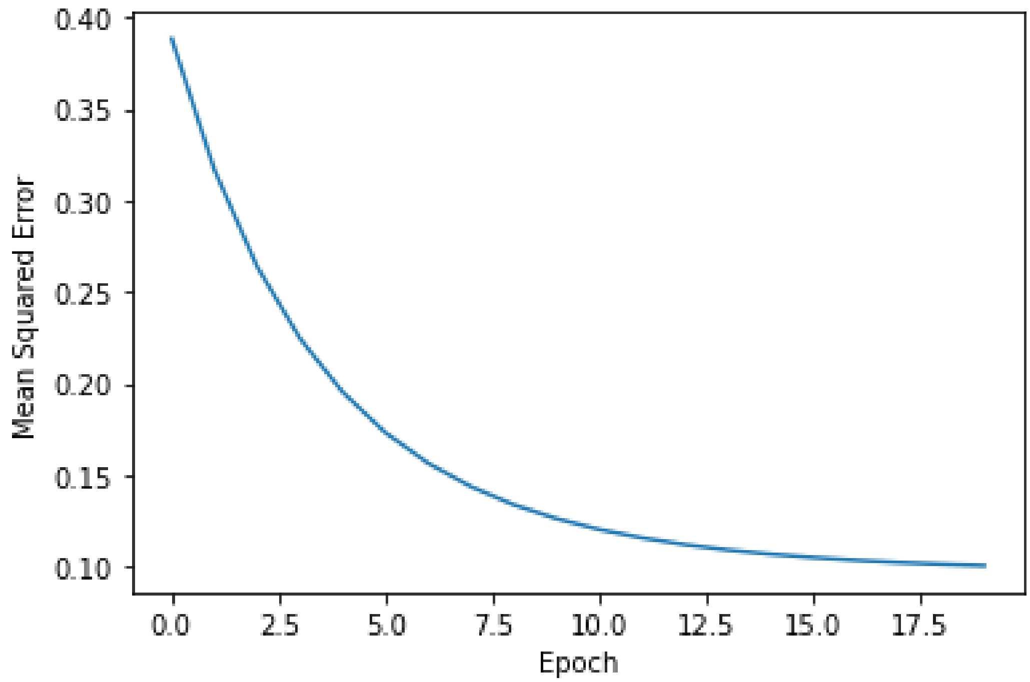
[234]:
```python
model = Adaline2(num_features=X_train.size(1))
cost = train(model,
             X_train, y_train.float(),
             num_epochs=20,
             learning_rate=0.01,
             seed=123,
             minibatch_size=10)
```

```
Epoch: 001 | MSE: 0.38849
Epoch: 002 | MSE: 0.31679
Epoch: 003 | MSE: 0.26379
Epoch: 004 | MSE: 0.22463
Epoch: 005 | MSE: 0.19527
Epoch: 006 | MSE: 0.17307
Epoch: 007 | MSE: 0.15629
Epoch: 008 | MSE: 0.14352
Epoch: 009 | MSE: 0.13360
```

```
Epoch: 010 | MSE: 0.12600
Epoch: 011 | MSE: 0.12007
Epoch: 012 | MSE: 0.11547
Epoch: 013 | MSE: 0.11178
Epoch: 014 | MSE: 0.10884
Epoch: 015 | MSE: 0.10656
Epoch: 016 | MSE: 0.10470
Epoch: 017 | MSE: 0.10320
Epoch: 018 | MSE: 0.10200
Epoch: 019 | MSE: 0.10105
Epoch: 020 | MSE: 0.10025
```

[235]:

```
plt.plot(range(len(cost)), cost)
plt.ylabel('Mean Squared Error')
plt.xlabel('Epoch')
plt.show()
```



[236]:

```
ones = torch.ones(y_train.size())
zeros = torch.zeros(y_train.size())
train_pred = model.forward(X_train)
train_acc = torch.mean(
    (torch.where(train_pred > 0.5,
                 ones,
                 zeros).int() == y_train).float())
```

```
ones = torch.ones(y_test.size())
zeros = torch.zeros(y_test.size())
test_pred = model.forward(X_test)
test_acc = torch.mean(
    (torch.where(test_pred > 0.5,
                 ones,
                 zeros).int() == y_test).float())

print('Training Accuracy: %.2f' % (train_acc*100))
print('Test Accuracy: %.2f' % (test_acc*100))
```

```
Training Accuracy: 92.86
Test Accuracy: 93.33
```

