# Neuron Layers

SC4001 – Tutorial 3

1. Design a softmax layer of neurons to perform the following classification, given the inputs $\boldsymbol{x} = (x_1, x_2)$ and target class labels $d$:

| $(x_1, x_2)$ | (0 4) | (-1 3) | (2 3) | (-2 2) | (0 2) | (1 2) | (-1 2) | (-3 1) | (-1 1) |
|---|---|---|---|---|---|---|---|---|---|
| $d$ | A | A | A | B | B | A | B | B | B |

| $(x_1, x_2)$ | (2 1) | (4 1) | (-2 0) | (1 0) | (3 0) | (-3 -1) | (-2 -1) | (2 -1) | (4 -1) |
|---|---|---|---|---|---|---|---|---|---|
| $d$ | C | C | B | C | C | B | B | C | C |

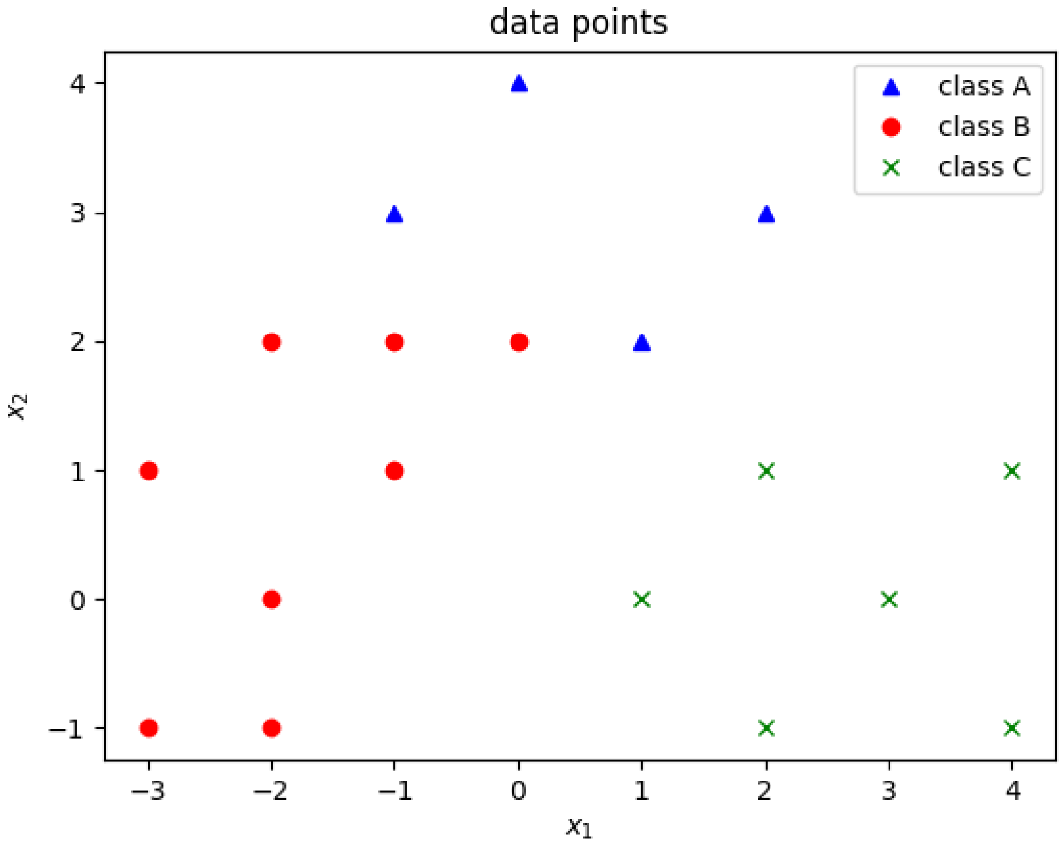
(a) Show one iteration of gradient descent learning at a learning factor 0.05.

Initialize the weights to $\begin{pmatrix} 0.88 & 0.08 & -0.34 \\ 0.68 & -0.39 & -0.19 \end{pmatrix}$ and biases to zero

(b) Find the weights and biases at convergence of learning

(c) Indicate the probabilities that the network predicts the classes of trained patterns.

(d) Plot the decision boundaries separating the three classes.


data points
class A
class B
class C
$x_1$
$x_2$
−3
−2
−1
0
1
2
3
4
−1
0
1
2
3
4

# GD for Softmax layer

Given training set $(\boldsymbol{X}, \boldsymbol{d})$

Set learning rate $\alpha$

Initialize $\boldsymbol{W}$ and $\boldsymbol{b}$

Iterate until convergence:

$$\boldsymbol{U} = \boldsymbol{X}\boldsymbol{W} + \boldsymbol{B}$$

$$f(\boldsymbol{U}) = \frac{e^{\boldsymbol{U}}}{\sum_{k=1}^{K} e^{\boldsymbol{U}_{\boldsymbol{k}}}}$$

$$\nabla_U J = -\left(\boldsymbol{K} - f(\boldsymbol{U})\right)$$

$$\boldsymbol{W} \leftarrow \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_U J$$

$$\boldsymbol{b} \leftarrow \boldsymbol{b} - \alpha (\nabla_U J)^T \mathbf{1}_P$$

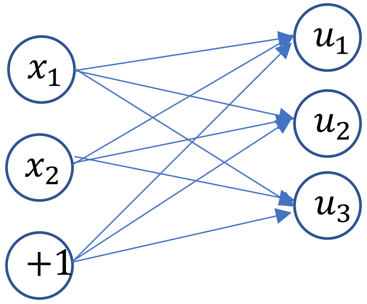
Labels for classes:

$$Class\ A \rightarrow 1, Class\ B \rightarrow 2, Class\ C \rightarrow 3$$

18 training patterns!

The data matrix and target vector:

$$\boldsymbol{X} = \begin{pmatrix} 0 & 4 \\ -1 & 3 \\ 2 & 3 \\ -2 & 2 \\ 0 & 2 \\ 1 & 2 \\ -1 & 2 \\ \vdots & \vdots \\ 4 & -1 \end{pmatrix}, \quad \boldsymbol{d} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 1 \\ 2 \\ \vdots \\ 3 \end{pmatrix}, \boldsymbol{K} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 1 \end{pmatrix}$$




$$f(\boldsymbol{U}) = \frac{e^{\boldsymbol{U}}}{\sum_{k=1}^{K} e^{\boldsymbol{U_k}}} = P(y = k|x)$$

$$\boldsymbol{Y} = \underset{k}{\operatorname{argmax}} f(\boldsymbol{U})$$

Learning rate $\alpha = 0.05$.

