

## ✓ Lab 2 Report:

### Iris Classification with Regression

#### ✓ Name:

```
# Import necessary packages

%matplotlib inline

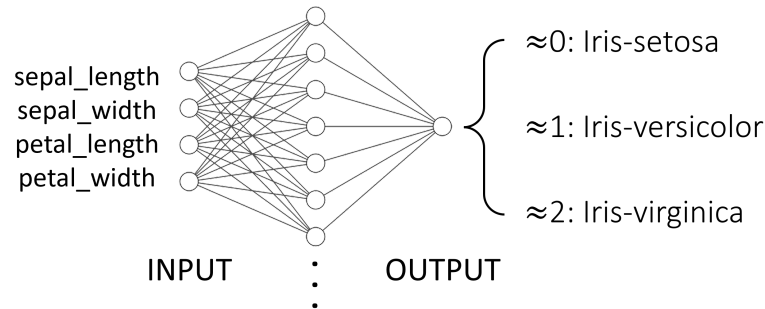
import numpy as np
import matplotlib.pyplot as plt
import torch

from IPython.display import Image # For displaying images in colab jupyter cell

Image('lab2_exercise1.PNG', width = 1000)
```



## Exercise 1: Iris Classification using Regression



In this exercise, you will train a neural network with a single hidden layer consisting of linear neurons to perform regression on iris datasets.

Your goal is to achieve a training accuracy of >90% under 50 epochs.

You are free to experiment with different data normalization methods, size of the hidden layer, learning rate and epochs.

You can round the output value to an integer (e.g. 0.34 -> 0, 1.78 -> 2) to compute the model accuracy.

Demonstrate the performance of your model via plotting the training loss and printing out the training accuracy.

42

#### ✓ Prepare Data

```
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
# iris dataset is available from scikit-learn package
iris = load_iris()

# Load the X (features) and y (targets) for training
X_train = iris['data']
y_train = iris['target']

# Load the name labels for features and targets
feature_names = iris['feature_names']
names = iris['target_names']

# split into train and testing data
# comment this
```

```

X_train, X_test, y_train, y_test = train_test_split(X_train, y_train, test_size=0.2, random_state=42)

# scale data
# use imported standard scaler class from scikit learn to quickly preprocess data
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.fit_transform(X_test)

# convert data to tensor format, optimized for training models
# the output (y_test, y_train) needs to be long datatype to avoid compatibility issues with loss function
X_train = torch.tensor(X_train, dtype=torch.float32)
y_train = torch.tensor(y_train, dtype=torch.long)
X_test = torch.tensor(X_test, dtype=torch.float32)
y_test = torch.tensor(y_test, dtype=torch.long)

# reform dataset now that everything is transformed to tensor datatype
# small datasets for fast and lightweight training, and randomize to mitigate bias
train_dataset = torch.utils.data.TensorDataset(X_train, y_train)
train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=16, shuffle=True)

# Print the first 10 training samples for both features and targets
print(X_train[:10, :], y_train[:10])

↵ tensor([[[-1.4739,  1.2037, -1.5625, -1.3126],
          [-0.1331,  2.9924, -1.2760, -1.0456],
          [ 1.0859,  0.0857,  0.3859,  0.2892],
          [-1.2301,  0.7565, -1.2187, -1.3126],
          [-1.7177,  0.3093, -1.3906, -1.3126],
          [ 0.5983, -1.2558,  0.7297,  0.9566],
          [ 0.7202,  0.3093,  0.4432,  0.4227],
          [-0.7426,  0.9801, -1.2760, -1.3126],
          [-0.9863,  1.2037, -1.3333, -1.3126],
          [-0.7426,  2.3216, -1.2760, -1.4461]]) tensor([0, 0, 1, 0, 0, 2, 1, 0, 0, 0])

# Print the dimensions of features and targets
print(X_train.shape, y_train.shape)

↵ torch.Size([120, 4]) torch.Size([120])

# feature_names contains name for each column in X_train
# For targets, 0 -> setosa, 1 -> versicolor, 2 -> virginica
print(feature_names, names)

↵ ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)'] ['setosa' 'versicolor' 'virginica']

# We can visualize the dataset before training
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16, 6))

# enumerate picks up both the index (0, 1, 2) and the element ('setosa', 'versicolor', 'virginica') from "names"
# loop 1: target = 0, target_name = 'setosa'
# loop 2: target = 1, target_name = 'versicolor' etc
for target, target_name in enumerate(names):

    # Subset the rows of X_train that fall into each flower category using boolean mapping
    X_plot = X_train[y_train == target]

    # Plot the sepal length versus sepal width for the flower category
    ax1.plot(X_plot[:, 0], X_plot[:, 1], linestyle='none', marker='o', label=target_name)

# Label the plot
ax1.set_xlabel(feature_names[0])
ax1.set_ylabel(feature_names[1])
ax1.axis('equal')
ax1.legend()


# Repeat the above process but with petal length versus petal width
for target, target_name in enumerate(names):

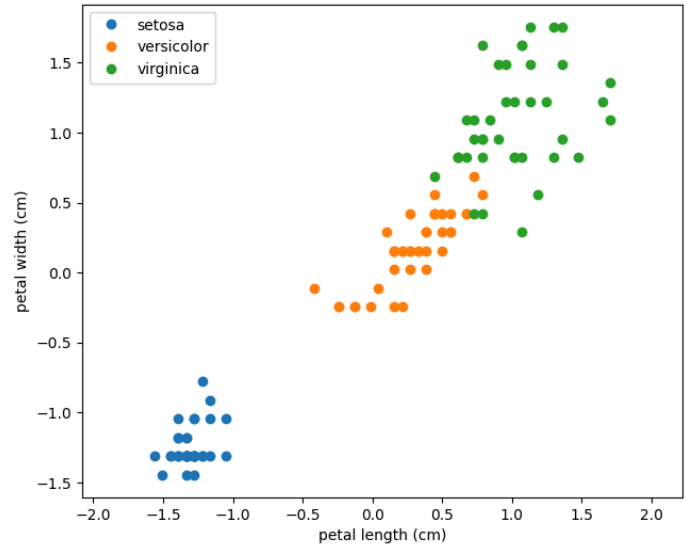
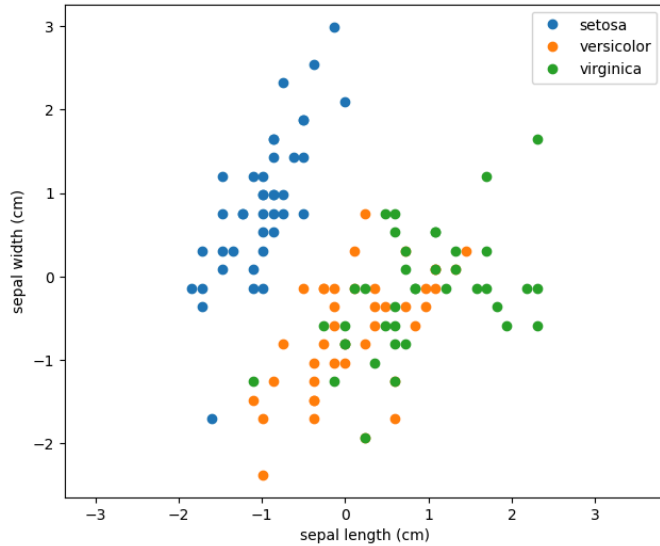
    X_plot = X_train[y_train == target]