0.7 Adaline With Pytorch With Automatic Diff

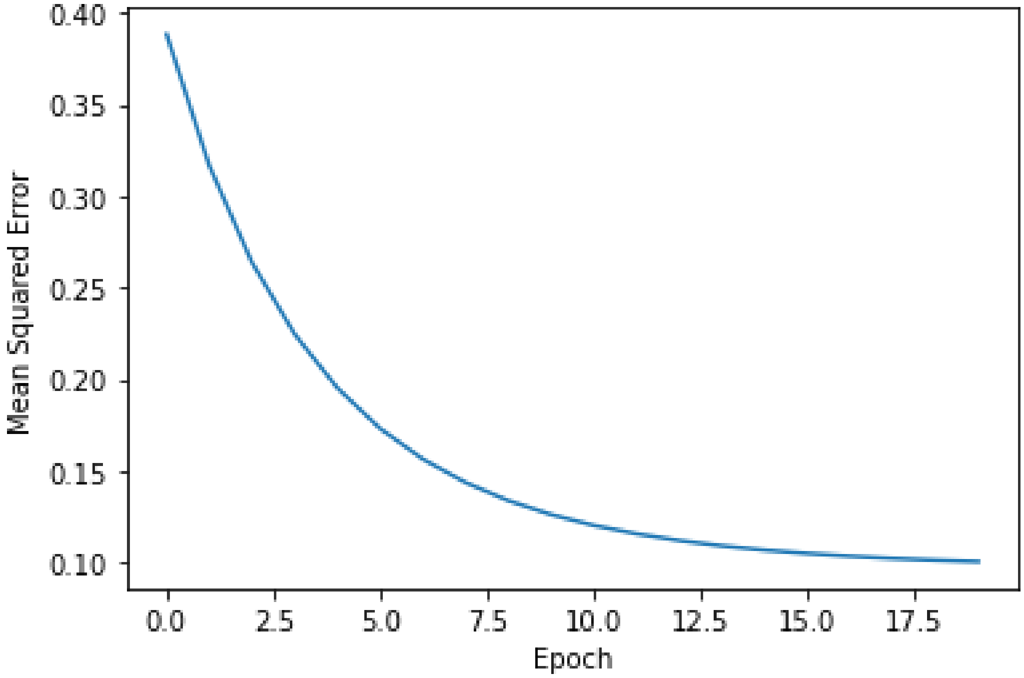
[260]:
```
class Adaline3(torch.nn.Module):
    def __init__(self, num_features):
        super(Adaline3, self).__init__()
        self.linear = torch.nn.Linear(num_features, 1)

        self.linear.weight.detach().zero_()
        self.linear.bias.detach().zero_()


    def forward(self,x):
        netinputs = self.linear(x)
        activations = netinputs
        return activations.view(-1)

def train(model, x, y, num_epochs, learning_rate = 0.01, seed = 123,␣
 ↪minibatch_size=10):
    cost = []
    torch.manual_seed(seed)
    optimiser = torch.optim.SGD(model.parameters(), lr=learning_rate)

    for e in range(num_epochs):
        shuffle_idx = torch.randperm(y.size(0),dtype=torch.long)
        minibatches = torch.split(shuffle_idx,minibatch_size)

        for minibatch_idx in minibatches:

            yhat = model.forward(x[minibatch_idx])
            loss = F.mse_loss(yhat,y[minibatch_idx])

            optimiser.zero_grad()
```

```
            loss.backward()

            optimiser.step()

        with torch.no_grad():
            yhat = model.forward(x)
            curr_loss = F.mse_loss(yhat, y)

            print('Epoch: %03d' % (e+1), end="")
            print(' | MSE: %.5f' % curr_loss)
            cost.append(curr_loss)

    return cost;
```

[261]:
```
model = Adaline3(num_features=X_train.size(1))
cost = train(model, X_train, y_train.float(),num_epochs=20,learning_rate=0.
 ↪01,seed=123,minibatch_size=10)
```

```
Epoch: 001 | MSE: 0.38849
Epoch: 002 | MSE: 0.31679
Epoch: 003 | MSE: 0.26379
Epoch: 004 | MSE: 0.22463
Epoch: 005 | MSE: 0.19527
Epoch: 006 | MSE: 0.17307
Epoch: 007 | MSE: 0.15629
Epoch: 008 | MSE: 0.14352
Epoch: 009 | MSE: 0.13360
Epoch: 010 | MSE: 0.12600
Epoch: 011 | MSE: 0.12007
Epoch: 012 | MSE: 0.11547
Epoch: 013 | MSE: 0.11178
Epoch: 014 | MSE: 0.10884
Epoch: 015 | MSE: 0.10656
Epoch: 016 | MSE: 0.10470
Epoch: 017 | MSE: 0.10320
Epoch: 018 | MSE: 0.10200
Epoch: 019 | MSE: 0.10105
Epoch: 020 | MSE: 0.10025
```

[262]:
```
plt.plot(range(len(cost)), cost)
plt.ylabel('Mean Squared Error')
plt.xlabel('Epoch')
plt.show()
```


0.40
0.35
0.30
0.25
0.20
0.15
0.10
Mean Squared Error
0.0
2.5
5.0
7.5
10.0
12.5
15.0
17.5
Epoch

```
[ ]:
```