Initialize weights and biases:

$$\boldsymbol{W} = \begin{pmatrix} 0.88 & 0.08 & -0.34 \\ 0.68 & -0.39 & -0.19 \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix}$$

**Epoch 1:**

$$\boldsymbol{U} = \boldsymbol{XW} + \boldsymbol{B} = \begin{pmatrix} 0 & 4 \\ -1 & 3 \\ 2 & 3 \\ -2 & 2 \\ \vdots & \vdots \\ 4 & -1 \end{pmatrix} \begin{pmatrix} 0.88 & 0.08 & -0.34 \\ 0.68 & -0.39 & -0.19 \end{pmatrix} + \begin{pmatrix} 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 \\ \vdots & \vdots & \vdots \\ 0.0 & 0.0 & 0.0 \end{pmatrix} = \begin{pmatrix} 2.72 & -1.54 & -0.75 \\ 1.17 & -1.23 & -0.23 \\ 3.8 & -1.0 & -1.23 \\ -0.39 & -0.93 & 0.30 \\ \vdots & \vdots & \vdots \\ 2.82 & 0.71 & -1.16 \end{pmatrix}$$

$$\boldsymbol{U} = \begin{pmatrix} 2.72 & -1.54 & -0.75 \\ 1.17 & -1.23 & -0.23 \\ 3.8 & -1.0 & -1.23 \\ -0.39 & -0.93 & 0.30 \\ \vdots & \vdots & \vdots \\ 2.82 & 0.71 & -1.16 \end{pmatrix}$$

$$f(u_k) = \frac{e^{u_k}}{\sum_{k'=1}^{K} e^{u_{k'}}}$$

$$f(\boldsymbol{U}) == \begin{pmatrix} \frac{e^{2.72}}{e^{2.72}+e^{-1.54}+e^{-0.75}} & \frac{e^{-1.54}}{e^{2.72}+e^{-1.54}+e^{-0.75}} & \frac{e^{-0.75}}{e^{2.72}+e^{-1.54}+e^{-0.75}} \\ \frac{e^{1.17}}{e^{1.17}+e^{-1.23}+e^{-0.23}} & \frac{e^{-1.23}}{e^{1.17}+e^{-1.23}+e^{-0.23}} & \frac{e^{-0.23}}{e^{1.17}+e^{-1.23}+e^{-0.23}} \\ \vdots & \vdots & \vdots \\ \frac{e^{2.82}}{e^{2.82}+e^{0.71}+e^{-1.16}} & \frac{e^{0.71}}{e^{2.82}+e^{0.71}+e^{-1.16}} & \frac{e^{-1.16}}{e^{2.82}+e^{0.71}+e^{-1.16}} \end{pmatrix} = \begin{pmatrix} 0.96 & 0.01 & 0.03 \\ 0.75 & 0.07 & 0.18 \\ \vdots & \vdots & \vdots \\ 0.88 & 0.11 & 0.02 \end{pmatrix}$$

$$\boldsymbol{d} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 2 \\ \vdots \\ 3 \end{pmatrix}$$

$$\boldsymbol{y} = \underset{k}{\operatorname{argmax}} f(\boldsymbol{U}) = \underset{k}{\operatorname{argmax}} \begin{pmatrix} 0.96 & 0.01 & 0.03 \\ 0.75 & 0.07 & 0.18 \\ 0.99 & 0.01 & 0.01 \\ 0.28 & 0.16 & 0.56 \\ \vdots & \vdots & \vdots \\ 0.88 & 0.11 & 0.02 \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 3 \\ \vdots \\ 1 \end{pmatrix}$$

$$\mathit{Classification\ error} = \sum_{p=1}^{18} 1(\boldsymbol{y} \neq \boldsymbol{d}) = 14$$

$$\mathit{entropy} = -\sum_{p=1}^{18} \mathit{log}\left(f\left(u_{pd_p}\right)\right)$$

$$= -\log(0.96) - \log(0.75) - \log(0.99) - \log(0.16) \cdots - \log(0.02)$$

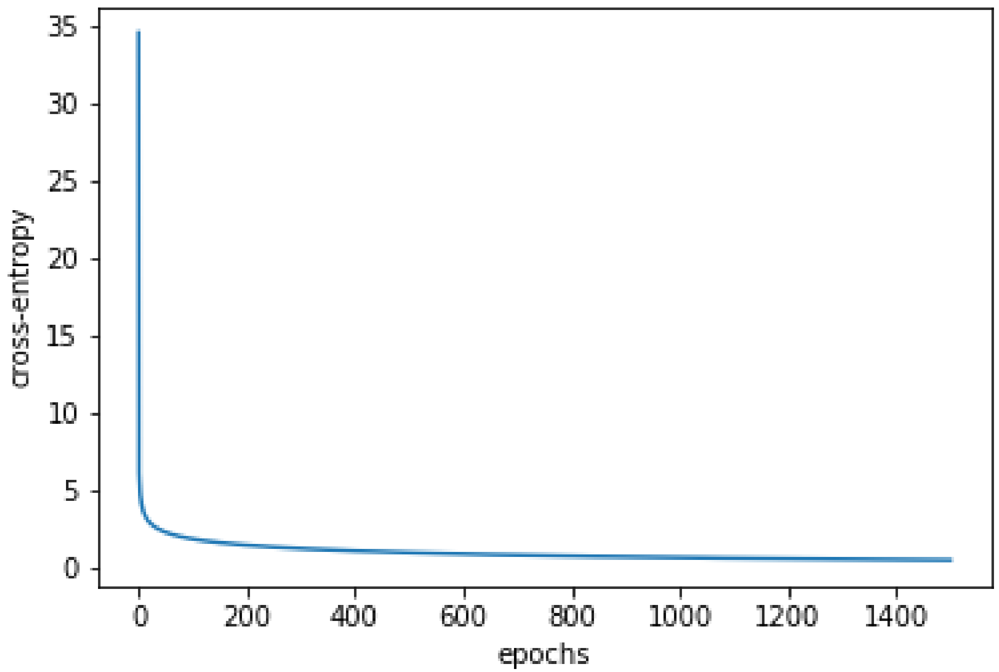
$$= 34.36$$

$$\nabla_U J = -\big(\boldsymbol{K} - f(\boldsymbol{U})\big) = -\left(\begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ \vdots & \vdots & \vdots \\ 0 & 0 & 1 \end{pmatrix} - \begin{pmatrix} 0.96 & 0.01 & 0.03 \\ 0.75 & 0.07 & 0.18 \\ \vdots & \vdots & \vdots \\ 0.88 & 0.11 & 0.02 \end{pmatrix}\right) = \begin{pmatrix} -0.04 & 0.01 & 0.03 \\ -0.25 & 0.07 & 0.18 \\ \vdots & \vdots & \vdots \\ 0.88 & 0.11 & -0.98 \end{pmatrix}$$

