Multiclass Classification using ANN

(Pattern recognition)

Iris flower
Case study

Problem to be solved

❖ Classify Iris flowers

The famous **Iris database** was first used by Sir R.A. Fisher. The dataset is taken from Fisher's paper.

- Fisher, R.A. "The use of multiple measurements in taxonomic problems“ Annual Eugenics, 7, Part II, 179-188 (1936); also in "Contributions to Mathematical Statistics" (John Wiley, NY, 1950).

This is perhaps the best-known database to be found in the pattern recognition literature.

The data set contains **3 classes of 50 instances each** (balanced dataset), where each class refers to a type of iris flower.

One class is linearly separable from the other 2; the latter are not linearly separable from each other.

Iris flowers

Iris setosa

Iris virginica

Iris versicolor

Iris dataset

Dataset Characteristics

Number of Instances: 150 (50 in each of three classes)
Number of Attributes: 4 numeric, predictive attributes and the class

Attribute Information

- sepal length in cm
- sepal width in cm
- petal length in cm
- petal width in cm

Class

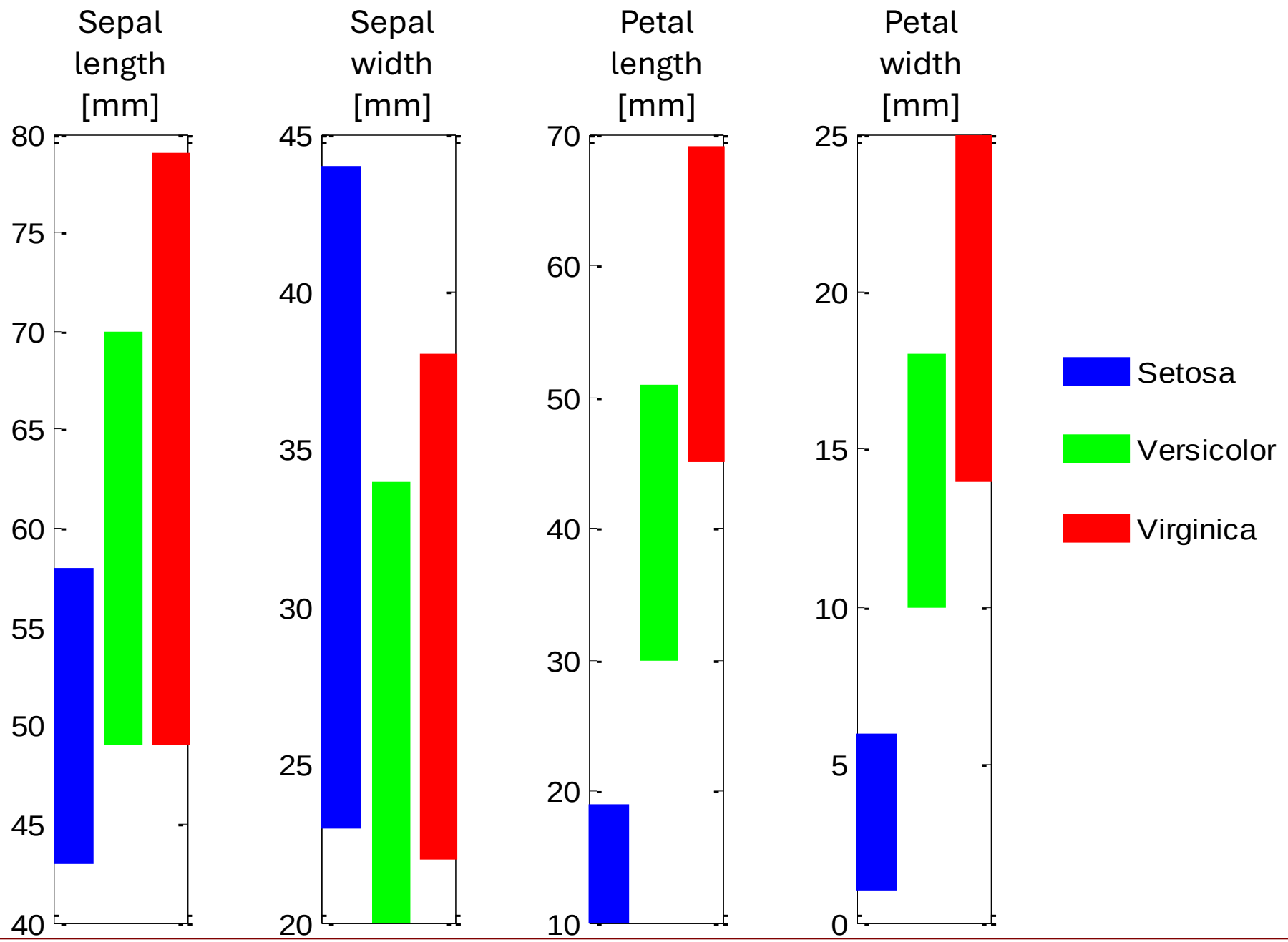
- Iris-Setosa
- Iris-Versicolour
- Iris-Virginica

Summary Statistics

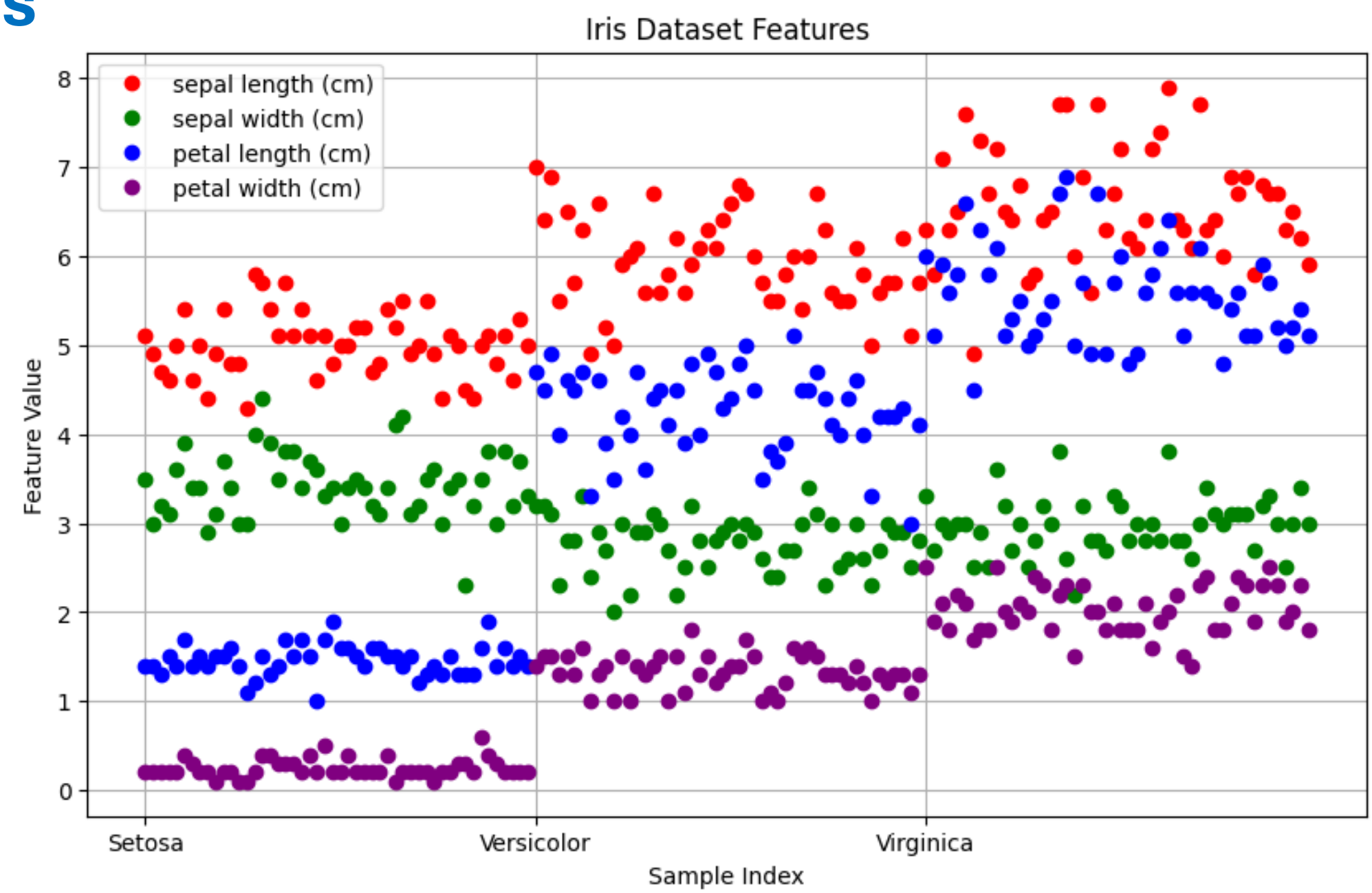
	Min	Max	Mean	SD	Class Correlation
sepal length:	4.3	7.9	5.84	0.83	0.7826
sepal width:	2.0	4.4	3.05	0.43	-0.4194
petal length:	1.0	6.9	3.76	1.76	0.9490 (high!)
petal width:	0.1	2.5	1.20	0.76	0.9565 (high!)

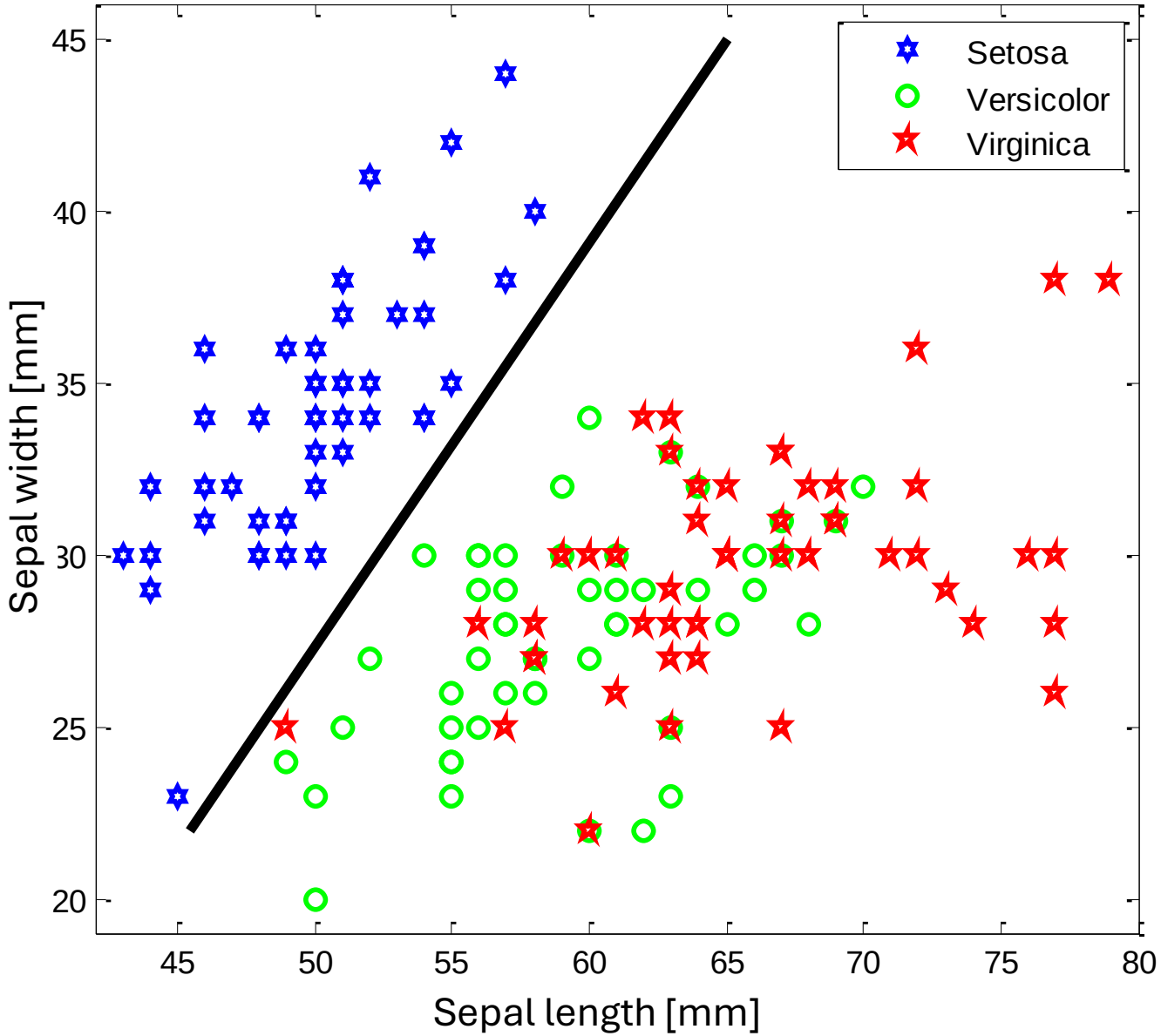
Missing Attribute Values: None
Class Distribution: 33.3% for each of 3 classes.
Creator: R.A. Fisher

