# Experiment No.: 3
# Title: Deep Neural Network

**Batch: HO-DL 1** **Roll No.: 16010422234** **Experiment No. 03**

---

**Aim:** To build deep neural networks capable of learning the complex kinds of relationships

---

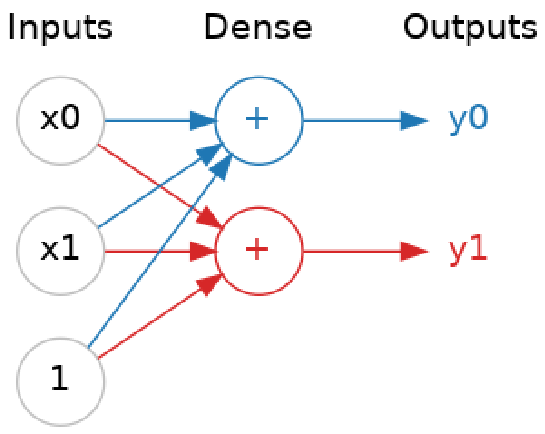
**Resources needed:** Python

---

**Theory:**

**Layers**

Neural networks typically organize their neurons into **layers**. When we collect together linear units having a common set of inputs we get a **dense** layer.



A dense layer of two linear units receiving two inputs and a bias.

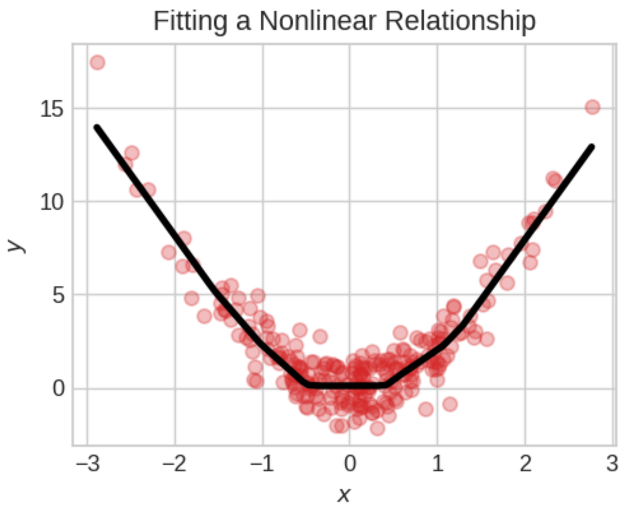
You could think of each layer in a neural network as performing some kind of relatively simple transformation. Through a deep stack of layers, a neural network can transform its inputs in more and more complex ways. In a well-trained neural network, each layer is a transformation getting us a little bit closer to a solution.

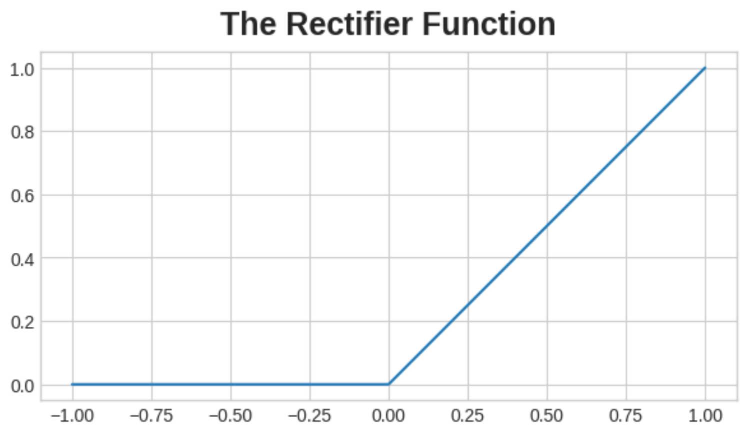
A "layer" in Keras is a very general kind of thing. A layer can be, essentially, any kind of *data transformation*. Many layers, like the convolutional and recurrent layers, transform data through use of neurons and differ primarily in the pattern of connections they form. Others though are used for feature engineering or just simple arithmetic.

**The Activation Function**

It turns out, however, that two dense layers with nothing in between are no better than a single dense layer by itself. Dense layers by themselves can never move us out of the world of lines and planes. What we need is something nonlinear. What we need are activation functions.