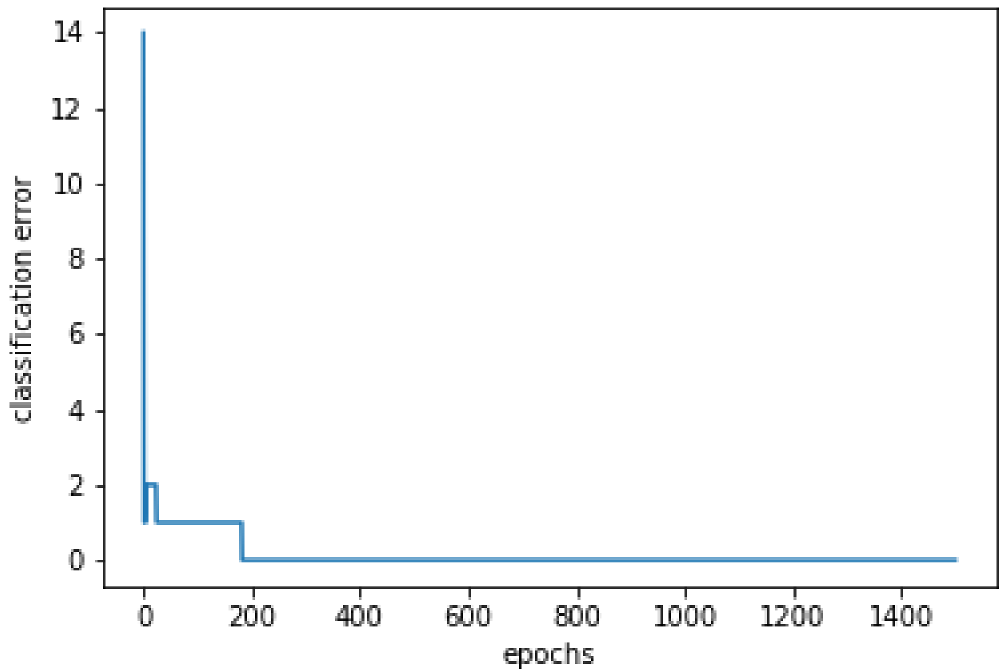
$$\boldsymbol{W} = \boldsymbol{W} - \alpha \boldsymbol{X}^T \nabla_U J = \begin{pmatrix} 0.88 & 0.08 & -0.34 \\ 0.68 & -0.39 & -0.19 \end{pmatrix} - 0.05 \begin{pmatrix} 0 & -1 & \cdots & 4 \\ 4 & 3 & \cdots & -1 \end{pmatrix} \begin{pmatrix} -0.04 & 0.01 & 0.03 \\ -0.25 & 0.07 & 0.18 \\ \vdots & \vdots & \vdots \\ 0.88 & 0.11 & -0.98 \end{pmatrix}$$

$$= \begin{pmatrix} 0.28 & -0.54 & 0.89 \\ 0.54 & -0.12 & -0.31 \end{pmatrix}$$

$$\boldsymbol{b} = \boldsymbol{b} - \alpha (\nabla_U J)^T \mathbf{1}_P = \begin{pmatrix} 0.0 \\ 0.0 \\ 0.0 \end{pmatrix} + 0.05 \begin{pmatrix} -0.04 & 0.01 & 0.03 \\ -0.25 & 0.07 & 0.18 \\ \vdots & \vdots & \vdots \\ 0.88 & 0.11 & -0.98 \end{pmatrix}^T \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix} = \begin{pmatrix} -0.32 \\ 0.27 \\ 0.06 \end{pmatrix}$$



35
30
25
20
15
10
5
0
cross-entropy
0
200
400
600
800
1000
1200
1400
epochs



14
12
10
8
6
4
2
0
classification error
0
200
400
600
800
1000
1200
1400
epochs

**At convergence:**

$$\boldsymbol{W} = \begin{pmatrix} -0.15 & -3.41 & 4.18 \\ 5.27 & -1.02 & -4.15 \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} -7.82 \\ 5.81 \\ 2.02 \end{pmatrix}$$

Entropy = 0.58

Classification Error = 0

**At convergence:**

$$\boldsymbol{X} = \begin{pmatrix} -1 & 2 \\ 0 & 4 \\ -1 & 3 \\ 0 & 2 \\ 3 & 0 \\ -2 & -1 \\ 4 & 1 \\ 1 & 2 \\ 2 & -1 \\ 2 & 3 \\ 2 & 1 \\ -2 & 0 \\ -3 & -1 \\ 1 & 0 \\ -1 & 1 \\ 4 & -1 \\ -3 & 1 \\ -2 & 2 \end{pmatrix} \quad f(\boldsymbol{U}) = \begin{pmatrix} 1.0 & 0.0 & 0.0 \\ 0.88 & 0.12 & 0.0 \\ 1.0 & 0.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.26 & 0.74 & 0.0 \\ 0.89 & 0.1 & 0.0 \\ 0.01 & 0.99 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.02 & 0.98 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 1.0 & 0.0 \\ 0.0 & 0.0 & 1.0 \\ 0.0 & 0.0 & 1.0 \end{pmatrix}, \boldsymbol{Y} = \begin{pmatrix} 1 \\ 1 \\ 1 \\ 2 \\ 2 \\ 1 \\ 2 \\ 2 \\ 2 \\ 3 \\ 3 \\ 2 \\ 3 \\ 3 \\ 2 \\ 2 \\ 3 \\ 3 \end{pmatrix}$$

Probabilities that input patterns belong to target classes are given in RED.

**At convergence:**

$$\boldsymbol{W} = \begin{pmatrix} -0.15 & -3.41 & 4.18 \\ 5.27 & -1.02 & -4.15 \end{pmatrix} \text{ and } \boldsymbol{b} = \begin{pmatrix} -7.82 \\ 5.81 \\ 2.02 \end{pmatrix}$$

Synaptic inputs at the softmax layer for an input $\boldsymbol{x} = (x_1, x_2)$:

Neuron of class A, $u_1 = \boldsymbol{w}_1{}^T\boldsymbol{x} + b_1 = -0.15x_1 + 5.27x_2 - 7.82$
Neuron of class B, $u_2 = \boldsymbol{w}_2{}^T\boldsymbol{x} + b_2 = -3.41x_1 - 1.02x_2 + 5.81$
Neuron of class C, $u_3 = \boldsymbol{w}_3{}^T\boldsymbol{x} + b_3 = 4.18x_1 - 4.15x_2 + 2.02$

Decision boundaries:

Between class A and class B is given when $u_1 = u_2$

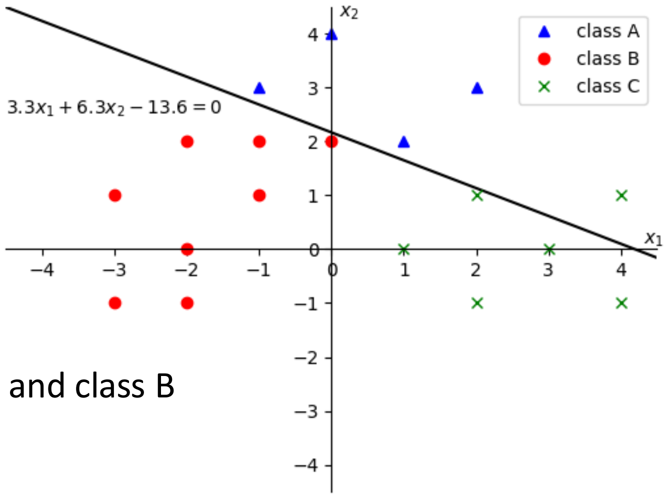
$$-0.15x_1 + 5.27x_2 - 7.82 = -3.41x_1 - 1.02x_2 + 5.81$$
$$3.25x_1 + 6.29x_2 - 13.63 = 0$$