Features


Sepal length [mm]
Sepal width [mm]
Petal length [mm]
Petal width [mm]
80
75
70
65
60
55
50
45
40
45
40
35
30
25
20
70
60
50
40
30
20
10
25
20
15
10
5
0
Setosa
Versicolor
Virginica

Features


Iris Dataset Features
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
Feature Value
0
1
2
3
4
5
6
7
8
Setosa
Versicolor
Virginica
Sample Index



Two-dimensional representation of the Iris dataset

Load dataset

```python
# Load the Iris dataset
from sklearn.datasets import load_iris
iris = load_iris()

# Extract features (X) and labels (y)
X = iris.data
y = iris.target
```

Data shape:

X shape: (150, 4)
y shape: (150,)

Iris flowers samples

First 5 samples of class 0 – **Setosa**

```
[[5.1   3.5   1.4   0.2]
 [4.9   3.    1.4   0.2]
 [4.7   3.2   1.3   0.2]
 [4.6   3.1   1.5   0.2]
 [5.    3.6   1.4   0.2]]
```

First 5 labels of class 0 – **Setosa**

```
[0 0 0 0 0]
```

First 5 samples of class 1 – **Versicolor**

```
[[7.    3.2   4.7   1.4]
 [6.4   3.2   4.5   1.5]
 [6.9   3.1   4.9   1.5]
 [5.5   2.3   4.    1.3]
 [6.5   2.8   4.6   1.5]]
```

First 5 labels of class 1 – **Versicolor**

```
[1 1 1 1 1]
```

First 5 samples of class 2 – **Virginica**

```
[[6.3   3.3   6.    2.5]
 [5.8   2.7   5.1   1.9]
 [7.1   3.    5.9   2.1]
 [6.3   2.9   5.6   1.8]
 [6.5   3.    5.8   2.2]]
```

First 5 labels of class 2 – **Virginica**

```
[2 2 2 2 2]
```

Normalize dataset

Data **normalization** is often **a crucial step** in machine learning, especially for algorithms like the ANN

1. Feature Scaling and Algorithm Performance

- **Gradient Descent-Based Algorithms (like ANNs):** These algorithms use gradient descent to find the optimal model parameters.
  - If features have vastly different scales, the loss function will have an elongated shape, making it difficult for gradient descent to converge efficiently.
  - Normalization helps to create a more spherical loss function, leading to faster and more stable convergence.
- **Distance-Based Algorithms (like k-Nearest Neighbors):** These algorithms rely on calculating distances between data points.
  - Features with larger ranges can disproportionately influence distance calculations, potentially leading to inaccurate results.
  - Normalization ensures that all features contribute equally to distance measures.

Normalize dataset

2. Preventing Feature Dominance

- Without normalization, features with larger values can dominate the model's learning process, even if they are not necessarily more important. Normalization levels the playing field, allowing the model to learn relationships between all features more effectively.

3. Improving Numerical Stability

- In some cases, features with very large or very small values can lead to numerical instability during calculations. Normalization can help to mitigate these issues.

4. Enhanced Model Interpretability

- When features are normalized, the model's coefficients (or weights) become more comparable, making it easier to interpret the relative importance of different features.

Normalize dataset

Specific to our ANN Model:

- ✓ The ANN model uses the 'Adam' optimizer, which is a type of gradient descent algorithm. Normalizing the Iris dataset features (sepal length, sepal width, petal length, petal width) ensures that features with different scales (like sepal length and petal width) do not disproportionately influence the weight updates during training. This leads to faster convergence and potentially better model performance.

In summary, normalization is often necessary to:

- ➢ Improve the performance and stability of many machine learning algorithms.
- ➢ Prevent feature dominance and ensure fair contribution from all features.
- ➢ Enhance numerical stability during calculations.
- ➢ Improve the interpretability of the model.

MinMaxScaler

- MinMaxScaler is a data preprocessing technique in scikit-learn used for feature scaling.
- It transforms features by scaling them to a given range, typically between 0 and 1.
- This process is also known as min-max normalization.

x_scaled = (x - x_min) / (x_max - x_min)

✓ **Preserves Data Shape**: MinMaxScaler preserves the original distribution of the data, meaning that the relative relationships between data points are maintained after scaling.

✓ **Handles Outliers**: Outliers can influence the scaling process, but their impact is limited since MinMaxScaler uses the minimum and maximum values of the entire dataset for scaling.

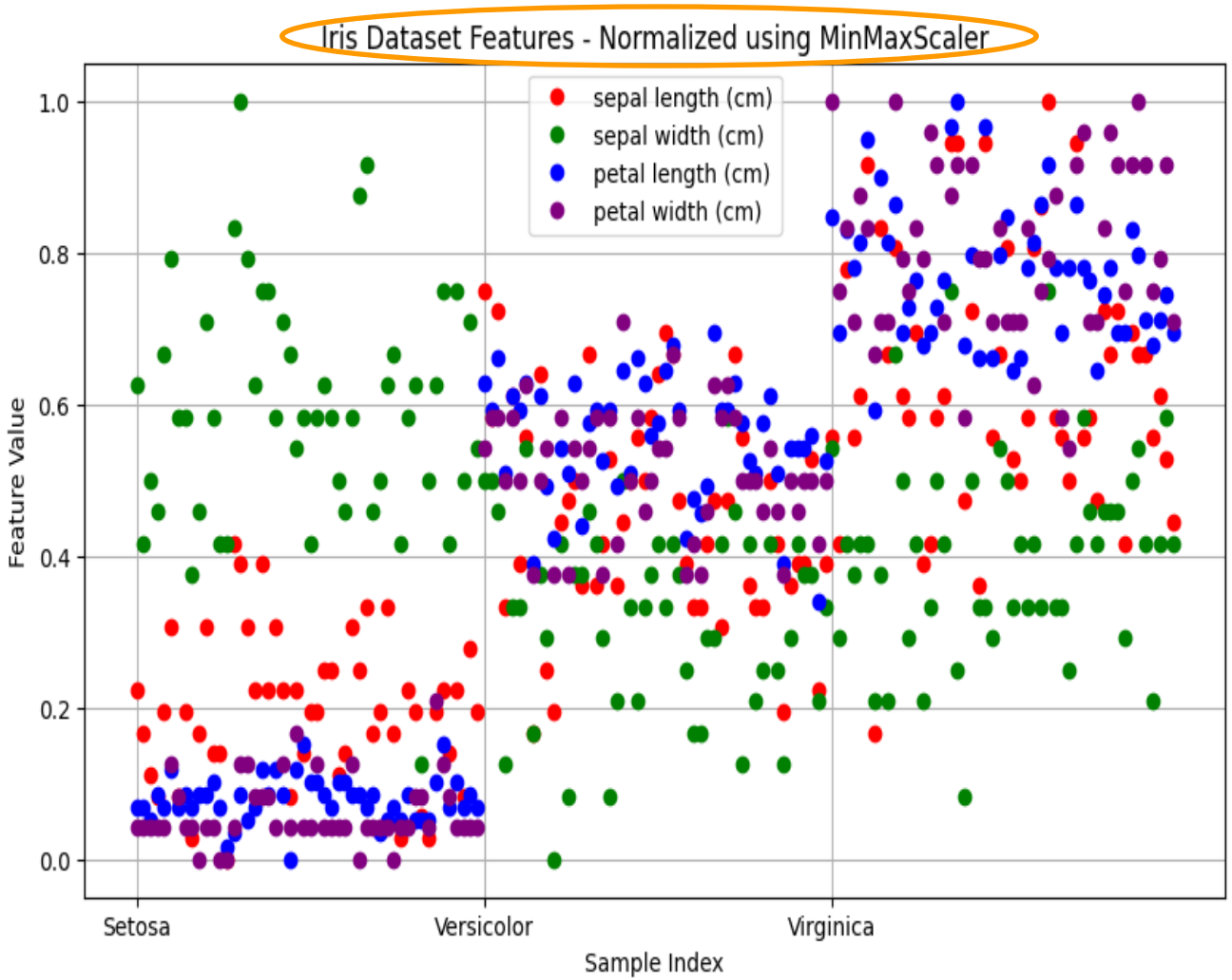
✓ **Simple and Intuitive**: The scaling process is easy to understand and interpret.

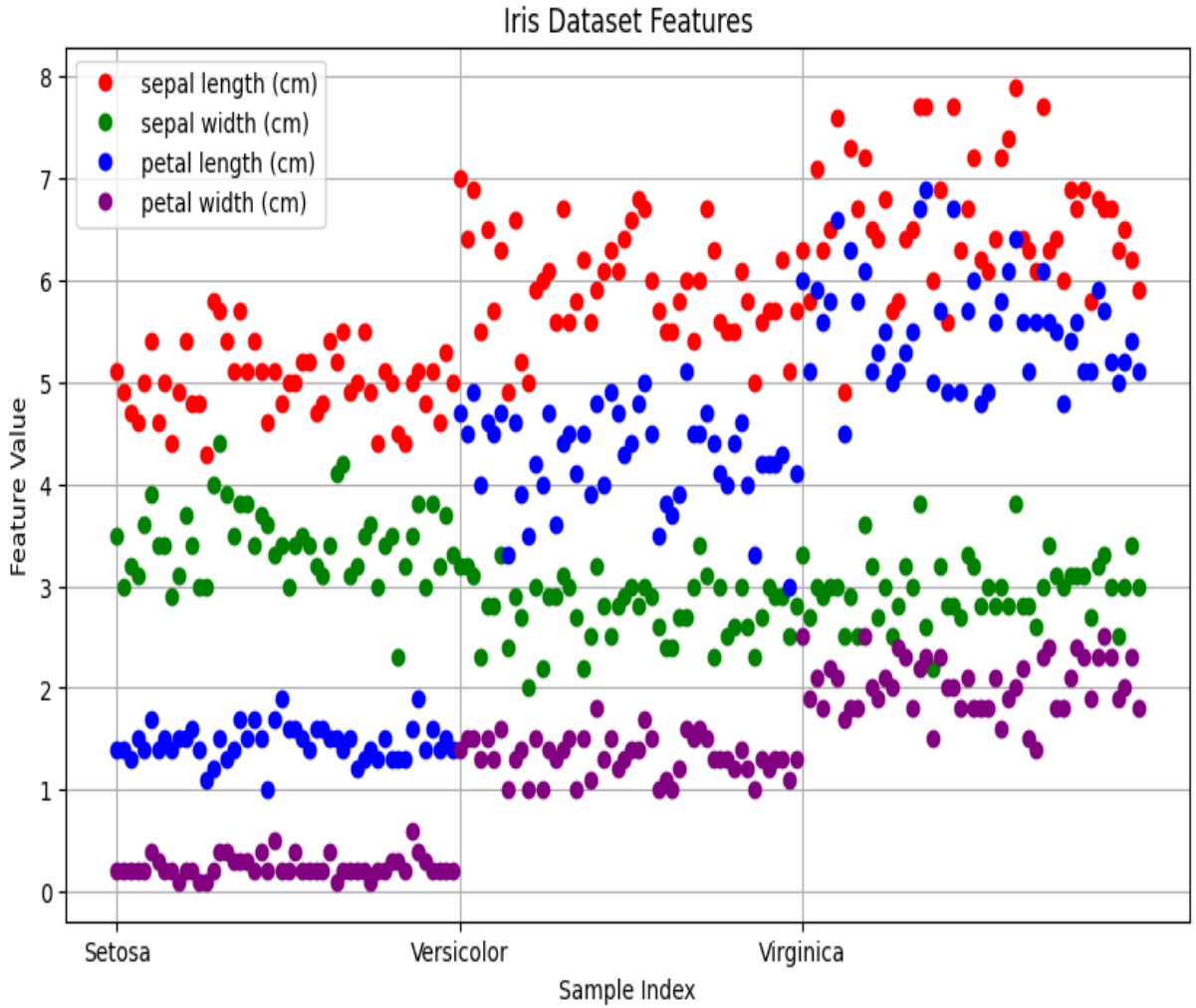
✓ **Suitable for Algorithms Sensitive to Feature Ranges**: Algorithms like k-Nearest Neighbors, Support Vector Machines, and Neural Networks often benefit from feature scaling using MinMaxScaler.