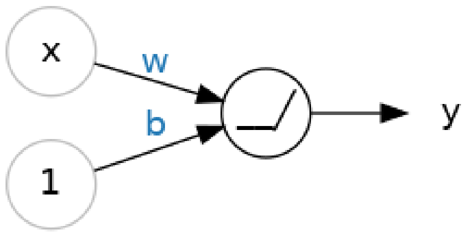
Without activation functions, neural networks can only learn linear relationships. In order to fit curves, we'll need to use activation functions. An activation function is simply some function we apply to each of a layer's outputs (its activations). The most common is the rectifier function max(0,x) .





The rectifier function has a graph that's a line with the negative part "rectified" to zero. Applying the function to the outputs of a neuron will put a bend in the data, moving us away from simple lines.

When we attach the rectifier to a linear unit, we get a rectified linear unit or ReLU. (For this reason, it's common to call the rectifier function the "ReLU function".) Applying a ReLU activation to a linear unit means the output becomes max(0, w * x + b), which we might draw in a diagram like:

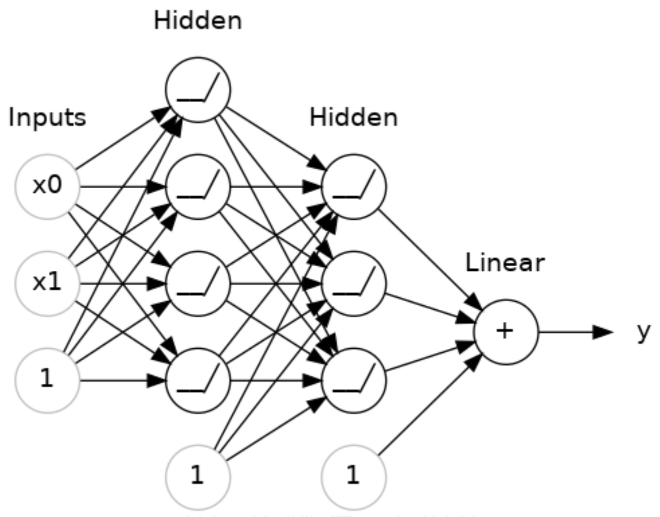


A rectified linear unit.

Diagram of a single ReLU. Like a linear unit, but instead of a '+' symbol we now have a hinge '_/'.

**Stacking Dense Layers**

Now that we have some nonlinearity, let's see how we can stack layers to get complex data transformations.



A stack of dense layers makes a "fully-connected" network.

The layers before the output layer are sometimes called hidden since we never see their outputs directly.

Now, notice that the final (output) layer is a linear unit (meaning, no activation function). That makes this network appropriate to a regression task, where we are trying to predict some arbitrary numeric value. Other tasks (like classification) might require an activation function on the output.

**Building Sequential Models**

The Sequential model we've been using will connect together a list of layers in order from first to last: the first layer gets the input, the last layer produces the output. This creates the model in the figure above:

**Activity:**
1. Download the required dataset.
2. Define the Input Shape.
3. Create a Model with some number of hidden layers and output layers.
4. Decide the values of hyperparameters.
5. Try to use different activation functions.
6. Analyze the effect of various values of hyperparameters.
7. Print the developed model.