Similarly, between class B and class C $u_2 = u_3$ :

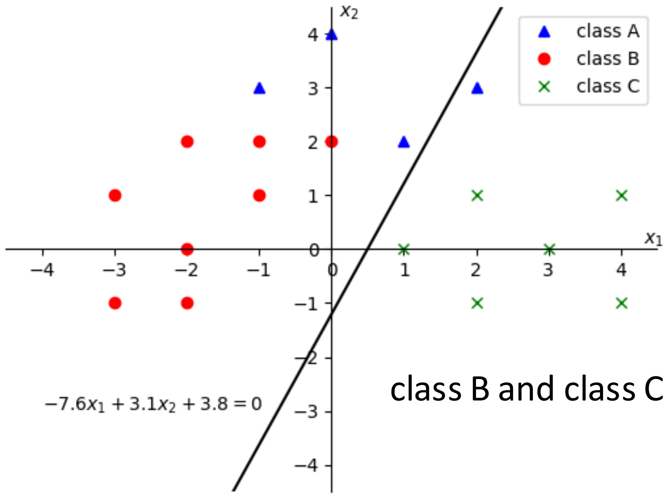
$$-7.59x_1 + 3.13x_2 + 3.79 = 0$$

between class A and class C $u_3 = u_1$ :

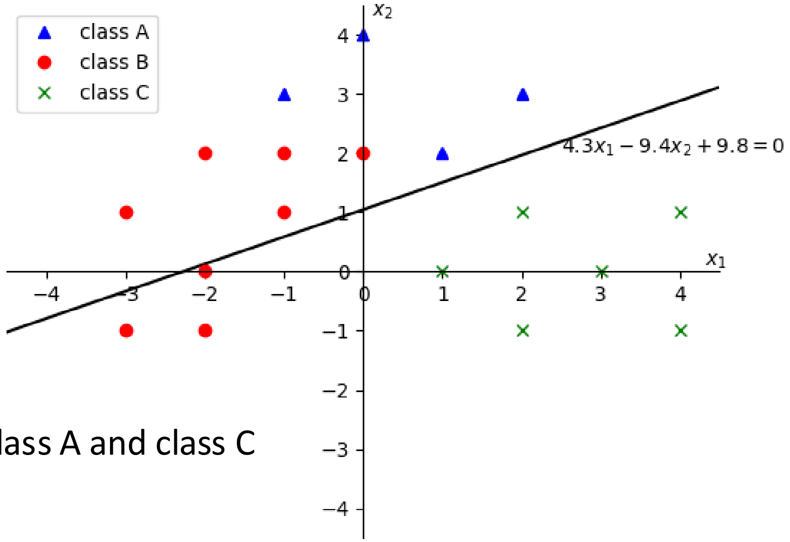
$$4.33x_1 - 9.42x_2 + 9.84 = 0$$



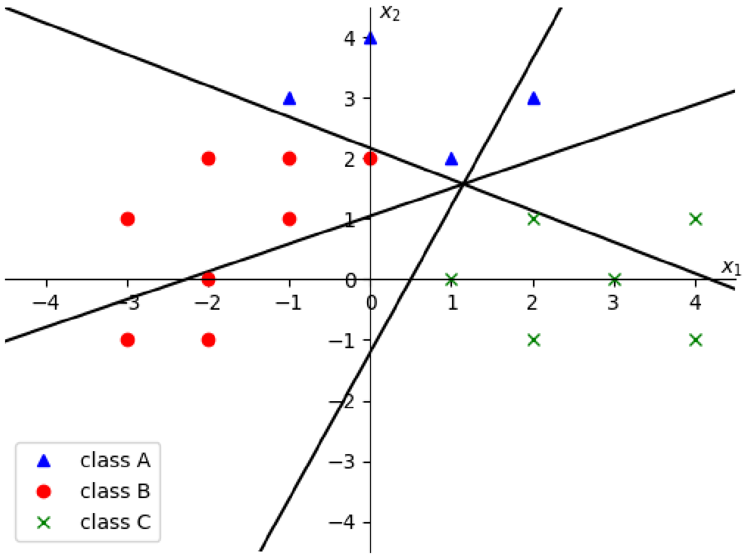

class A and class B



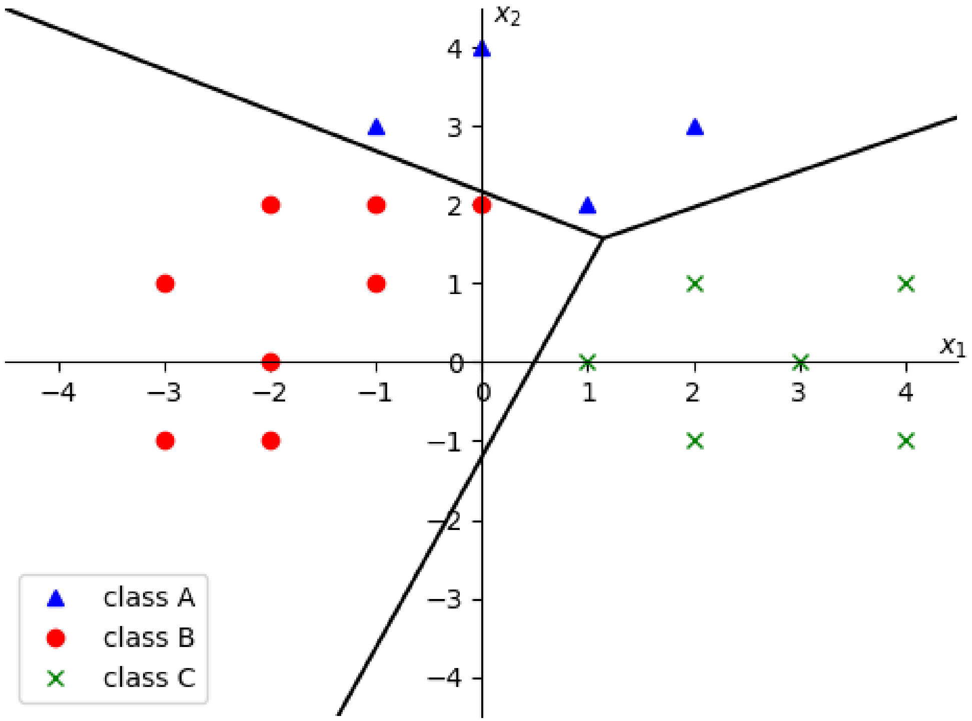

class B and class C




class A and class C



Decision boundaries leant by the softmax layer



2. Read the Linnerud dataset from sklearn.datasets package by using

**from** sklearn.datasets **import** load_linnerud

The Linnerud dataset is a multi-output regression dataset consisting of three exercise (data) and three physiological variables (targets) collected from twenty middle-aged men in a fitness club:

physiological - Weight, Waist and Pulse
exercise - Chins, Situps and Jumps.

Divide the dataset into train and test partitions at 0.75:0.25 ratio and train a perceptron layer to predict physiological data from exercise variables by implementing with

a) direct gradients
b) 'autograd' functions available in pytorch

Remember to Gaussian normalize inputs and scale the output data. You can use preprocessing API in sklearn:

**from** sklearn **import** preprocessing

Draw learning curves and find mean square error and $R^2$ values of prediction. Which physiological variable can be better predicted by exercise data?