- **Sensitive to New Data:** If new data points with values outside the original range are introduced, the scaler needs to be refitted to include these values, which can affect the scaling of existing data.
- **May Squash Data:** If the data has a wide range, MinMaxScaler can compress the data into a smaller range, potentially losing some information.

```python
from sklearn.preprocessing import MinMaxScaler  # Import MinMaxScaler
# Normalize the data using MinMaxScaler
scaler = MinMaxScaler()
X_normalized = scaler.fit_transform(X)
```

MinMaxScaler





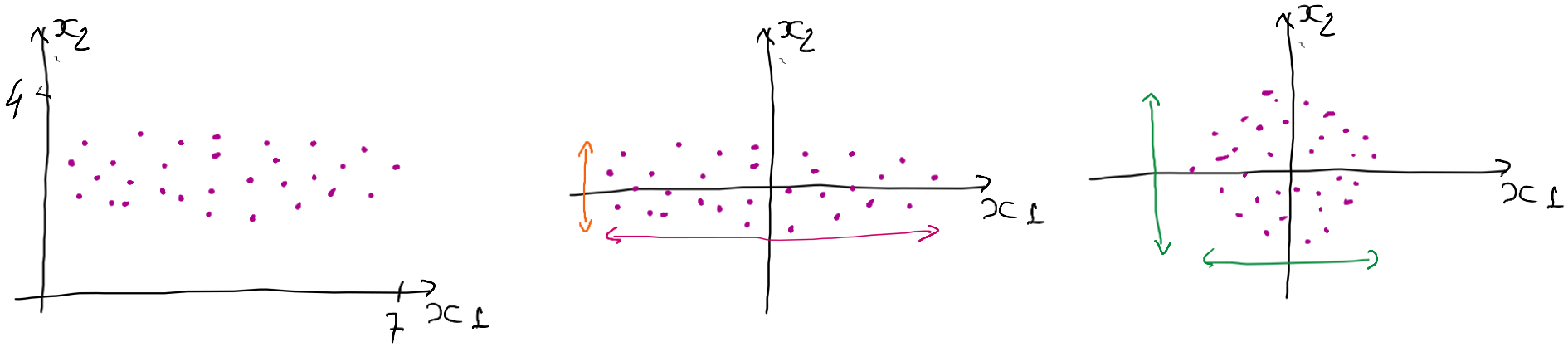
Standard Scaler

$$x\_scaled = \frac{x - x\_mean}{standard\_deviation}$$

Applied separately on each data feature

Use the same μ, σ to normalize all data sets

- ✓ **Training**
- ✓ **Validation**
- ✓ **Test**



Initial dataset

Subtract mean (zero out the mean)

Normalize the variance

Standardizes features by removing the mean and scaling to unit variance.

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

$$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)} \; ; \; \mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$$

$$x := x - \mu$$

μ – mean

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)}\right)^2$$

σ^2 – variance

$$x := \frac{x}{\sigma}$$

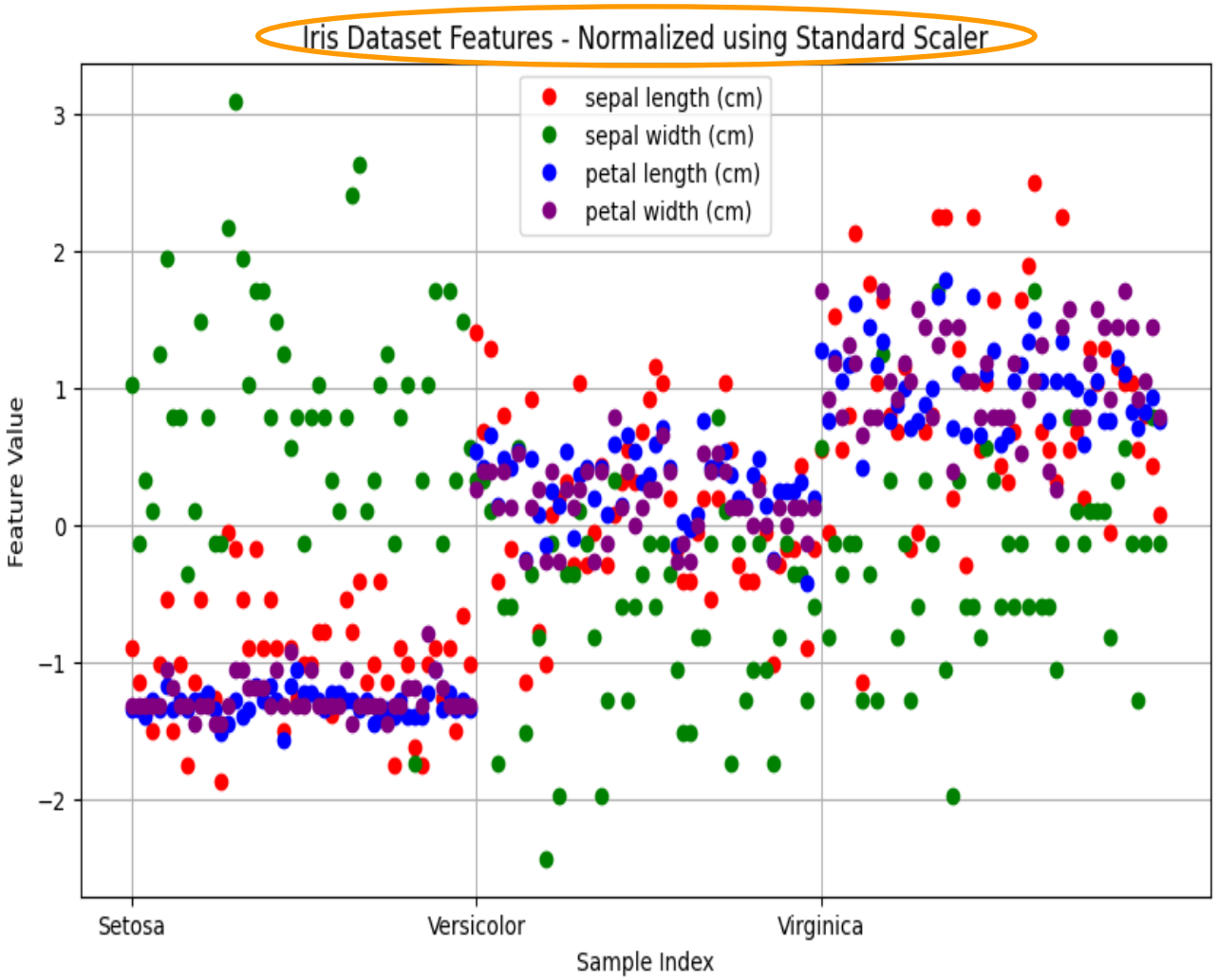
σ – standard deviation $\quad \sigma = \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}$

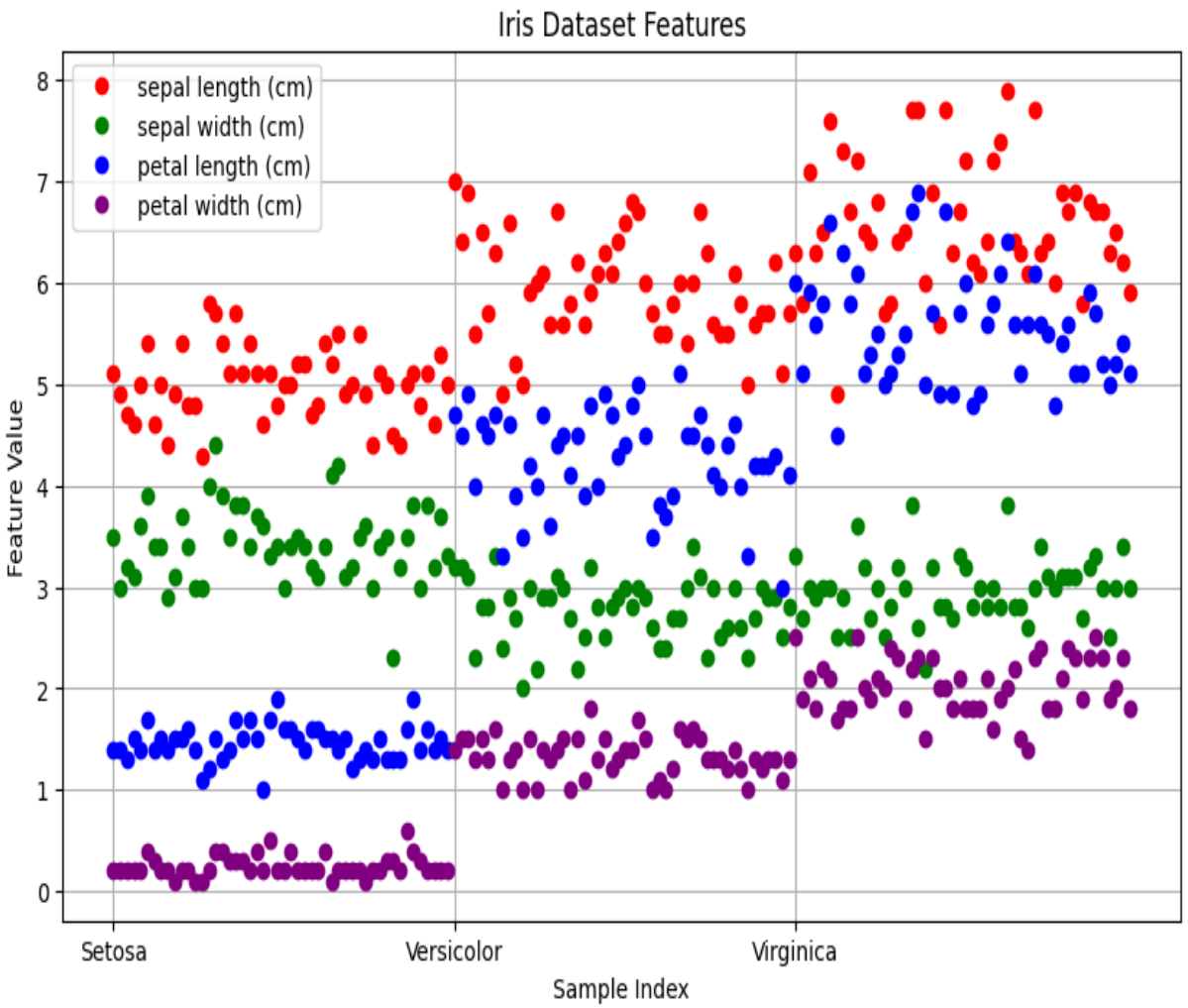
1. **Centering:** The mean of the feature is subtracted from each feature value (x).
   This shifts the distribution of the feature so that its mean becomes 0.
2. **Scaling:** Each centered feature value is then divided by the standard deviation.
   This scales the distribution so that its variance becomes 1.