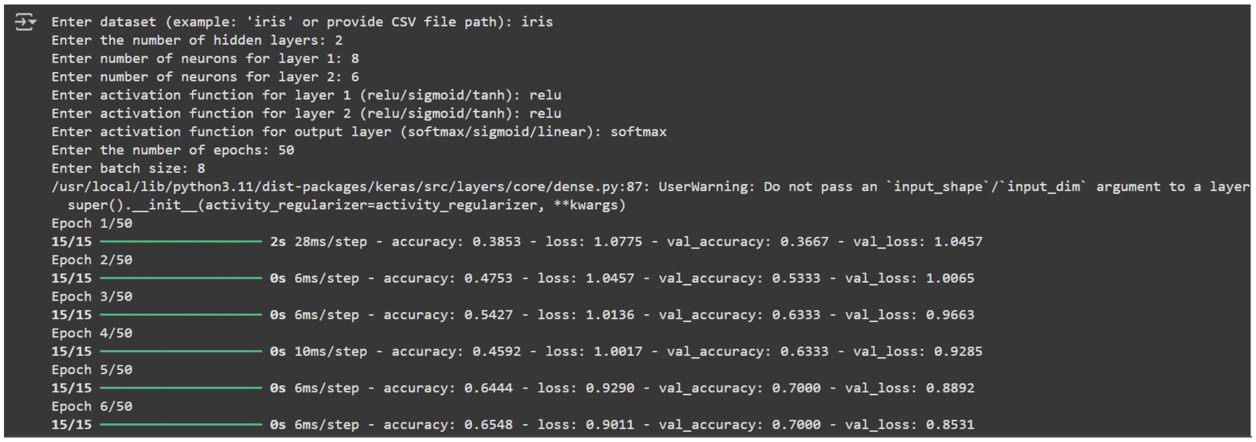
---

**Program:**

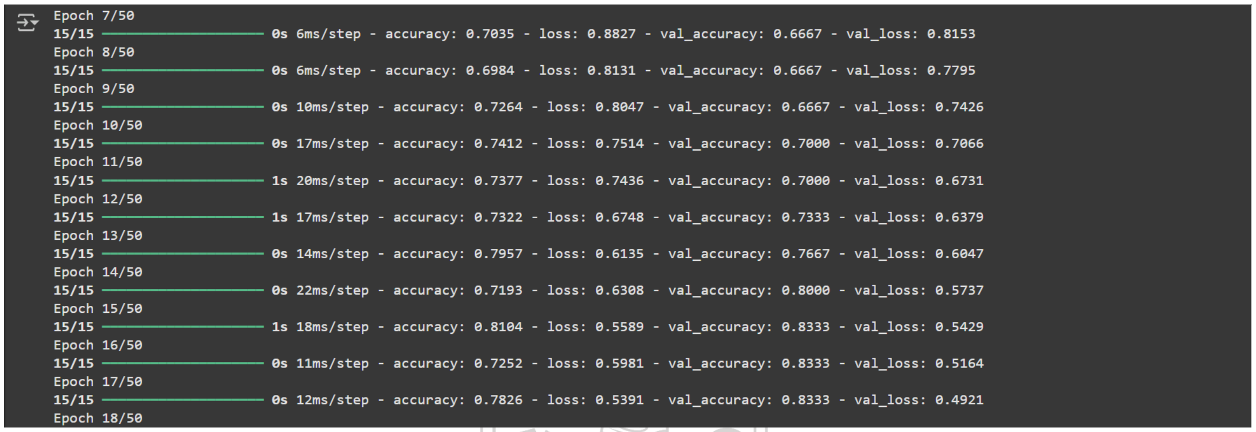
```python
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

# Step 1: Load Dataset
dataset_choice = input("Enter dataset (example: 'iris' or provide CSV file
path): ").strip()

if dataset_choice.lower() == "iris":
    from sklearn.datasets import load_iris
    iris = load_iris()
    X, y = iris.data, iris.target
else:
    df = pd.read_csv(dataset_choice)
    X = df.iloc[:, :-1].values  # Features
    y = df.iloc[:, -1].values   # Target

# Step 2: Data Preprocessing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Step 3: User Inputs for Model Parameters
```

```
num_layers = int(input("Enter the number of hidden layers: "))
num_neurons = [int(input(f"Enter number of neurons for layer {i+1}: "))
for i in range(num_layers)]
activation_functions = [input(f"Enter activation function for layer {i+1}
(relu/sigmoid/tanh): ") for i in range(num_layers)]
output_activation = input("Enter activation function for output layer
(softmax/sigmoid/linear): ")
epochs = int(input("Enter the number of epochs: "))
batch_size = int(input("Enter batch size: "))

# Step 4: Build Model
model = Sequential()
# Input Layer
model.add(Dense(num_neurons[0], activation=activation_functions[0],
input_shape=(X_train.shape[1],)))

# Hidden Layers
for i in range(1, num_layers):
    model.add(Dense(num_neurons[i], activation=activation_functions[i]))

# Output Layer
num_classes = len(set(y))
if num_classes > 2:
    model.add(Dense(num_classes, activation=output_activation))
else:
    model.add(Dense(1, activation=output_activation))

# Step 5: Compile Model
loss_function = "sparse_categorical_crossentropy" if num_classes > 2 else
"binary_crossentropy"
model.compile(optimizer='adam', loss=loss_function, metrics=['accuracy'])

# Step 6: Train Model
model.fit(X_train, y_train, epochs=epochs, batch_size=batch_size,
validation_data=(X_test, y_test))

# Step 7: Evaluate Model
test_loss, test_acc = model.evaluate(X_test, y_test)
print(f"\nTest Accuracy: {test_acc:.4f}")

# Step 8: Display Model Summary
model.summary()
```