Compare the time-taken for a weight update and the number of epochs required for convergence for (a) and (b).

```python
from sklearn.datasets import load_linnerud
from sklearn.model_selection import train_test_split
```

```python
X, y = load_linnerud(return_X_y=True) # 20 data points
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, random_state=1)
```

**X: input features**

| | chins | sit_ups | jumps |
|---|---|---|---|
| 0 | 5.0 | 162.0 | 60.0 |
| 1 | 2.0 | 110.0 | 60.0 |
| 2 | 12.0 | 101.0 | 101.0 |
| 3 | 12.0 | 105.0 | 37.0 |
| 4 | 13.0 | 155.0 | 58.0 |

**Y: output labels**

| | weight | waist | pulse |
|---|---|---|---|
| 0 | 191.0 | 36.0 | 50.0 |
| 1 | 189.0 | 37.0 | 52.0 |
| 2 | 193.0 | 38.0 | 58.0 |
| 3 | 162.0 | 35.0 | 62.0 |
| 4 | 189.0 | 35.0 | 46.0 |

Normalizing input variables such that $x' \sim N(0, 1)$

```
# preprocess input and output data
from sklearn import preprocessing

# standard Gaussian scaling for inputs
X_scaler = preprocessing.StandardScaler().fit(X_train)
X_scaled = X_scaler.transform(X_train)


print(X_scaler.mean_)
print(X_scaler.var_)

[ 9.06666667 161.6 77.8 ]
[ 28.46222222 3795.17333333 2861.62666667]
```

$$x' = \frac{x - \mu}{\sigma}$$

Scaling output variables such that $y' \sim [0, 1]$

```
# linear scaling up to [0,1] for outputs
y_scaler = preprocessing.MinMaxScaler().fit(y_train)
y_scaled = y_scaler.transform(y_train)


print(y_scaler.scale_)  #
print(y_scaler.min_)

[0.00917431 0.06666667 0.03571429]
[-1.26605505 -2.06666667 -1.64285714]
```

$$y' = \underbrace{\left(\frac{1}{y_{max} - y_{min}}\right)}_{scale} y + \underbrace{\left(\frac{-y_{min}}{y_{max} - y_{min}}\right)}_{min}$$

## Implementing **Perceptron Layer** using pytorch libraries:

**from** torch **import** nn   # torch nn module provides classes for build custom neural networks

**nn.Module**

- The base class for all neural network modules.
- Your models should also subclass this class.

**nn.Linear**

- A class that implements a linear transformation $y = w^T x + b$

**nn.Sigmoid**

- A class that implements a Sigmoid activation function

**nn.Sequential**

- A sequential container.
- A sub-class of the base class, which can accommodate a sequence of modules, allows treating the whole container as a single module

# GD for a perceptron layer with PyTorch nn.Module class

```
# Create a perceptron layer class
class PerceptronLayer(nn.Module):
  def __init__(self, no_inputs, no_outputs):
      super().__init__()
      self.perceptron_layer = nn.Sequential(
          nn.Linear(no_inputs, no_outputs),
          nn.Sigmoid()
      )

  def forward(self, x):
      logits = self.perceptron_layer(x)
      return logits


# create an instance of the layer
no_inputs, no_outputs = 3, 3
model = PerceptronLayer(no_inputs, no_outputs)
```

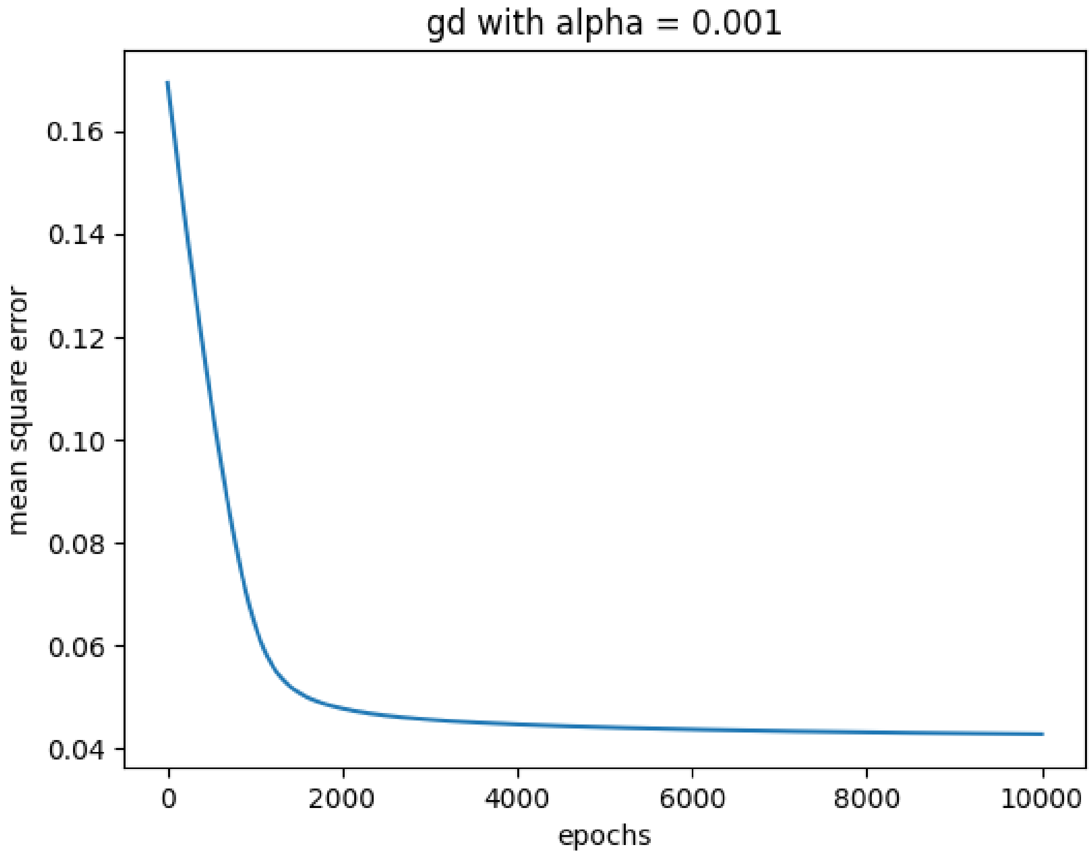
## Instantiate the loss function and the optimizer

```
loss_fn = torch.nn.MSELoss()
optimizer = torch.optim.SGD(model.parameters(), lr=.001)
```