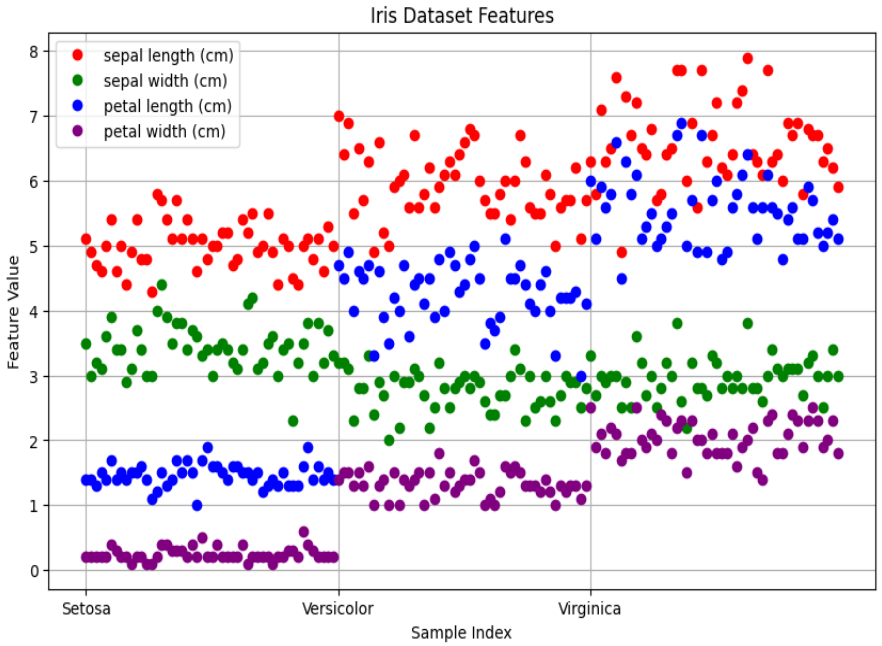
Standard Scaler

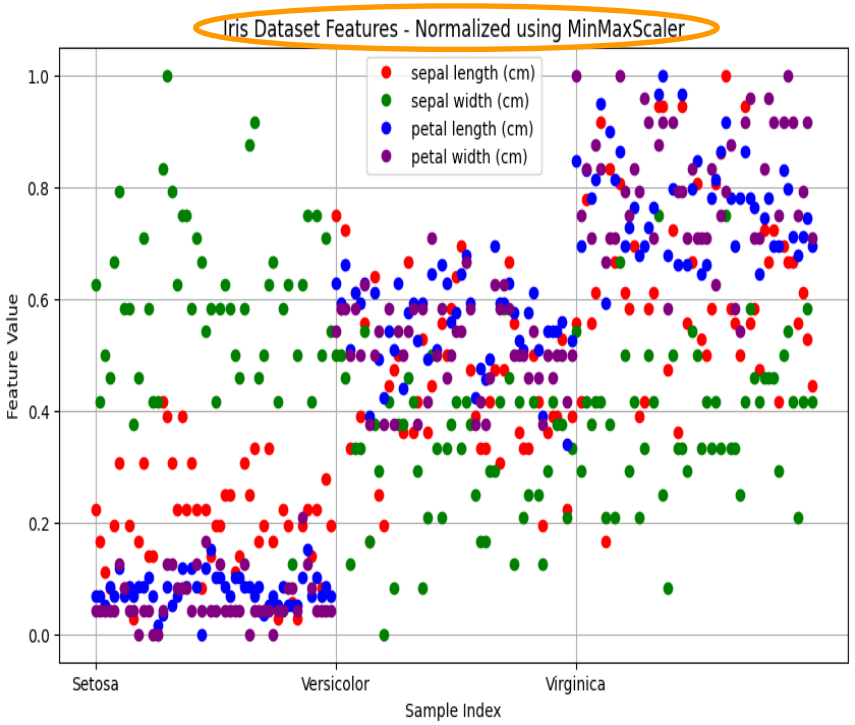
```python
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X_normalized_ss = scaler.fit_transform(X)
```

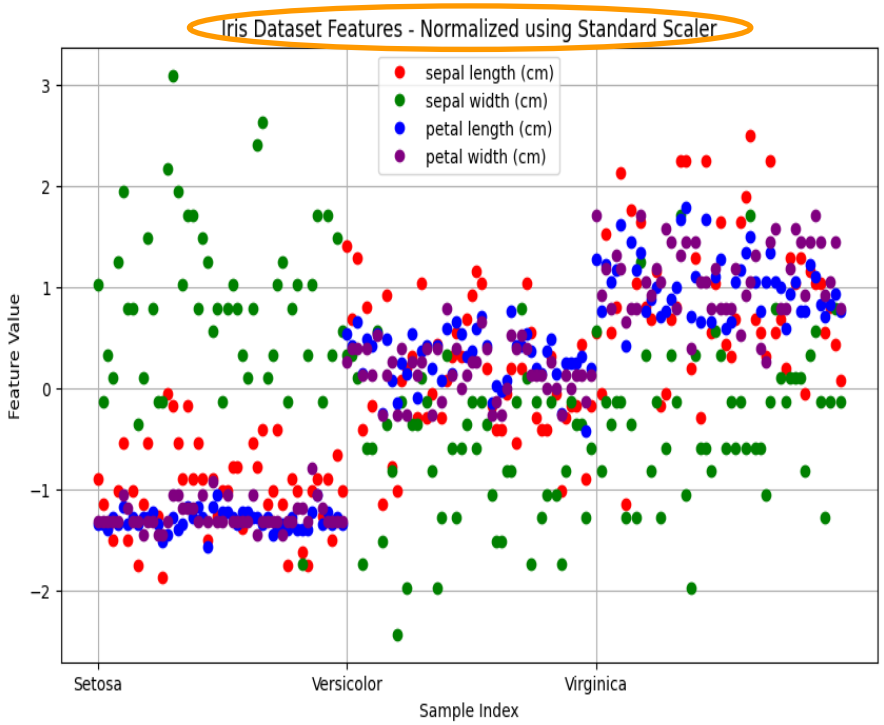
```
sepal length (cm): Mean: 5.84 Std_dev: 0.83
sepal width (cm):  Mean: 3.06 Std_dev: 0.43
petal length (cm): Mean: 3.76 Std_dev: 1.76
petal width (cm):  Mean: 1.20 Std_dev: 0.76
```






Iris Dataset Features
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
Feature Value
Setosa
Versicolor
Virginica
Sample Index


Iris Dataset Features - Normalized using MinMaxScaler
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
Feature Value
Setosa
Versicolor
Virginica
Sample Index


Iris Dataset Features - Normalized using Standard Scaler
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
Feature Value
Setosa
Versicolor
Virginica
Sample Index

Data set split

Train set: used to learn the parameters of the model

Val set (validation set): supervises the learning generality (identify overfitting);

Test set: used as a proxy for unseen data and evaluate our model on test-set (brand-new data set)

Now we will **split** the full data set in 2 subsets:

- **test_set** 10% of the entire data set (**15** examples)
- train_set 90% of the entire data set
  - Later on, when we will use .fit method to train our ANN model; from the train_set
    - 22.22% (**30** examples) will be used for **validation**
    - 77.78% (**105** examples) for real model **trainig**

Small / moderate data set:

- **70% / 20% /10%**

Data set split

```python
from sklearn.model_selection import train_test_split
# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(
    X_normalized, y, test_size=15, random_state=42  # Extract 15 samples (10%) for testing
)
```

X_train shape: (135, 4)
X_test shape: (15, 4)
y_train shape: (135,)
y_test shape: (15,)

labels for test set are:
[1 0 2 1 1 0 1 2 1 1 2 0 0 0 0]

ANN model

Secvential vs. Functional API

Sequential API

- **Structure:** Builds a linear stack of layers, where each layer has exactly one input and one output.
- **Simplicity:** Easy to use for simple models with a straightforward flow.
- **Limitations:** Not suitable for complex architectures like those with multiple inputs/outputs, shared layers, or non-linear connections.

Functional API

- **Structure:** Defines a computational graph where layers are connected like functions, allowing for flexible and complex topologies.
- **Flexibility:** Can build models with multiple inputs/outputs, shared layers, residual connections, and more.
- **Control:** Offers greater control over data flow and model structure.

ANN model

Secvential vs. Functional API

Key Differences

Feature	Sequential API	Functional API
Model Structure	Linear stack of layers	Flexible computational graph
Complexity	Simple models	Complex models
Flexibility	Limited	Highly flexible
Inputs/Outputs	Single input, single output	Multiple inputs/outputs
Layer Sharing	Not supported	Supported
Non-linear Connections	Not supported	Supported
Ease of Use	Easier for beginners	More advanced

When to Use Which

- **Sequential API:** Ideal for simple models with a sequential flow of layers, like basic classification or regression tasks.
- **Functional API:** Preferred for complex architectures, models with multiple inputs/outputs, shared layers, or when you need more control over the data flow and model structure (dynamic architecture – conditions, loops).

ANN model Secvential

How it Works:

1. **Create a Sequential Model:** You start by creating an instance of the keras.Sequential class.

2. **Add Layers:** You add layers to the model using the add() method. Each layer you add is stacked on top of the previous one.

3. **Compile the Model:** You compile the model using compile(), specifying the optimizer, loss function, and metrics.

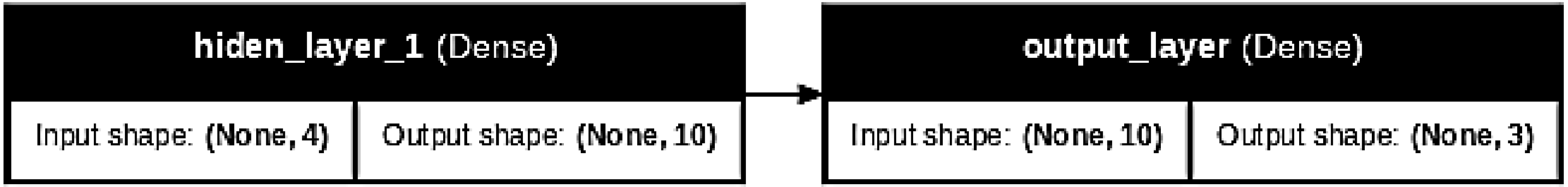
4. **Train the Model:** You train the model using fit(), providing the training data and labels.

5. **Make Predictions:** You use the trained model to make predictions on new data using predict().

ANN model

Secvential version 1

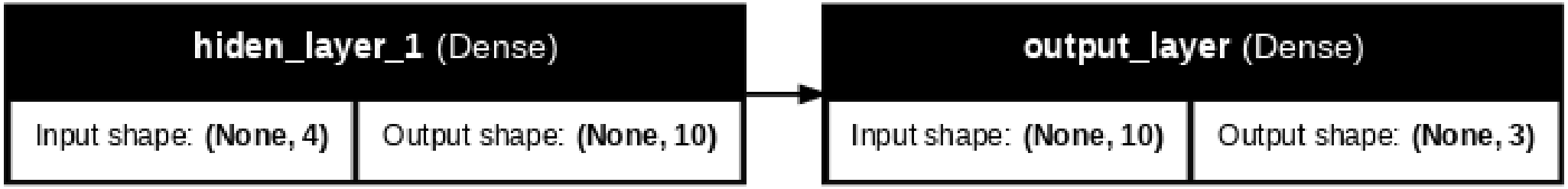
```python
## Version 1    Create a Sequential model incrementally via the add() method
from tensorflow import keras
model = keras.Sequential(name = 'My_ANN')
# Define the input shape using an Input layer as the first layer
# This specifies the shape of the input data (4,)
model.add(keras.layers.Input(shape=(4,), name = "input"))
# Add a 1st dense layer with 10 neurons and ReLU activation
# Dense layer performs a linear transformation followed by activation
model.add(keras.layers.Dense(10, activation='relu', name = 'hiden_layer_1'))
# Add the output layer with 3 neurons (one for each class) and softmax activation
# Softmax outputs probabilities for each class, summing to 1
model.add(keras.layers.Dense(3, activation='softmax', name = 'output_layer'))
```



ANN model

Secvential version 2

```python
## Version 2 Create a Sequential model
## by passing a list of layers to the Sequential constructor:
from tensorflow import keras
model = keras.Sequential([
    # Input shape
    keras.layers.Input(shape=(4,), name = "input"),
    # Hidden layer with 10 neurons and ReLU activation
    keras.layers.Dense(10, activation='relu', name = 'hiden_layer_1'),
    # Output layer with 3 neurons (for 3 classes) and softmax activation
    keras.layers.Dense(3, activation='softmax', name = 'output_layer')
])
```



ANN model

Secvential

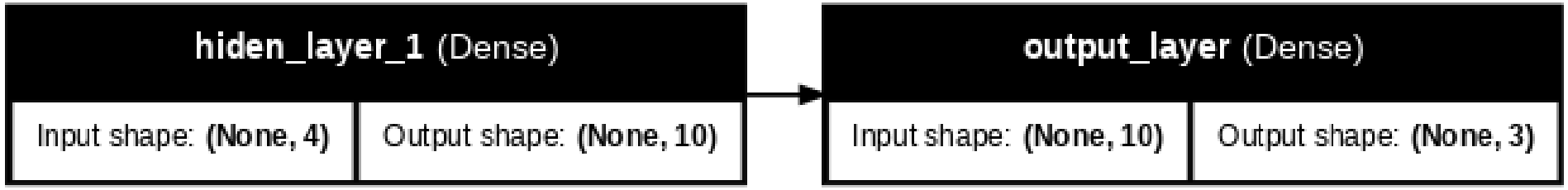
```
model.summary()
```

Model: "My_ANN"

Layer (type)	Output Shape	Param #
hiden_layer_1 (Dense)	(None, 10)	50
output_layer (Dense)	(None, 3)	33

Total params: 83 (332.00 B)
Trainable params: 83 (332.00 B)
Non-trainable params: 0 (0.00 B)

```
from tensorflow import keras
from keras import utils
utils.plot_model(model, to_file='model_diagram.png', show_shapes=True,
    show_layer_names=True, dpi=64, rankdir='LR')  # Adjust dpi and rankdir
```



ANN model

Functional API

1. **Define Input Layers:** You start by defining input tensors using keras.Input(), specifying the shape and data type of the input data.