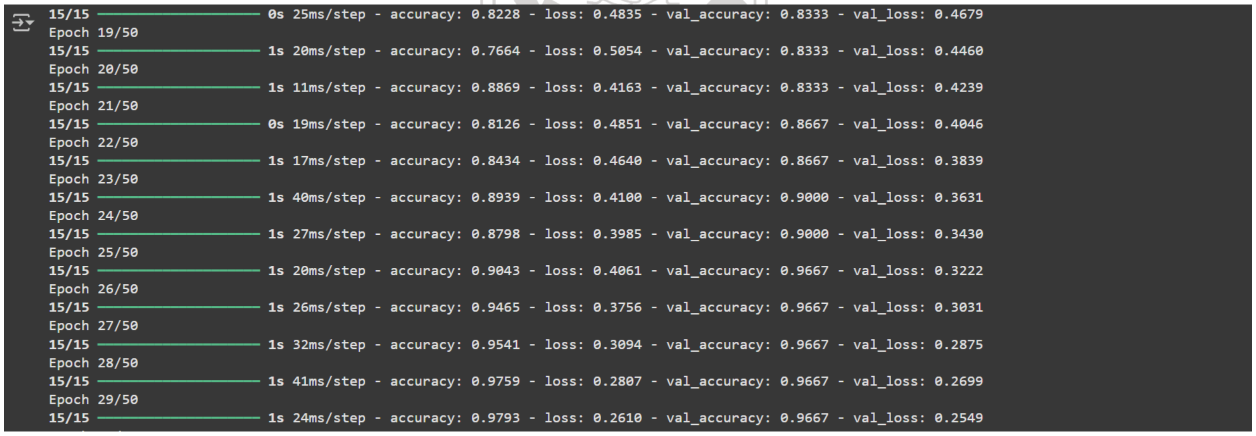
**Output:**



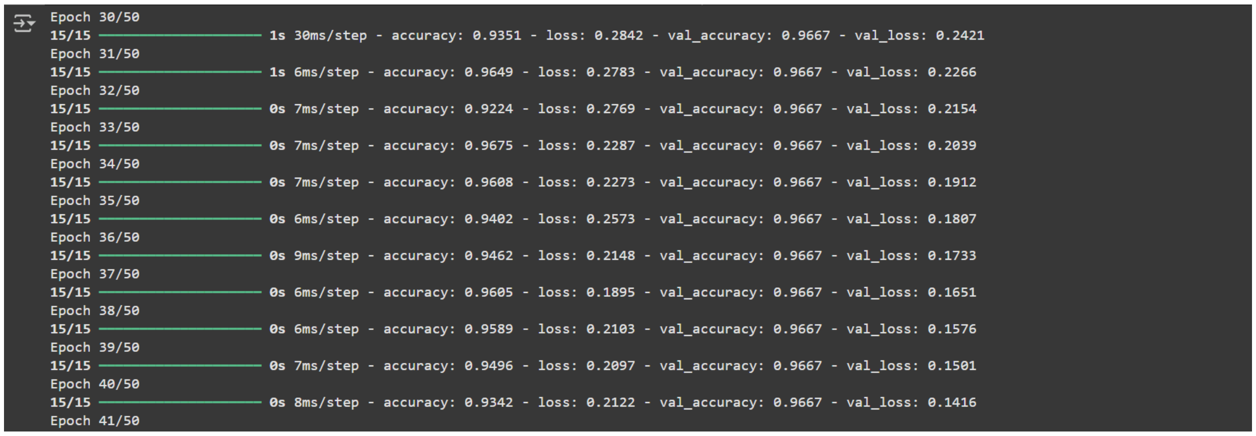
```
Enter dataset (example: 'iris' or provide CSV file path): iris
Enter the number of hidden layers: 2
Enter number of neurons for layer 1: 8
Enter number of neurons for layer 2: 6
Enter activation function for layer 1 (relu/sigmoid/tanh): relu
Enter activation function for layer 2 (relu/sigmoid/tanh): relu
Enter activation function for output layer (softmax/sigmoid/linear): softmax
Enter the number of epochs: 50
Enter batch size: 8
/usr/local/lib/python3.11/dist-packages/keras/src/layers/core/dense.py:87: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
Epoch 1/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 2s 28ms/step - accuracy: 0.3853 - loss: 1.0775 - val_accuracy: 0.3667 - val_loss: 1.0457
Epoch 2/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.4753 - loss: 1.0457 - val_accuracy: 0.5333 - val_loss: 1.0065
Epoch 3/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.5427 - loss: 1.0136 - val_accuracy: 0.6333 - val_loss: 0.9663
Epoch 4/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.4592 - loss: 1.0017 - val_accuracy: 0.6333 - val_loss: 0.9285
Epoch 5/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.6444 - loss: 0.9290 - val_accuracy: 0.7000 - val_loss: 0.8892
Epoch 6/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.6548 - loss: 0.9011 - val_accuracy: 0.7000 - val_loss: 0.8531
```