```

```
ax2.plot(X_plot[:, 2], X_plot[:, 3], linestyle='none', marker='o', label=target_name)
```

```
ax2.set_xlabel(feature_names[2])
ax2.set_ylabel(feature_names[3])
ax2.axis('equal')
ax2.legend()
```

 <matplotlib.legend.Legend at 0x303b54740>



## ✓ Define Model

```
class irisClassification(torch.nn.Module):
    # input dim auto initialized to 4 because Iris has 4 features
    # hidden dim auto is initialized to 16 neurons because the problem scope is not intensive
    # output dim auto is initialized to 3 because there are 3 types (Setosa, Versicolor, Virginica)
    def __init__(self, input_dim=4, hidden_dim=16, output_dim=3):
        super(irisClassification, self).__init__()

        self.layer1 = torch.nn.Linear(input_dim, hidden_dim)
        self.layer2 = torch.nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        # use relu activation function for speed and efficiency
        temp = torch.nn.functional.relu(self.layer1(x))
        out = torch.nn.functional.relu(self.layer2(temp))

        return out
```

## ✓ Define Hyperparameters

```
model = irisClassification()

# choose a higher learning rate to account for limited number of epochs
learning_rate = 0.15
epochs = 40 # < 50

# We will use gradient descent for our optimizer and Cross Entropy Loss function
optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)
loss_func = torch.nn.CrossEntropyLoss()
```

## ✓ Identify Tracked Values

```
# follow models performance over each epoch. Identify a metric and track it over epochs
training_loss_list = []
```