2. **Create Layers as Functions:** Each Keras layer can be treated as a callable function. You apply a layer to an input tensor to produce an output tensor, like this:

   output_tensor = layer(input_tensor).

3. **Connect Layers:** You connect layers by passing the output tensor of one layer as the input tensor to the next layer, creating a computational graph.

4. **Define the Model:** You create a keras.Model instance, specifying the input tensors and output tensors of the model.

5. **Compile the Model:** You compile the model using compile(), specifying the optimizer, loss function, and metrics.

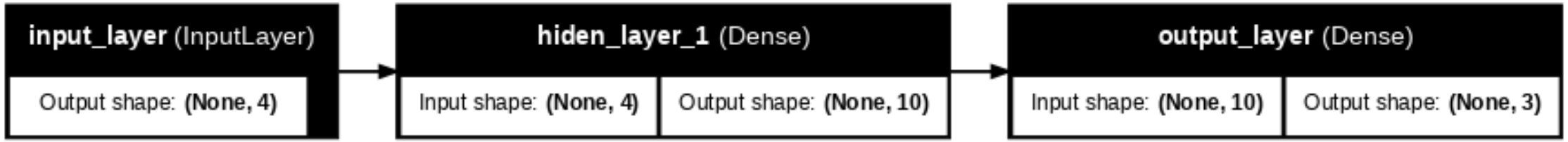
6. **Train the Model:** You train the model using fit(), providing the training data and labels.

7. **Make Predictions:** You use the trained model to make predictions on new data using predict().

ANN model

Functional API

```
## Version 3   With the "Functional API"
## You start from Input, you chain layer calls to specify the model's forward pass,
## and finally you create your model from inputs and outputs
from tensorflow import keras
# Input shape
inputs = keras.Input(shape=(4,), name = "input_layer")
# 1st layer as a function; input - "inputs",  output - "x"
x = keras.layers.Dense(10, activation="relu", name = 'hiden_layer_1')(inputs)
# 2nd layer as a function; input - "x",  output - "outputs"
outputs = keras.layers.Dense(3, activation="softmax", name = 'output_layer')(x)
# Create the model from input to output
model = keras.Model(inputs=inputs, outputs=outputs)
```



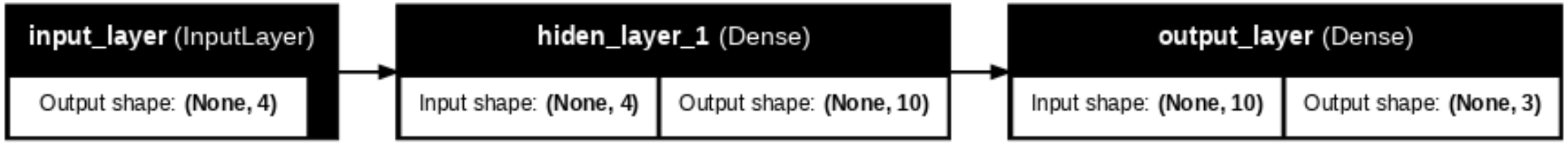
ANN model Functional API

```
model.summary()
```

Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 4)	0
hiden_layer_1 (Dense)	(None, 10)	50
output_layer (Dense)	(None, 3)	33

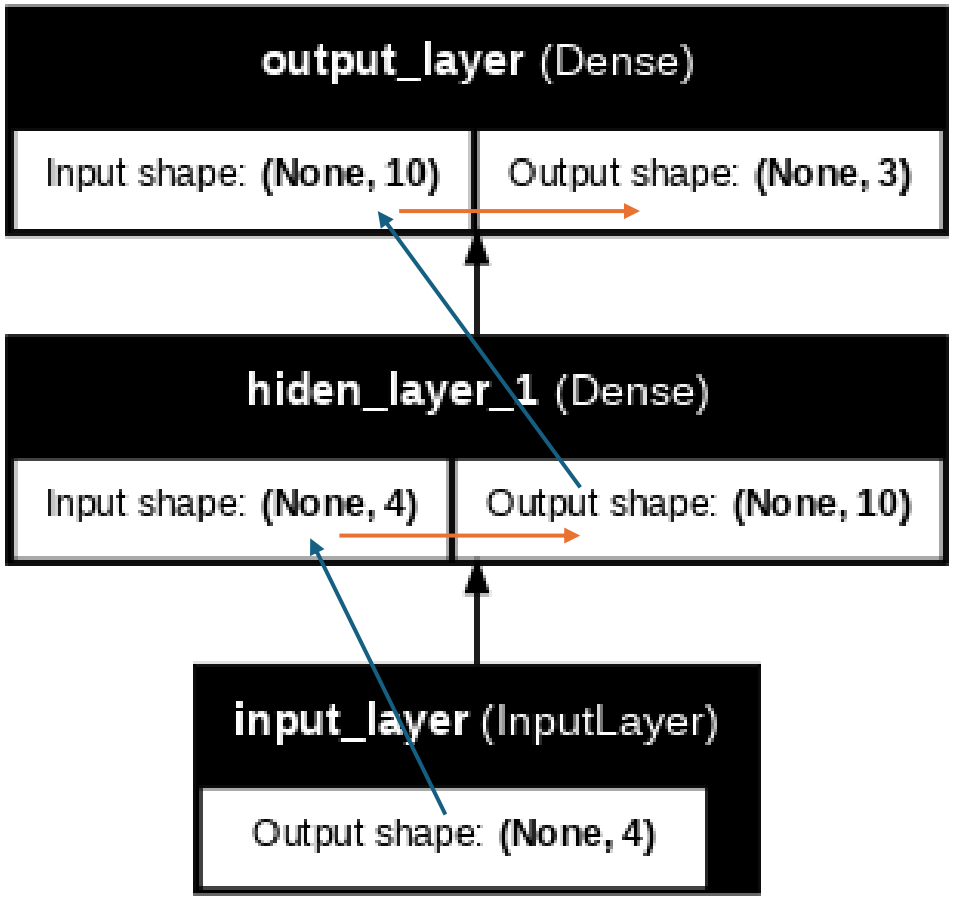
Total params: 83 (332.00 B)
Trainable params: 83 (332.00 B)
Non-trainable params: 0 (0.00 B)

```
from tensorflow import keras
from keras import utils
utils.plot_model(model, to_file='model_diagram.png', show_shapes=True,
    show_layer_names=True, dpi=64, rankdir='LR')  # Adjust dpi and rankdir
```



ANN model Functional API

```python
from tensorflow import keras
from keras import utils
utils.plot_model(model, to_file='model_diagram.png', show_shapes=True,
    show_layer_names=True, dpi=64, rankdir='BT')  # Adjust dpi and rankdir
```



ANN model Methods

The **tf.keras.Model** class features built-in training and evaluation **methods**:

- tf.keras.Model.**fit**: Trains the model for a fixed number of epochs.
- tf.keras.Model.**predict**: Generates output predictions for the input samples.
- tf.keras.Model.**evaluate**: Returns the loss and metrics values for the model; configured via the tf.keras.Model.compile method.

These methods give you access to the following built-in training features:

- Callbacks. You can leverage built-in callbacks for early stopping, model checkpointing, and TensorBoard monitoring. You can also implement custom callbacks.
- Distributed training. You can easily scale up your training to multiple GPUs, TPUs, or devices.
- Step fusing. With the steps_per_execution argument in tf.keras.Model.compile, you can process multiple batches in a single tf.function call, which greatly improves device utilization on TPUs.

For a detailed overview of how to use fit, see the training and evaluation guide. To learn how to customize the built-in training and evaluation loops, see Customizing what happens in fit().

[https://www.tensorflow.org/guide/keras]

Configure (compile) the ANN - loss function

sparse_categorical_crossentropy vs. binary_crossentropy

sparse_categorical_crossentropy

- **When to use:** This loss function is used for **multi-class classification problems** where the target labels are **integers representing the classes**.
- **Example:** In Iris dataset example, there are 3 classes (Setosa, Versicolor, Virginica), and the target labels are encoded as 0, 1, and 2 respectively.
- **How it works:** It calculates the cross-entropy loss between the true labels and the predicted probabilities for each class. The model outputs a probability distribution over the classes, and the loss function penalizes the model if the predicted probabilities don't align with the true class label.

binary_crossentropy

- **When to use:** This loss function is primarily used for **binary classification problems** where the target labels are either 0 or 1.
- **Example:** Classifying images as either "cat" or "dog" is a binary classification problem. The target labels would be 0 for "cat" and 1 for "dog".
- **How it works:** It calculates the cross-entropy loss between the true label (0 or 1) and the predicted probability of the positive class (usually class 1). The model outputs a single probability value, and the loss function penalizes the model if this probability doesn't match the true label.

Configure (compile) the ANN - loss function

sparse_categorical_crossentropy vs. binary_crossentropy

Key Differences

1. Number of classes: sparse_categorical_crossentropy is for multi-class problems (more than 2 classes), while binary_crossentropy is for binary problems (2 classes).

2. Target label format: sparse_categorical_crossentropy expects integer labels, whereas binary_crossentropy expects labels to be either 0 or 1.

3. Output layer activation: With sparse_categorical_crossentropy, you typically use a softmax activation in the output layer to produce a probability distribution over the classes. With binary_crossentropy, you usually use a sigmoid activation to produce a single probability value.

Configure (compile) the ANN

Configures the learning process for the ANN model by specifying

- how it will measure its performance (**loss function**)
- how it will **update** its weights (**optimizer**)
- what **metrics** it will track during training

loss='sparse_categorical_crossentropy' - specifies the loss function to be used during training.

- sparse_categorical_crossentropy' is suitable for multi-class classification where the target labels are integers representing the classes (0, 1, 2, etc.).
- It calculates the cross-entropy loss between the true labels and the predicted probabilities.

optimizer='Adam'- specifies the optimization algorithm to use during training.

- 'Adam' is a popular adaptive learning rate optimization algorithm that adjusts the learning rate for each parameter based on its past gradients.

metrics=['accuracy'] - Specifies the metrics to be evaluated during training and testing.

- 'accuracy' is a common metric for classification, representing the percentage of correctly classified samples.

Configure (compile) the ANN

```
Model.compile(
    optimizer="rmsprop",
    loss=None,
    loss_weights=None,
    metrics=None,
    weighted_metrics=None,
    run_eagerly=False,
    steps_per_execution=1,
    jit_compile="auto",
    auto_scale_loss=True,
)
```

```
from tensorflow import keras
from keras import optimizers

model.compile(
    optimizer  =keras.optimizers.Adam(learning_rate=3e-3),
    loss = 'sparse_categorical_crossentropy',
    metrics = [
        "accuracy"
    ]
)
```

Train the ANN model

In essence, **model.fit()** is where the actual learning happens for your neural network. Think of it as the process of teaching your model to recognize patterns and make accurate predictions based on the data you provide.

1. Data Input (X_train, y_train): You provide the training data to the fit() function.

X_train represents the input features (sepal length, sepal width, petal length, petal width in your case), and y_train represents the corresponding target labels (species of Iris). This is the information your model will learn from.

2. Epochs (epochs=600): An epoch refers to one complete pass through the entire training dataset. You're specifying that the model should iterate over the data 600 times. This allows the model to gradually adjust its weights and improve its predictions.

3. Verbose (verbose=1): This parameter controls the amount of output displayed during training. A value of 1 means you'll see progress updates during each epoch.

4. Batch Size (batch_size=32): Instead of updating the model's weights after every single training sample, you're using batches of 32 samples. This helps make the training process more efficient and stable. The model calculates the average loss for a batch and updates its weights accordingly.

Train the ANN model

5. Validation Split (validation_split=0.2): You're setting aside 20% of your training data as a validation set. This subset is used to monitor the model's performance during training. It helps you detect overfitting, which is when the model starts to memorize the training data too well and performs poorly on unseen data.

6.Training Process:

- The model iterates through the training data in batches.
- For each batch, it calculates the loss using the specified loss function.
- It then uses the optimizer (Adam) to update the weights of the model to minimize the loss.
- This process is repeated for the specified number of epochs.