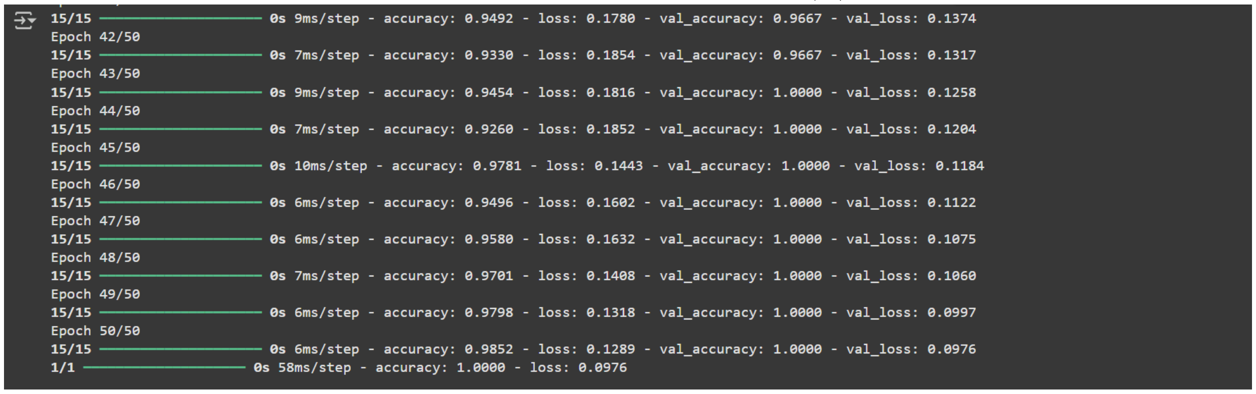
```
Epoch 7/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.7035 - loss: 0.8827 - val_accuracy: 0.6667 - val_loss: 0.8153
Epoch 8/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.6984 - loss: 0.8131 - val_accuracy: 0.6667 - val_loss: 0.7795
Epoch 9/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.7264 - loss: 0.8047 - val_accuracy: 0.6667 - val_loss: 0.7426
Epoch 10/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 17ms/step - accuracy: 0.7412 - loss: 0.7514 - val_accuracy: 0.7000 - val_loss: 0.7066
Epoch 11/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.7377 - loss: 0.7436 - val_accuracy: 0.7000 - val_loss: 0.6731
Epoch 12/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - accuracy: 0.7322 - loss: 0.6748 - val_accuracy: 0.7333 - val_loss: 0.6379
Epoch 13/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 14ms/step - accuracy: 0.7957 - loss: 0.6135 - val_accuracy: 0.7667 - val_loss: 0.6047
Epoch 14/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 22ms/step - accuracy: 0.7193 - loss: 0.6308 - val_accuracy: 0.8000 - val_loss: 0.5737
Epoch 15/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 18ms/step - accuracy: 0.8104 - loss: 0.5589 - val_accuracy: 0.8333 - val_loss: 0.5429
Epoch 16/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 11ms/step - accuracy: 0.7252 - loss: 0.5981 - val_accuracy: 0.8333 - val_loss: 0.5164
Epoch 17/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 12ms/step - accuracy: 0.7826 - loss: 0.5391 - val_accuracy: 0.8333 - val_loss: 0.4921
Epoch 18/50
```