## GD for a perceptron layer with PyTorch autograd

```
no_epochs, lr = 50000, 0.001
for epoch in range(no_epochs):
    # Compute prediction and loss
    pred = model(torch.tensor(X_scaled, dtype=torch.float))
    loss = loss_fn(pred, torch.tensor(y_scaled, dtype=torch.float))

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

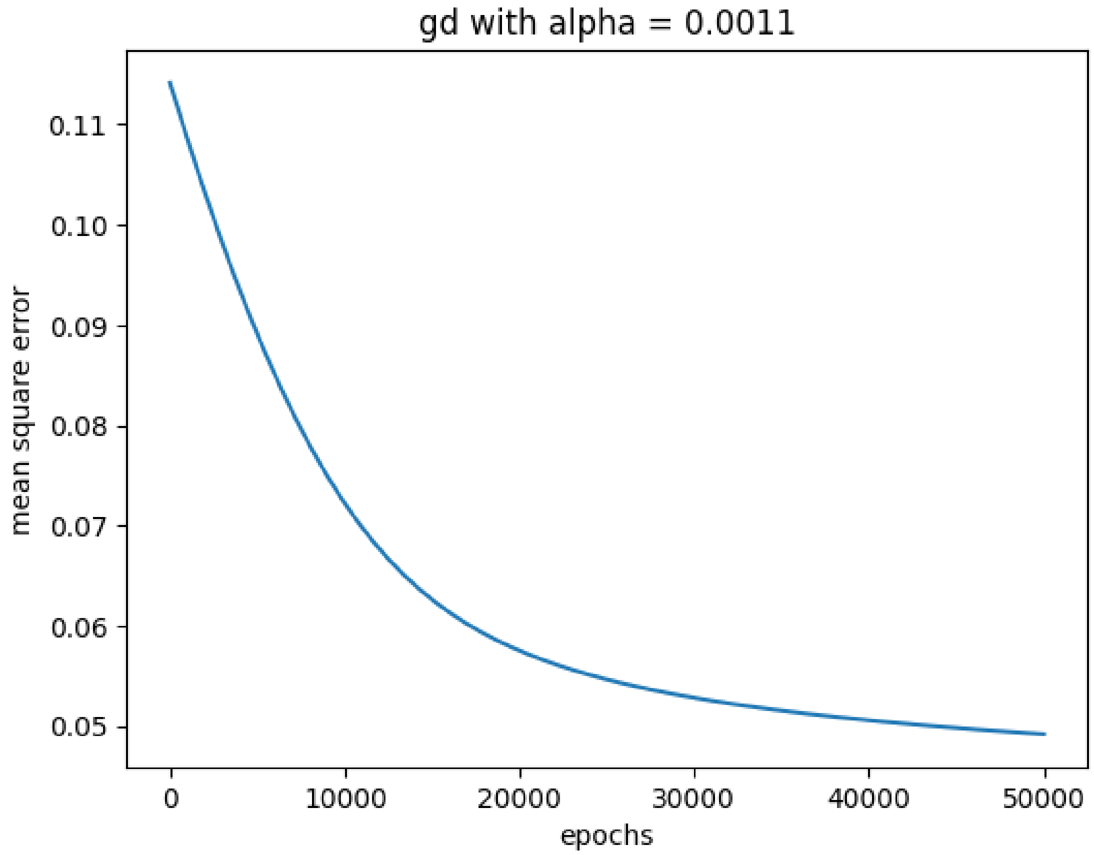
**Direct gradients (using formulae taught in the class)**




Time for weight update = 0.074ms

Number of epochs = 10,000

**Autograd**

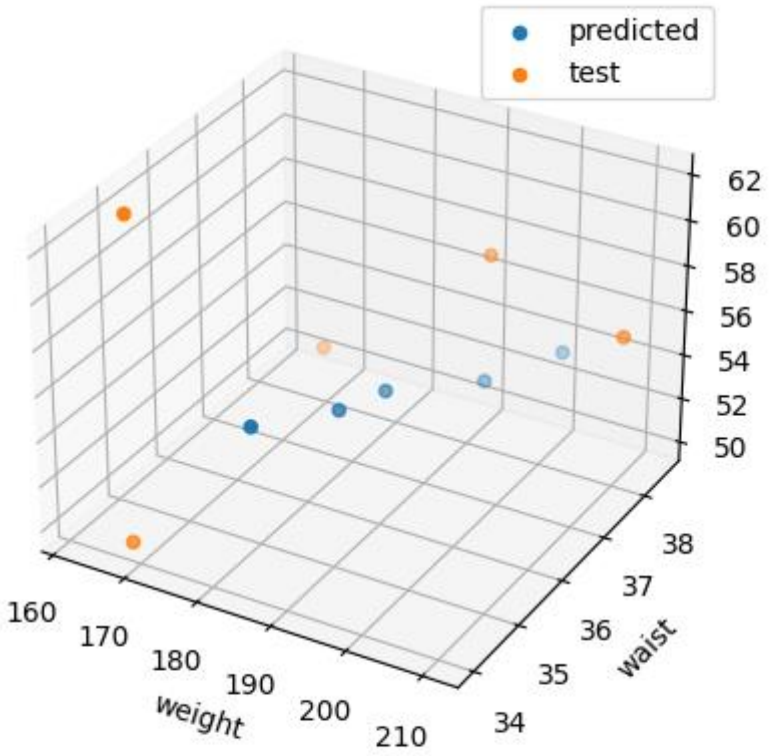



Time for weight update = 0.114ms

Number of epochs = 50,000

```
# scaling testing inputs
X_scaled = X_scaler.transform(X_test)

# predict the outputs
y_pred = model(X_scaled)

# scaling predicted outputs
y_scaled = y_scaler.inverse_transform(y_pred.detach().numpy())



# computing mean square error
from sklearn.metrics import root_mean_squared_error

rms = root_mean_squared_error(y_scaled, y_test, multioutput='raw_values')
mse = rms*rms