7.Validation:

- After each epoch, the model's performance is evaluated on the validation set.
- This helps you track how well the model is generalizing to unseen data.
- If the validation loss starts increasing while the training loss continues to decrease, it might be a sign of overfitting.

8.Output (hist):

- The fit() function returns a history object (hist in your code) that contains information about the training process, such as the loss and accuracy values for each epoch.
- You can use this history object to plot graphs and analyze the model's learning progress.

Train the ANN model

```
Model.fit(
    x=None,
    y=None,
    batch_size=None,
    epochs=1,
    verbose="auto",
    callbacks=None,
    validation_split=0.0,
    validation_data=None,
    shuffle=True,
    class_weight=None,
    sample_weight=None,
    initial_epoch=0,
    steps_per_epoch=None,
    validation_steps=None,
    validation_batch_size=None,
    validation_freq=1,
)
```

Train the ANN model

```python
hist = model.fit(
    X_train,                # The training data (input features).
    y_train,                # The target labels for the training data.
    epochs=600,             # The number of training epochs (iterations over the entire dataset).
                            # One epoch means that the model has seen and learned
                            # from all the training samples
    verbose=1,              # Controls the verbosity of the training output.
    batch_size=32,          # The number of samples per gradient update.
                            # Experimenting with different batch sizes is recommended
                            # to find the best value for your specific case.
    validation_split=0.2    # The fraction of the training data to be used as validation data.
                            # In this case, 20% of the training data will be used for validation.
                            # The model's performance will be evaluated on this validation set
                            # during training to monitor for overfitting and
                            # to get an estimate of how well the model generalizes to unseen data.
)
```

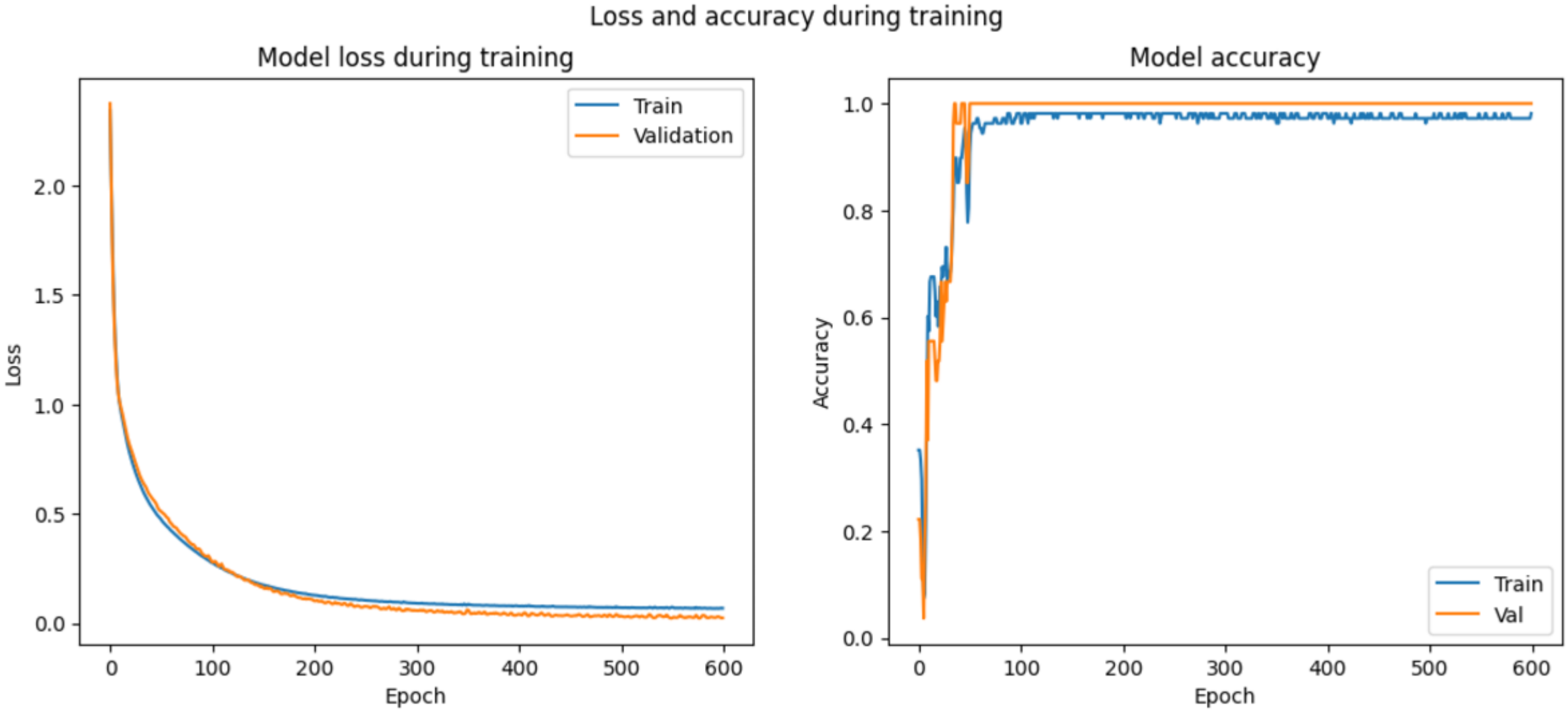
Train the ANN model

```
Epoch 1/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 3s 303ms/step - accuracy: 0.3845 - loss: 2.2734 - val_accuracy: 0.2222 - val_loss: 2.3747
Epoch 2/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 43ms/step - accuracy: 0.3543 - loss: 2.0709 - val_accuracy: 0.2222 - val_loss: 2.0468
Epoch 3/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 26ms/step - accuracy: 0.2958 - loss: 1.9306 - val_accuracy: 0.1852 - val_loss: 1.7680
Epoch 4/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 48ms/step - accuracy: 0.2779 - loss: 1.6472 - val_accuracy: 0.1111 - val_loss: 1.5479
Epoch 5/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 28ms/step - accuracy: 0.1807 - loss: 1.3974 - val_accuracy: 0.1111 - val_loss: 1.3899
```

```
Epoch 596/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9764 - loss: 0.0680 - val_accuracy: 1.0000 - val_loss: 0.0307
Epoch 597/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9639 - loss: 0.0726 - val_accuracy: 1.0000 - val_loss: 0.0285
Epoch 598/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 16ms/step - accuracy: 0.9701 - loss: 0.0693 - val_accuracy: 1.0000 - val_loss: 0.0256
Epoch 599/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 13ms/step - accuracy: 0.9795 - loss: 0.0594 - val_accuracy: 1.0000 - val_loss: 0.0244
Epoch 600/600
4/4 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.9884 - loss: 0.0644 - val_accuracy: 1.0000 - val_loss: 0.0251
```

Analyze the learning progress

(learning rate = 0.003)



Train Loss = 0.0692
Validation Loss = 0.0251
Test_loss = 0.0811

Train Accuracy = 0.9815
Validation Accuracy = 1.0000
Test_accuracy = 0.9333

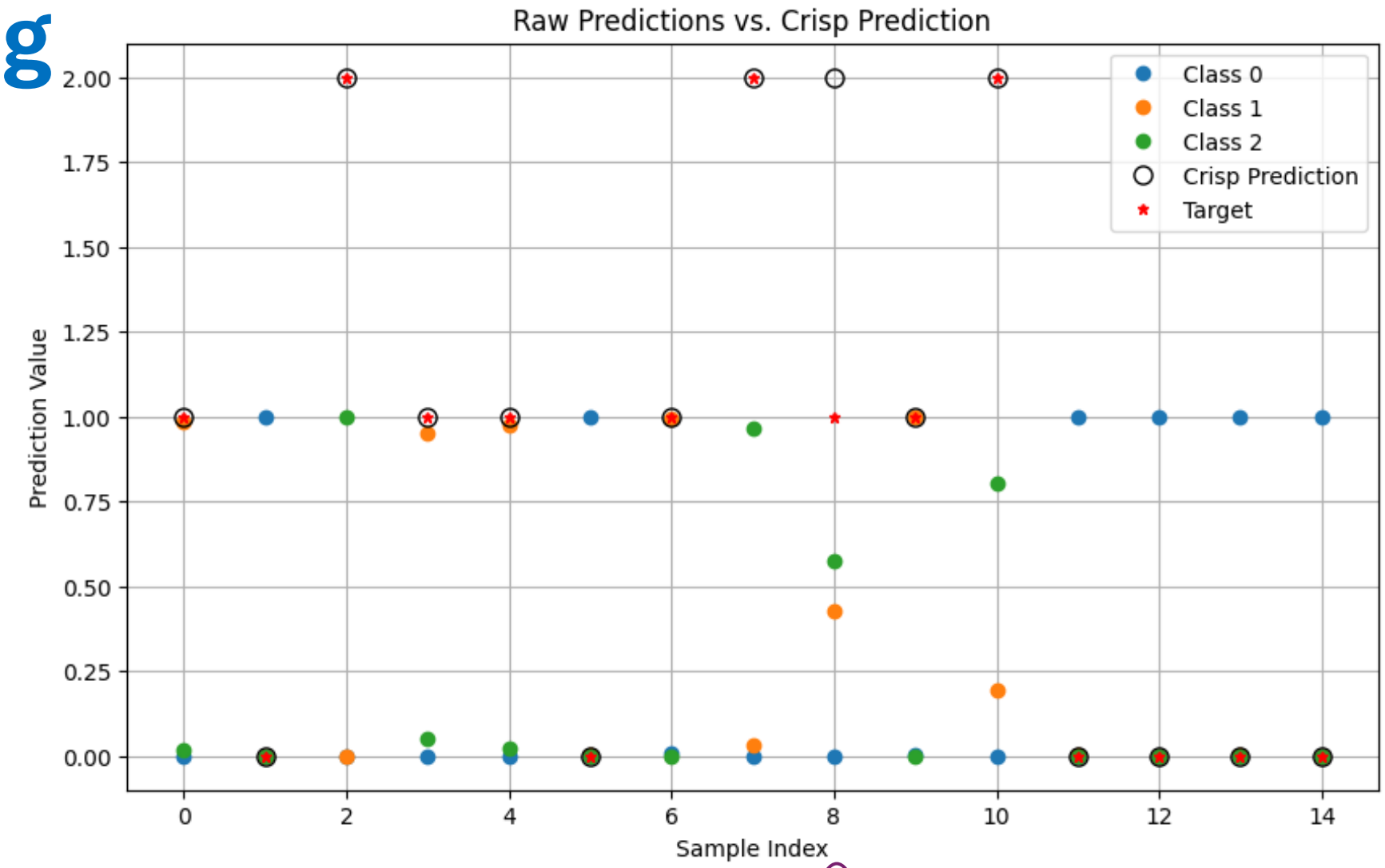
Evaluate the ANN model

```
test_loss, test_accuracy = model.evaluate(
    X_test,
    y_test,
    verbose = 0)
# Extract loss and accuracy values from the training history
train_loss = hist.history['loss'][-1]
train_accuracy = hist.history['accuracy'][-1]
val_loss = hist.history['val_loss'][-1]
val_accuracy = hist.history['val_accuracy'][-1]
```

Understanding predictions

Raw predictions:

```
[[0.      0.983   0.016]
 [0.999   0.001   0.    ]
 [0.      0.001   0.999]
 [0.      0.95    0.05 ]
 [0.      0.976   0.024]
 [0.998   0.002   0.    ]
 [0.008   0.992   0.    ]
 [0.      0.032   0.968]
 [0.      0.426   0.574]
 [0.003   0.996   0.001]
 [0.      0.195   0.804]
 [0.998   0.002   0.    ]
 [1.      0.      0.    ]
 [0.998   0.002   0.    ]
 [1.      0.      0.    ]]
```



Predicted [1 0 2 1 1 0 1 2 2 1 2 0 0 0 0]