## ✓ Train Model

```
# here we train the model
for epoch in range(epochs):
    epoch_loss = 0.0

    # train model in batches for computational efficiency and faster convergence
    # compute loss for each batch and update weights/biases
    for batch_X, batch_y in train_loader:
        outputs = model(batch_X)
        loss = loss_func(outputs, batch_y)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        epoch_loss += loss.item()

    # every 5th epoch, print the loss
    if epoch % 5 == 0:
        print(f"Epoch {epoch}: Loss = {loss.item():.4f}")

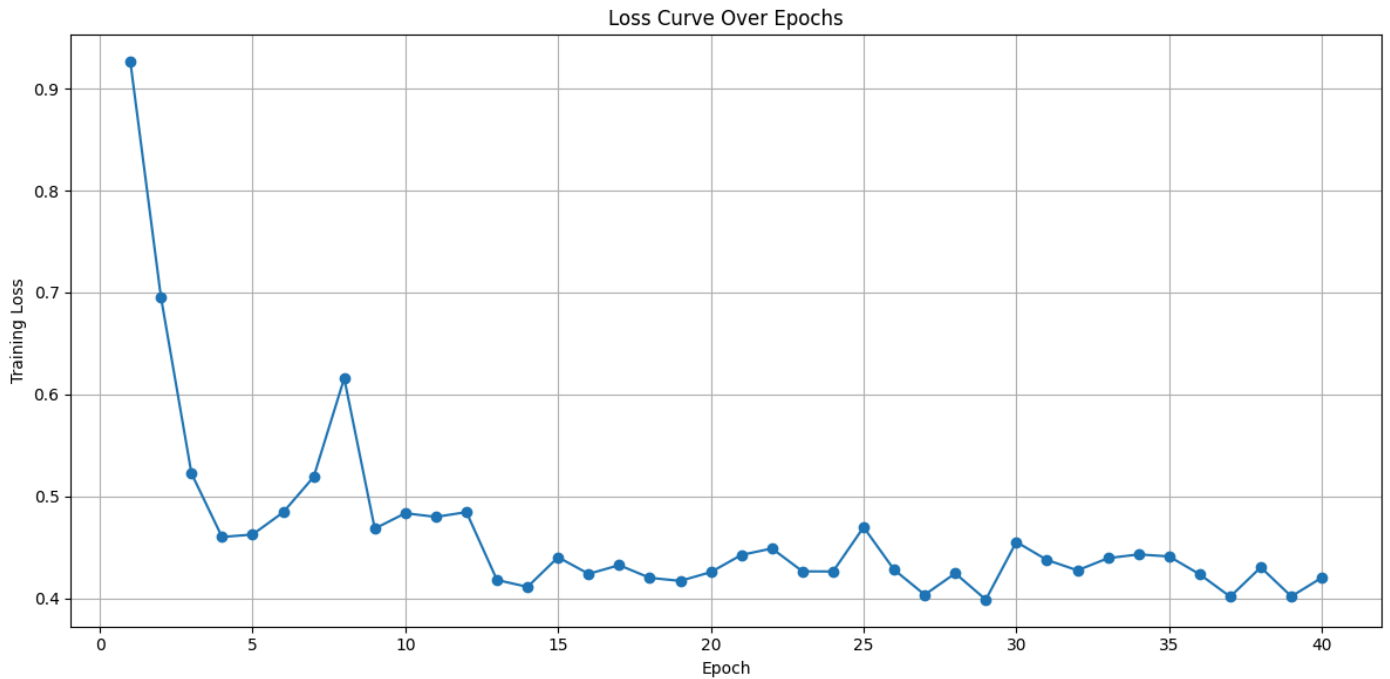
    # track loss per epoch
    avg_loss = epoch_loss / len(train_loader)
    training_loss_list.append(avg_loss)
```

```
↻ Epoch 0: Loss = 0.7122
Epoch 5: Loss = 0.5579
Epoch 10: Loss = 0.2564
Epoch 15: Loss = 0.2763
Epoch 20: Loss = 0.6282
Epoch 25: Loss = 0.4147
Epoch 30: Loss = 0.4136
Epoch 35: Loss = 0.5524
```

## ✓ Visualize and Evaluate Model

```
# Plot your training loss throughout the training
# Include proper x and y labels for the plot

plt.figure(figsize=(12, 6))
plt.plot(range(1, epochs + 1), training_loss_list, marker='o')
plt.xlabel('Epoch')
plt.ylabel('Training Loss')
plt.title('Loss Curve Over Epochs')
plt.grid(True)
plt.tight_layout()
plt.show()
```



```
# Confirm that your model's training accuracy is >90%
# Training accuracy = (# of correct predictions) / (total # of training samples)
# You can round the model predictions to integer (e.g. 0.34 -> 0, 1.78 -> 2)
```

```
with torch.no_grad():
    preds = model(X_test)
    predicted_classes = preds.argmax(dim=1)
    accuracy = (predicted_classes == y_test).float().mean()
    print(f"Test Accuracy: {accuracy.item():.4f}")
```

Test Accuracy: 0.9333

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
```

```
# Get predictions from model
with torch.no_grad():
    logits = model(X_test)
    preds = torch.argmax(logits, dim=1)
```

```
# Perform PCA on X_test to reduce to 2D
pca = PCA(n_components=2)
X_test_2d = pca.fit_transform(X_test)
```

```
# Plot true labels vs predicted labels
fig, ax = plt.subplots(1, 2, figsize=(12, 5))
```

```
# Ground truth
ax[0].scatter(X_test_2d[:, 0], X_test_2d[:, 1], c=y_test, cmap='viridis', edgecolor='k')
ax[0].set_title('True Labels')
ax[0].set_xlabel('PCA 1')
ax[0].set_ylabel('PCA 2')
```

```
# Predicted labels
ax[1].scatter(X_test_2d[:, 0], X_test_2d[:, 1], c=preds, cmap='viridis', edgecolor='k')
ax[1].set_title('Predicted Labels')
ax[1].set_xlabel('PCA 1')
ax[1].set_ylabel('PCA 2')
```

```
plt.suptitle("Iris Dataset Classification - PCA View")
plt.tight_layout()
```



Iris Dataset Classification - PCA View