# computing R2
from sklearn.metrics import r2_score

r2 = r2_score(y_scaled, y_test,  multioutput='raw_values')
```

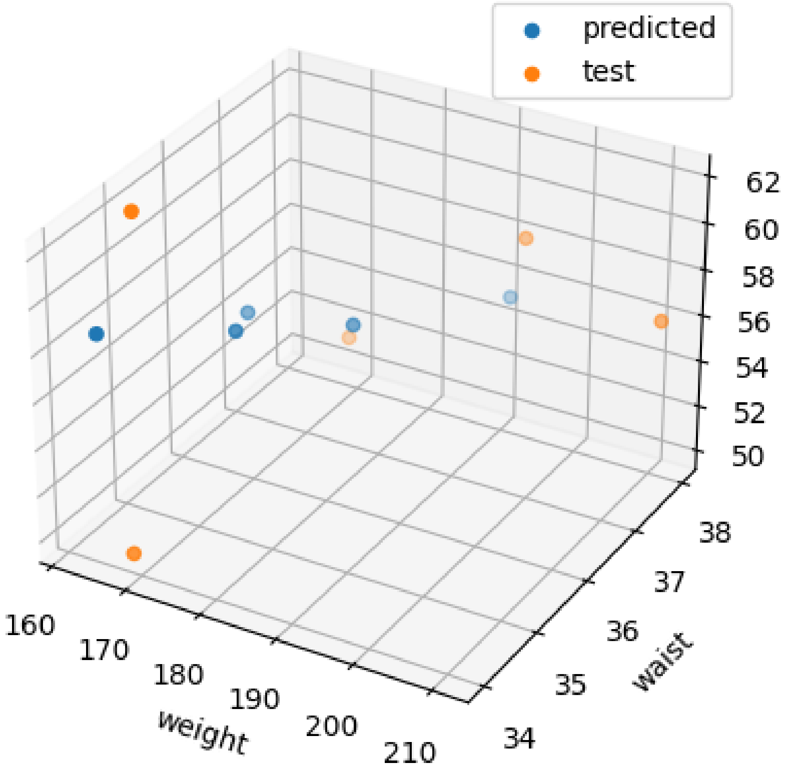
## Direct gradients




MSE = [407.06558087, 1.21179368, 22.40310961]

$R^2$ = [-15.16134924, 0.53783585, -11.22642492]

## Autograd




MSE = [356.95227178, 1.61016719, 21.11763412]

$R^2$ = [ -3.18467906, 0.06741378, -11.90990362]

Exercise best predicts the waist size!

# $R^2$: coefficient of determination

Let $y_i$ be the predicted value of target $d_i$ and the number of samples be $P$:

The **residual** sum of square error $SS_{res} = \sum_{p=1}^{P}\left(y_p - d_p\right)^2$

The mean of output targets, $\bar{y} = \frac{1}{P}\sum_{p=1}^{P} d_p$.

The **total** sum of squares (proportional to the variance), $SS_{tot} = \sum_{p=1}^{P}\left(\bar{y} - d_p\right)^2$

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

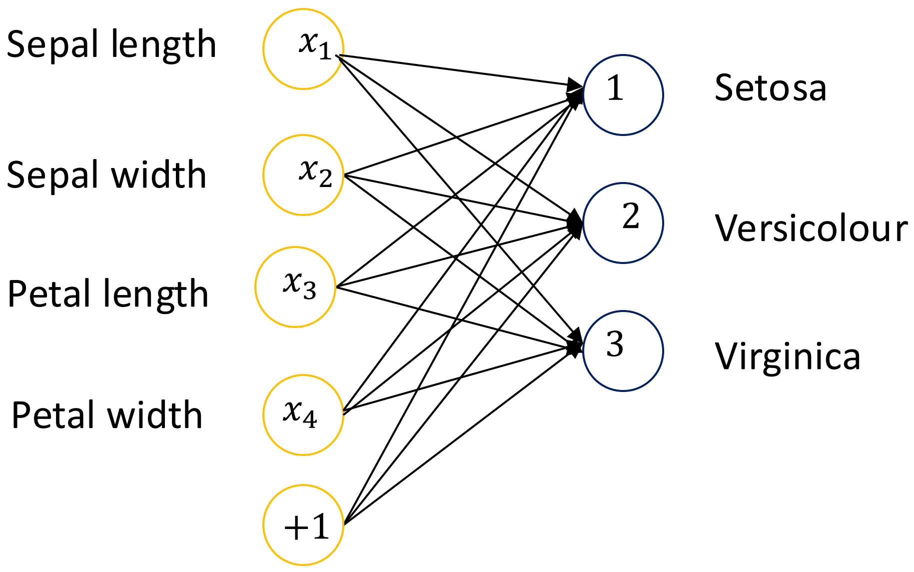
In the best case, the modelled value exactly match the observed value $SS_{res}$=0, then $R^2 = 1$.
The baseline model which predicts $\bar{y}$, then $R^2 = 0$.
The models that have worse prediction than baseline models have negative $R^2$.

3. Use mini-batch gradient decent learning to train a softmax layer to classify Iris dataset (https://archive.ics.uci.edu/ml/datasets/Iris). The dataset contains 150 data points. Use 90 data points for training the classifier and the remaining 60 data points for testing. Plot cross-entropies and classification accuracies against epochs for both train and test data. Set learning rate = 0.1, batch size = 16, and number of epochs = 1000.

   You can use the following python commands to load Iris data:

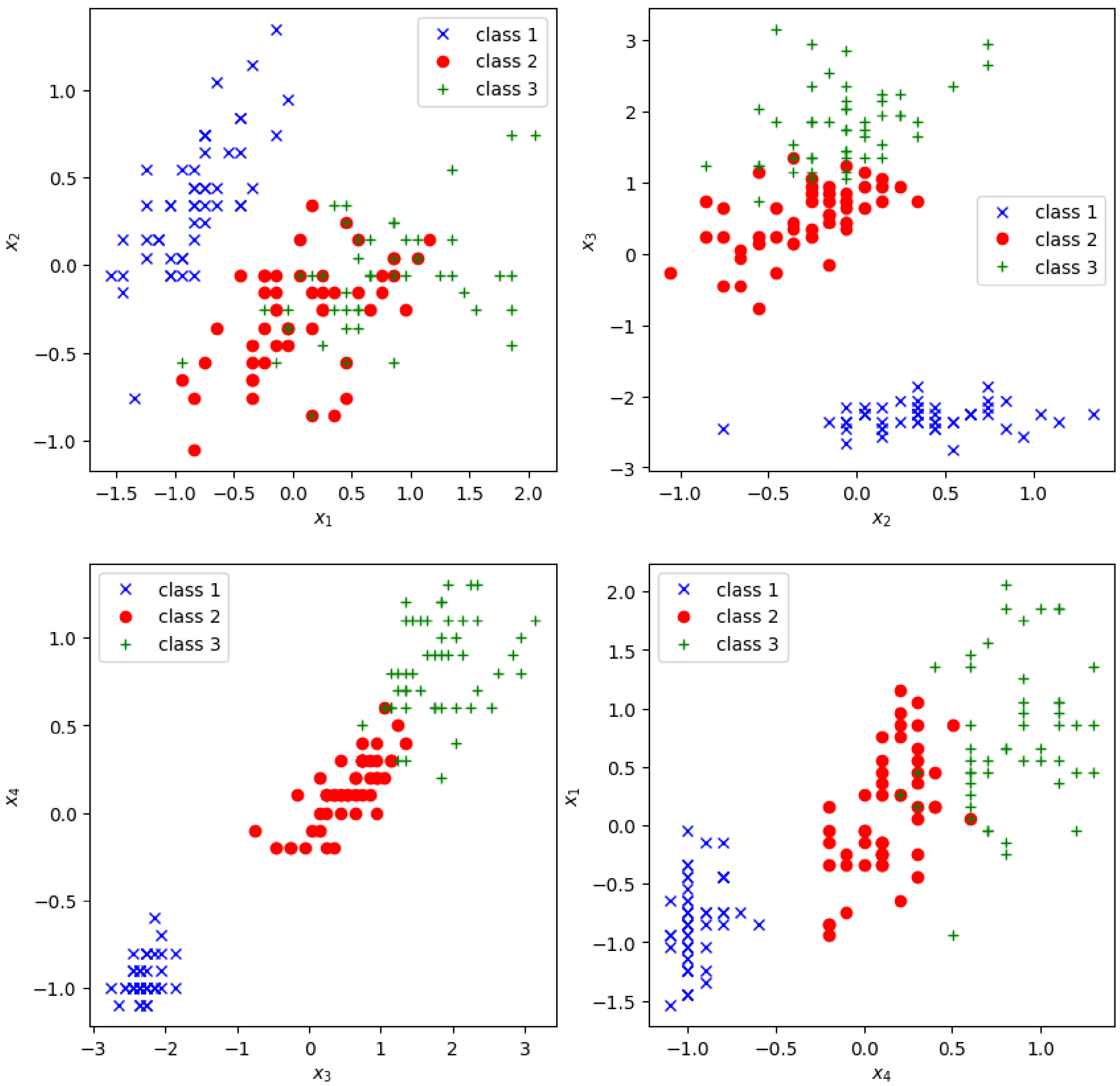
   ```
   from sklearn import datasets
   iris = datasets.load_iris()
   ```

   Repeat the classification with batch sizes = 2, 4, 8, 16, 24, 32, and 64, and compare the accuracies and the times taken for an epoch at different batch sizes.




**Four features** and **three labels**

**90** data points for training and **60** data points for testing boundaries in 2-dimensional feature spaces