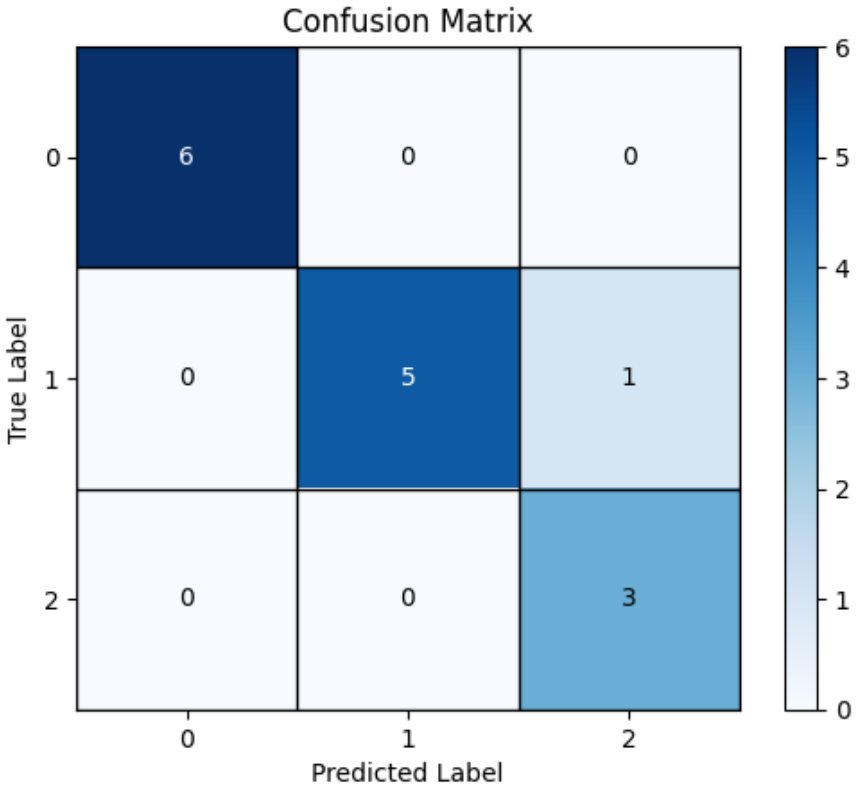
Target [1 0 2 1 1 0 1 2 1 1 2 0 0 0 0]

ANN model in prediction mode

```python
# Make predictions on the test examples
predictions = model.predict(X_test)
# Format predictions to 3 decimal places
formatted_predictions = np.around(predictions,decimals=3)
# Print the formatted predictions
print("\nRaw predictions, for each example:\n")
print(formatted_predictions)
# Get the predicted class labels
# (argmax finds the index of the maximum probability)
predicted_labels = predictions.argmax(axis=1)
# Print the predicted labels vs target labels
print("\nPredicted Labels:", predicted_labels)
print("Target Labels:   ", y_test)
```

(learning rate = 0.003)

Test data



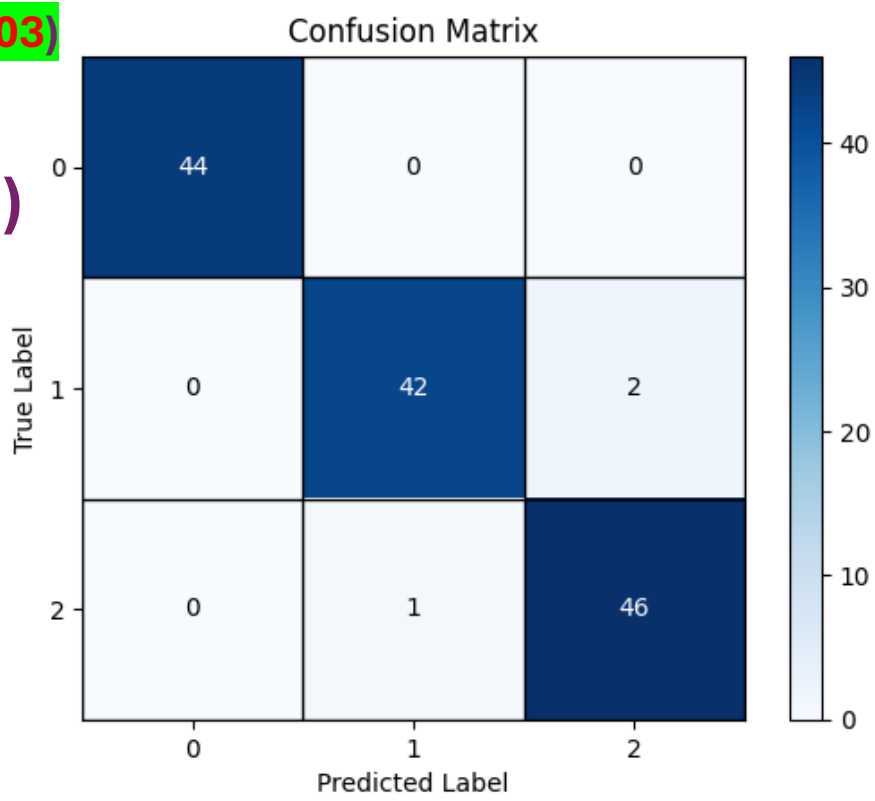

```
>>  Metrics classification report:

              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       1.00      0.83      0.91         6
           2       0.75      1.00      0.86         3

    accuracy                           0.93        15
   macro avg       0.92      0.94      0.92        15
weighted avg       0.95      0.93      0.94        15
```

Train (+ validation) data



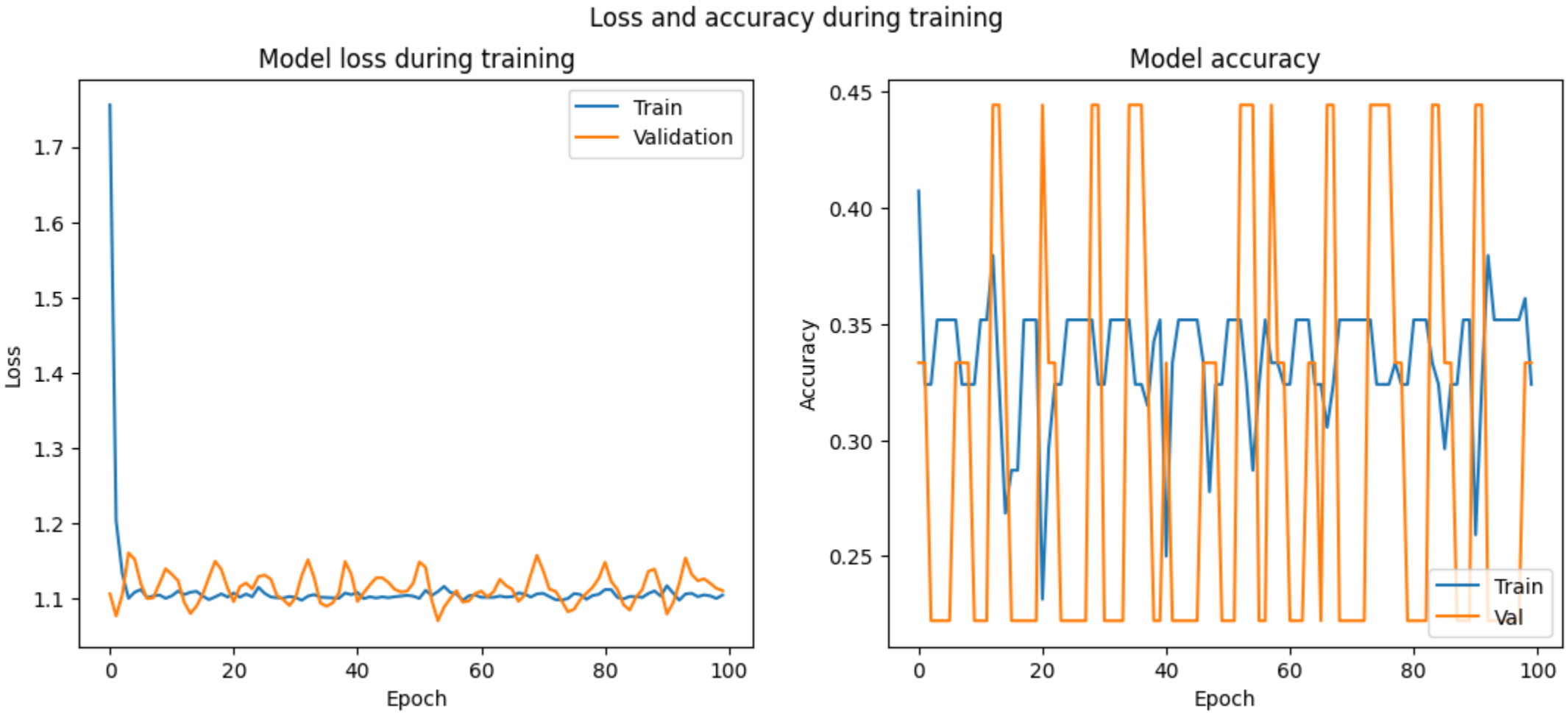

```
>>  Metrics classification report:

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        44
           1       0.98      0.95      0.97        44
           2       0.96      0.98      0.97        47

    accuracy                           0.98       135
   macro avg       0.98      0.98      0.98       135
weighted avg       0.98      0.98      0.98       135
```

Analyze the learning progress (learning rate = 0.2)

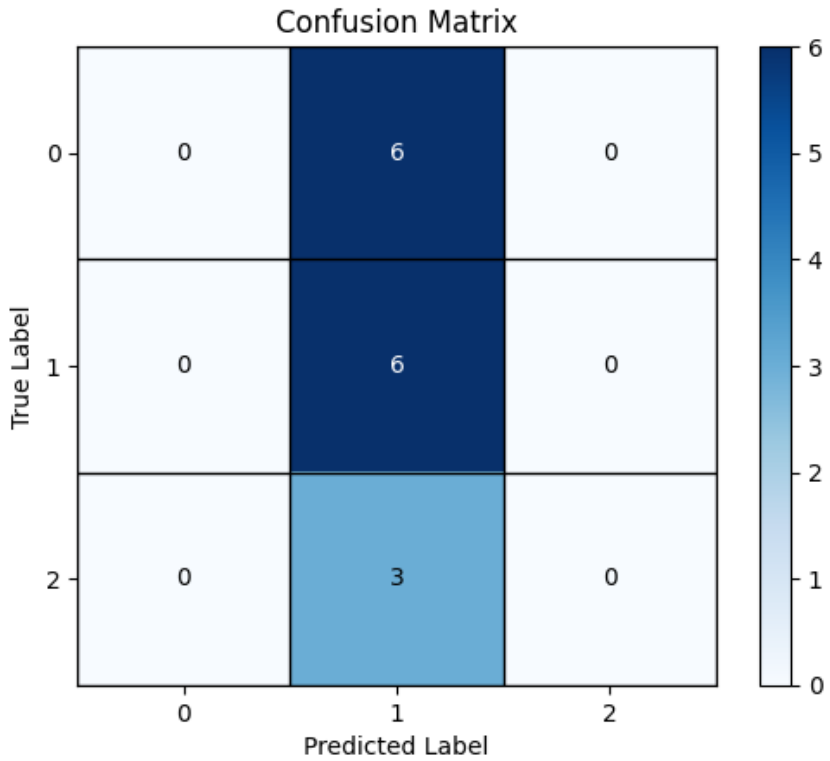



```
Train Loss =          1.1045
Validation Loss =     1.1104
Test_loss =           1.0838
```

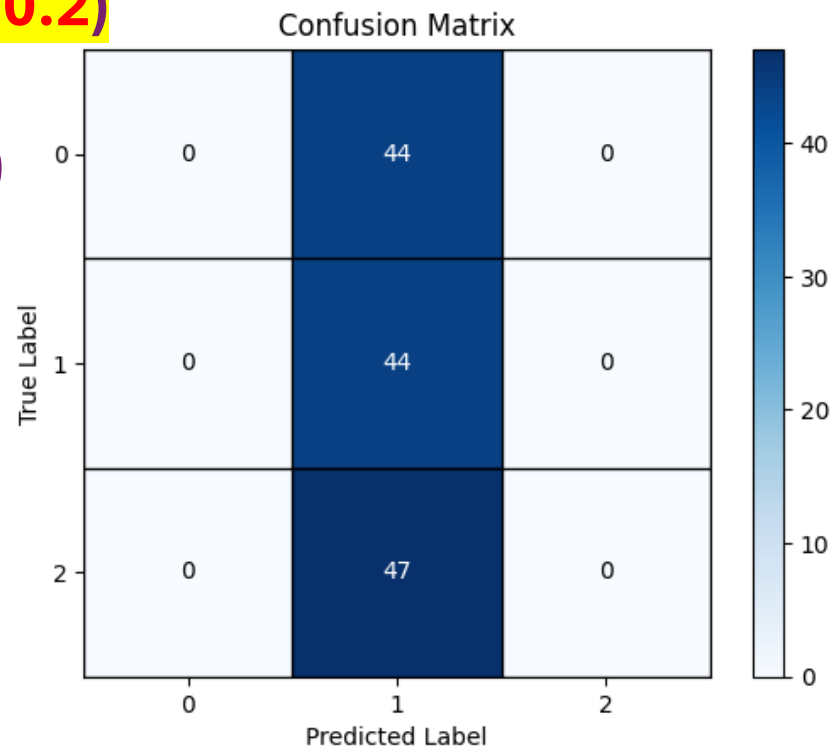
```
Train Accuracy =          0.3241
Validation Accuracy =     0.3333
Test_accuracy =           0.4000
```