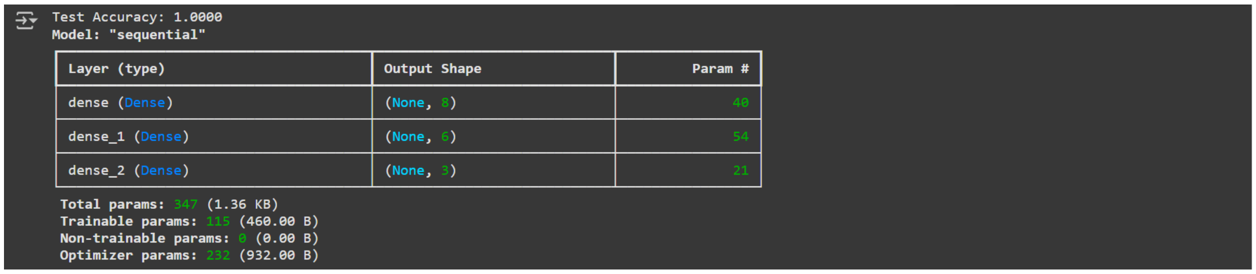
```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 25ms/step - accuracy: 0.8228 - loss: 0.4835 - val_accuracy: 0.8333 - val_loss: 0.4679
Epoch 19/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.7664 - loss: 0.5054 - val_accuracy: 0.8333 - val_loss: 0.4460
Epoch 20/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 11ms/step - accuracy: 0.8869 - loss: 0.4163 - val_accuracy: 0.8333 - val_loss: 0.4239
Epoch 21/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 19ms/step - accuracy: 0.8126 - loss: 0.4851 - val_accuracy: 0.8667 - val_loss: 0.4046
Epoch 22/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 17ms/step - accuracy: 0.8434 - loss: 0.4640 - val_accuracy: 0.8667 - val_loss: 0.3839
Epoch 23/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 40ms/step - accuracy: 0.8939 - loss: 0.4100 - val_accuracy: 0.9000 - val_loss: 0.3631
Epoch 24/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.8798 - loss: 0.3985 - val_accuracy: 0.9000 - val_loss: 0.3430
Epoch 25/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 20ms/step - accuracy: 0.9043 - loss: 0.4061 - val_accuracy: 0.9667 - val_loss: 0.3222
Epoch 26/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.9465 - loss: 0.3756 - val_accuracy: 0.9667 - val_loss: 0.3031
Epoch 27/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 32ms/step - accuracy: 0.9541 - loss: 0.3094 - val_accuracy: 0.9667 - val_loss: 0.2875
Epoch 28/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 41ms/step - accuracy: 0.9759 - loss: 0.2807 - val_accuracy: 0.9667 - val_loss: 0.2699
Epoch 29/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 24ms/step - accuracy: 0.9793 - loss: 0.2610 - val_accuracy: 0.9667 - val_loss: 0.2549
```



```
Epoch 30/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 30ms/step - accuracy: 0.9351 - loss: 0.2842 - val_accuracy: 0.9667 - val_loss: 0.2421
Epoch 31/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 1s 6ms/step - accuracy: 0.9649 - loss: 0.2783 - val_accuracy: 0.9667 - val_loss: 0.2266
Epoch 32/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9224 - loss: 0.2769 - val_accuracy: 0.9667 - val_loss: 0.2154
Epoch 33/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9675 - loss: 0.2287 - val_accuracy: 0.9667 - val_loss: 0.2039
Epoch 34/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9608 - loss: 0.2273 - val_accuracy: 0.9667 - val_loss: 0.1912
Epoch 35/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9402 - loss: 0.2573 - val_accuracy: 0.9667 - val_loss: 0.1807
Epoch 36/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9462 - loss: 0.2148 - val_accuracy: 0.9667 - val_loss: 0.1733
Epoch 37/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9605 - loss: 0.1895 - val_accuracy: 0.9667 - val_loss: 0.1651
Epoch 38/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9589 - loss: 0.2103 - val_accuracy: 0.9667 - val_loss: 0.1576
Epoch 39/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9496 - loss: 0.2097 - val_accuracy: 0.9667 - val_loss: 0.1501
Epoch 40/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 8ms/step - accuracy: 0.9342 - loss: 0.2122 - val_accuracy: 0.9667 - val_loss: 0.1416
Epoch 41/50
```



```
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9492 - loss: 0.1780 - val_accuracy: 0.9667 - val_loss: 0.1374
Epoch 42/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9330 - loss: 0.1854 - val_accuracy: 0.9667 - val_loss: 0.1317
Epoch 43/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 9ms/step - accuracy: 0.9454 - loss: 0.1816 - val_accuracy: 1.0000 - val_loss: 0.1258
Epoch 44/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9260 - loss: 0.1852 - val_accuracy: 1.0000 - val_loss: 0.1204
Epoch 45/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 10ms/step - accuracy: 0.9781 - loss: 0.1443 - val_accuracy: 1.0000 - val_loss: 0.1184
Epoch 46/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9496 - loss: 0.1602 - val_accuracy: 1.0000 - val_loss: 0.1122
Epoch 47/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9580 - loss: 0.1632 - val_accuracy: 1.0000 - val_loss: 0.1075
Epoch 48/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 7ms/step - accuracy: 0.9701 - loss: 0.1408 - val_accuracy: 1.0000 - val_loss: 0.1060
Epoch 49/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9798 - loss: 0.1318 - val_accuracy: 1.0000 - val_loss: 0.0997
Epoch 50/50
15/15 ━━━━━━━━━━━━━━━━━━━━ 0s 6ms/step - accuracy: 0.9852 - loss: 0.1289 - val_accuracy: 1.0000 - val_loss: 0.0976
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 58ms/step - accuracy: 1.0000 - loss: 0.0976
```



```
Test Accuracy: 1.0000
Model: "sequential"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| dense (Dense) | (None, 8) | 40 |
| dense_1 (Dense) | (None, 6) | 54 |
| dense_2 (Dense) | (None, 3) | 21 |