```
# datasets import
from sklearn import datasets
from sklearn.model_selection import train_test_split

iris = datasets.load_iris()

# mean correct data
iris.data -= np.mean(iris.data, axis=0)

# dataset split for train and test
x_train, x_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=2)
```

```python
from torch import nn

class SoftmaxLayer(nn.Module):
  def __init__(self, no_inputs, no_outputs):
        super().__init__()
        self.softmax_layer = nn.Sequential(
            nn.Linear(no_inputs, no_outputs),     # applies a linear transformation y = w^T x + b
            nn.Softmax(dim=1)       # implements softmax; sum up across rows to 1.0
        )

  def forward(self, x):
        logits = self.softmax_layer(x)
        return logits
```

Mini-batch gradient descent
Batch size = 16, learning factor = 0.1

Torch provides two data primitives **Dataset** and **Dataloader** to easily manipulate data for mini-batch learning

```
from torch.utils.data import Dataset
from torch.utils.data import DataLoader
```

We will be creating a subclass of **Dataset** class and using **Dataloaders** to implement **mini-batch gradient descent**

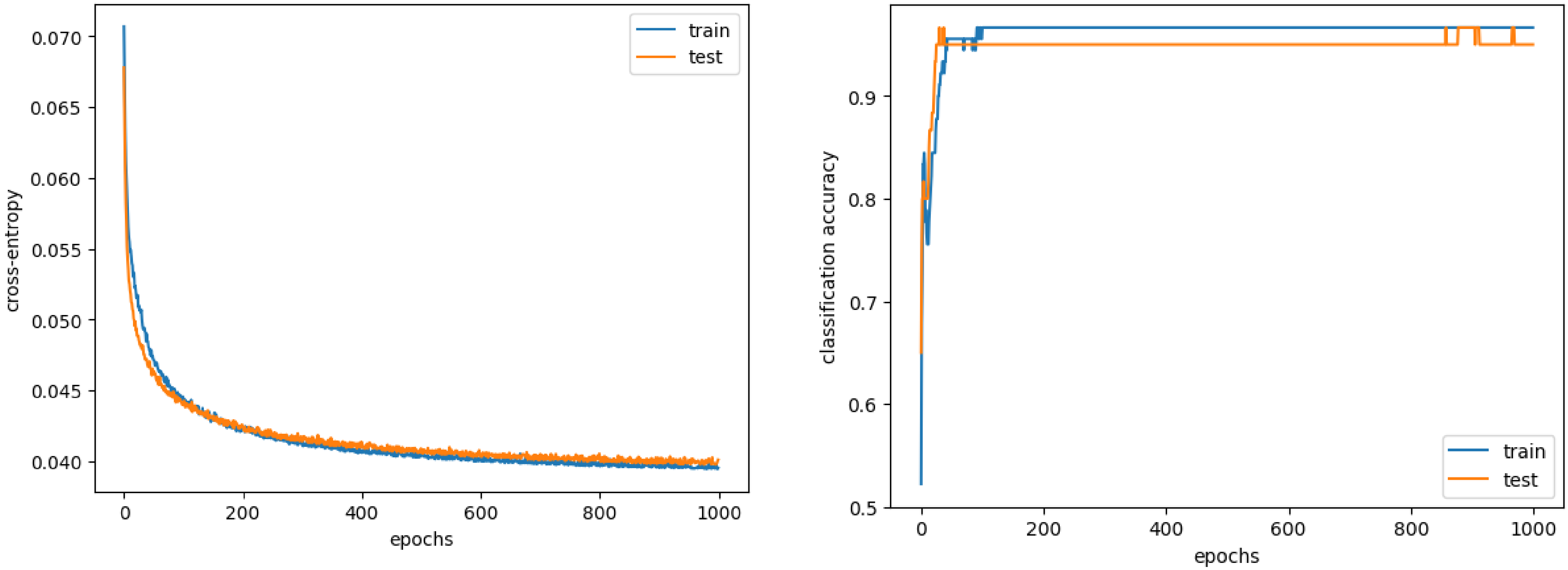
```python
# create a Dataset class
# A custom Dataset class must implement three functions: __init__, __len__, and __getitem__.
class MyDataset(Dataset):
  def __init__(self, X, y):
    self.X =torch.tensor(X, dtype=torch.float)
    self.y =torch.tensor(y)

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

  def __getitem__(self,idx):
    return self.X[idx], self.y[idx]


# create Dataset objects for train and test data
train_data = MyDataset(x_train, y_train)
test_data = MyDataset(x_test, y_test)

# create DataLoader objects that is iterative over the batches
train_dataloader = DataLoader(train_data, batch_size=batch_size, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=batch_size, shuffle=True)
```

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset) # dataset size
    num_batches = len(dataloader)

    train_loss, correct = 0, 0
    for batch, (X, y) in enumerate(dataloader):

        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        optimizer.zero_grad().  #initialize gradient calculations
        loss.backward()          # compute gradients
        optimizer.step()         # execute one step of SGD

        train_loss += loss.item()
        correct += (pred.argmax(dim=1) == y).type(torch.float).sum().item()

    train_loss /= size
    correct /= size
    return train_loss, correct
```

```python
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(dim=1) == y).type(torch.float).sum().item()

    test_loss /= size
    correct /= size

    return test_loss, correct
```

```
train_loss_, train_acc_, test_loss_, test_acc_ = [], [], [], []

for epoch in range(no_epochs):
    train_loss, train_acc = train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loss, test_acc = test_loop(test_dataloader, model, loss_fn)

    train_loss_.append(train_loss), train_acc_.append(train_acc)
    test_loss_.append(test_loss), test_acc_.append(test_acc)
```

# Learning curves at batch-size = 16



**At convergence (1000 epochs):**

weight = [[-0.875575 1.9825934 -4.016873 -1.6209128 ]
[ 0.70730793 -1.0039392 -0.2982792 -2.5409048 ]
[-0.3355393 -0.79086286 3.4620962 3.9241226 ]]

bias = [-0.6561739 3.990793 -2.9169736]

train_loss = 0.039542
train_acc = 0.966667
test_loss = 0.040103,
test_acc = 0.950000

# Accuracies and time-to-update weights against batch-size