(learning rate = 0.2)

Test data



Train (+ validation) data




```
>>  Metrics classification report:

              precision    recall  f1-score   support

           0       0.00      0.00      0.00         6
           1       0.40      1.00      0.57         6
           2       0.00      0.00      0.00         3

    accuracy                           0.40        15
   macro avg       0.13      0.33      0.19        15
weighted avg       0.16      0.40      0.23        15
```

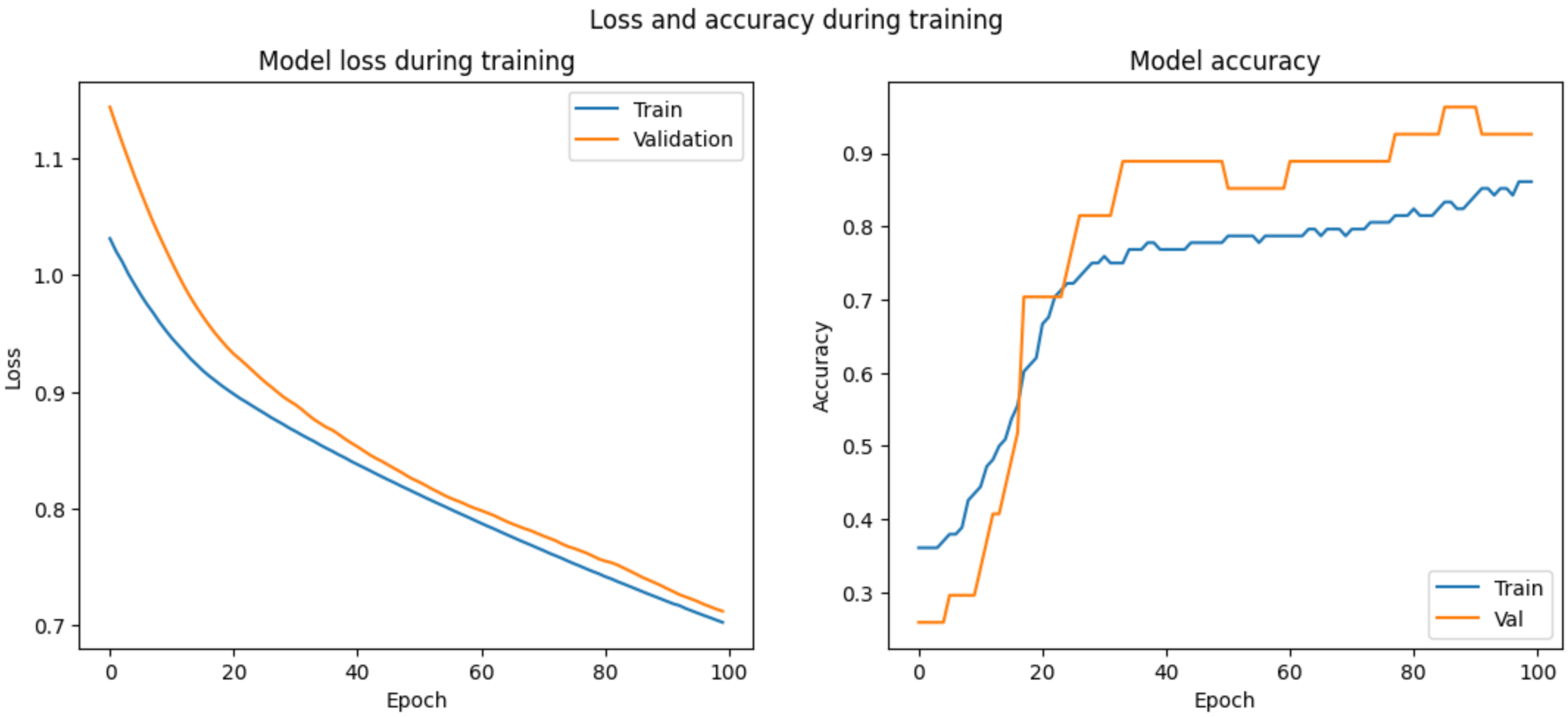
```
>>  Metrics classification report:

              precision    recall  f1-score   support

           0       0.00      0.00      0.00        44
           1       0.33      1.00      0.49        44
           2       0.00      0.00      0.00        47

    accuracy                           0.33       135
   macro avg       0.11      0.33      0.16       135
weighted avg       0.11      0.33      0.16       135
```

Evaluate the ANN model (learning rate = 0.0002)

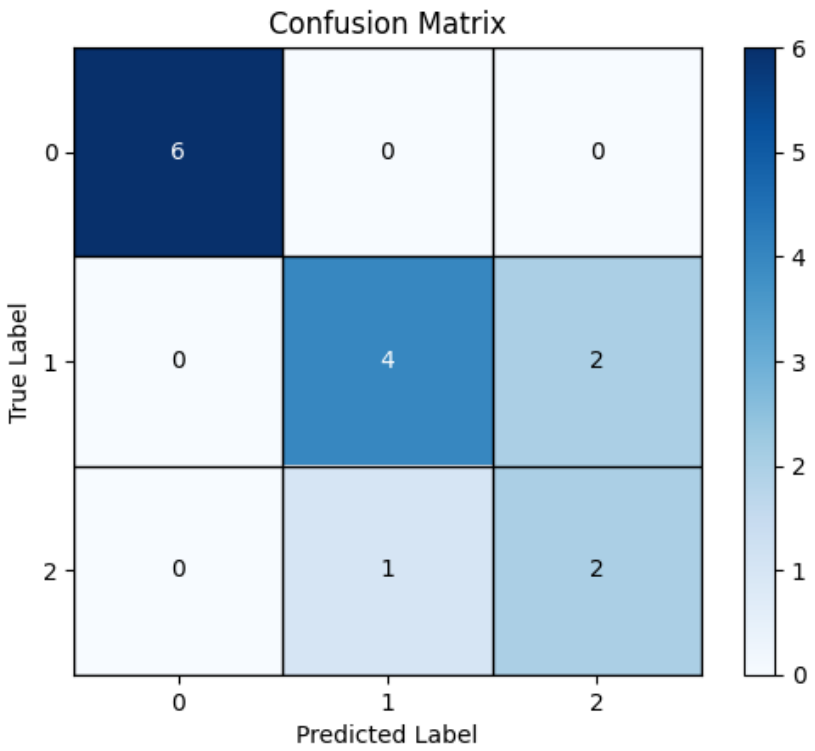



```
Train Loss =          0.7026
Validation Loss =     0.7122
Test_loss =           0.6960
```

```
Train Accuracy =          0.8611
Validation Accuracy =     0.9259
Test_accuracy =           0.8000
```

(learning rate = 0.0002)

Test data



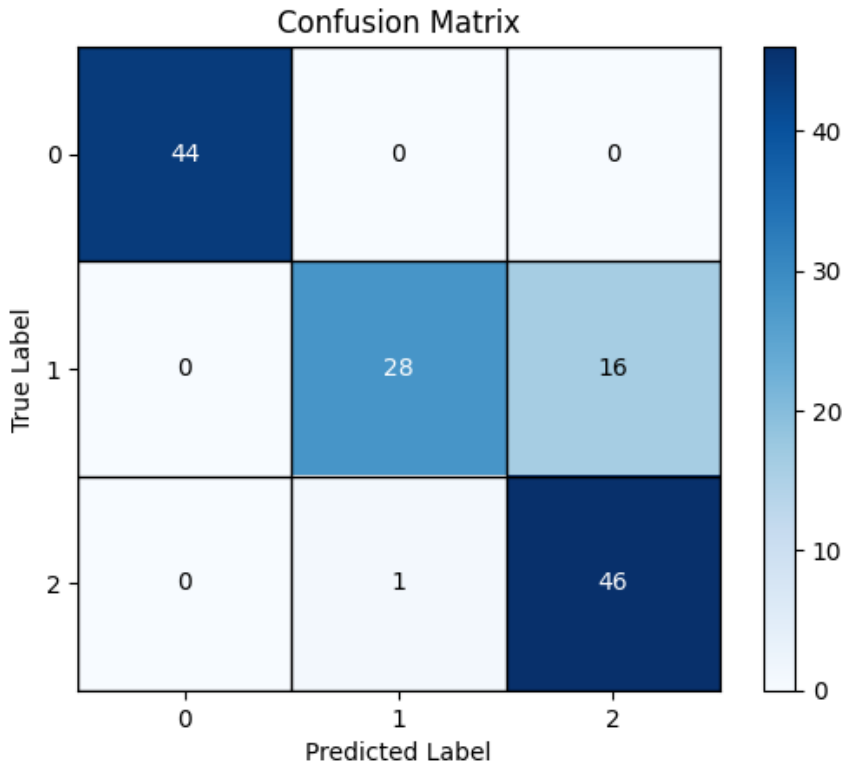

```
>>  Metrics classification report:

              precision    recall  f1-score   support

           0       1.00      1.00      1.00         6
           1       0.80      0.67      0.73         6
           2       0.50      0.67      0.57         3

    accuracy                           0.80        15
   macro avg       0.77      0.78      0.77        15
weighted avg       0.82      0.80      0.81        15
```

Train (+ validation) data




```
>>  Metrics classification report:

              precision    recall  f1-score   support

           0       1.00      1.00      1.00        44
           1       0.97      0.64      0.77        44
           2       0.74      0.98      0.84        47

    accuracy                           0.87       135
   macro avg       0.90      0.87      0.87       135
weighted avg       0.90      0.87      0.87       135
```

Save the ANN model

```
# save with .keras extension
model.save('ANN_iris.keras')
# # save with .h5 extension  % old version, not to use anymore
# model.save('ANN_iris.h5')
```

Purpose of Saving

- **Persistence:** Saving the model allows you to preserve its learned weights and architecture.
- **Reusability:** You can load the saved model later to avoid retraining, saving time and resources.
- **Sharing:** You can share your trained model with others, enabling them to use it without training from scratch.

How it Works

- **model.save()**: This function is a core part of Keras (and TensorFlow) and handles the process of saving your model.
- **'ANN_iris.keras'**: This argument specifies the file path and name where the model will be saved.
  - **Using the .keras extension is recommended for saving models**, although .h5 can also be used for legacy compatibility.

Save the ANN model

Benefits of .keras Format

- **Self-Contained:** The .keras format saves your model's architecture, weights, training configuration, and even the optimizer state. It's essentially a complete snapshot of your model.
- **Human-Readable:** The model architecture is saved in a JSON-like format, making it relatively easy to understand and modify.
- **Easy Loading:** Loading a model saved in this format is straightforward, as you'll see in the model.load_model() function in the code.

What happens when you save

- A directory named ANN_iris.keras will be created in your Colab environment's current working directory.
- This directory will contain the necessary files, including the model's architecture, weights, and configuration, to allow you to load and reuse it later.
- This method provides a reliable and straightforward approach to saving your trained model, ensuring you can reuse it for predictions and share it efficiently.

ANN model in JSON format

Load the ANN model

```
loaded_model = keras.models.load_model('ANN_iris.keras')
```

Purpose of Loading

- **Reuse:** Loading allows you to bring your previously saved model back into memory and make predictions or perform other tasks without retraining.
- **Efficiency:** Saves the time and resources that would be required for retraining a model from scratch.
- **Sharing:** Enables others to use a trained model that you've shared with them.

How it Works

- **keras.models.load_model():** This function from the Keras API is specifically designed to load saved models.
- **'ANN_iris.keras'**: This argument provides the file path to the model you want to load. This should match the name you used when saving the model.

Load the ANN model

What happens when you load

- The load_model() function reads the saved model files (ANN_iris.keras in this case).
- It reconstructs the model's architecture based on the saved configuration.
- It loads the trained weights into the model's layers.
- The loaded model is assigned to the variable loaded_model, which can now be used for prediction or any other operations.

Benefits:

- **Seamless Continuation:** You can seamlessly pick up where you left off with your model.
- **Reduced Training Time:** No need to retrain your model, which can be computationally expensive, especially for complex architectures.
- **Model Sharing:** Makes it easy to share models within your team or with the wider community.
- **Using the Loaded Model:** After loading, the loaded_model object is a fully functional Keras model, identical to the one you trained and saved.