```
 Total params: 347 (1.36 KB)
 Trainable params: 115 (460.00 B)
 Non-trainable params: 0 (0.00 B)
 Optimizer params: 232 (932.00 B)
```

---

**Post lab questions:**

1. **Deep learning works well despite of ________________________ problem(s).**
   a. Sharp Minima
   b. Numerical instability (vanishing/exploding gradient)
   c. High capacity (susceptible to overfitting)
   **d. All of the above**

   **Ans:** All of the above

2. **The number of neurons in the output layer should match the number of classes (where no of classes are greater than 2 in a supervised learning task. True or False?**
   **a. True**
   b. False

   **Ans:** True

3. **List down activation function functions most widely used at hidden layer and output layer.**

   **Ans: Hidden Layer:** ReLU (Rectified Linear Unit), Leaky ReLU, Tanh, Sigmoid

   **Output Layer:**
   - Softmax (for multi-class classification)
   - Sigmoid (for binary classification)
   - Linear (for regression tasks)

---

**CO–2: Comprehend the Deep Network concepts.**

---

**Conclusion:**

In this experiment, we built deep neural networks to understand the role of layers, activation functions, and model architecture. We explored the significance of non-linearity using activation functions like ReLU, which helps in learning complex relationships in data. The experiment demonstrated how stacking dense layers improves the model's ability to transform input data. We also analyzed the effect of various hyperparameters on the network's performance, highlighting the importance of proper tuning for better accuracy and generalization.

---

**Grade: AA / AB / BB / BC / CC / CD / DD**

**Signature of faculty in-charge with date**

---

**References:**

**Books/ Journals/ Websites**

1. Josh Patterson and Adam Gibson, "Deep Learning A Practitioner's Approach", O'Reilly Media, 2017
2. Nikhil Buduma, "Fundamentals of Deep Learning Designing Next-Generation Machine Intelligence Algorithms", O'Reilly Media 2017
3. Ian Goodfellow Yoshua Bengio Aaron Courville. "Deep Learning", MIT Press 2017

---