**8a. Train a CNN model on a dataset and evaluate overfitting/underfitting: Apply L1/L2 regularization and dropout to improve generalization. Use K-fold cross-validation to evaluate the performance of a CNN model.**

**Aim:** The aim is to train a CNN model on a dataset, apply L1/L2 regularization and dropout to improve generalization, and use K-fold cross-validation to evaluate its performance and assess overfitting/underfitting.

## Description:

A **Convolutional Neural Network (CNN)** is a deep learning model designed for processing and analyzing visual data, such as images and videos. It mimics the way the human brain recognizes patterns using hierarchical feature extraction.

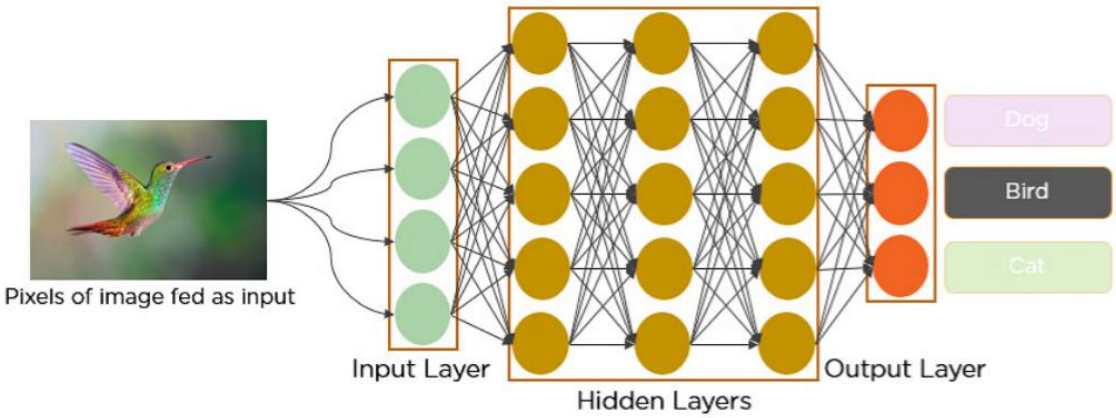



Fig: Convolutional Neural Network to identify the image of a bird

**Key Components of a CNN:**

1. **Convolutional Layers** – Apply filters to extract important features like edges, shapes, and textures from input images.
2. **Pooling Layers** – Reduce the spatial dimensions of feature maps while retaining essential information, improving efficiency and reducing overfitting.
3. **Fully Connected (Dense) Layers** – Flatten extracted features and classify the image based on learned patterns.
4. **Activation Functions (e.g., ReLU, Softmax)** – Introduce non-linearity and help in making classification decisions.
5. **Dropout & Regularization** – Prevent overfitting by randomly dropping connections and adding constraints on weights.

CNNs are widely used in image recognition, object detection, and computer vision tasks due to their ability to learn spatial hierarchies and detect complex patterns automatically.

1. **Overfitting –** Occurs when a model learns the training data too well, capturing noise and irrelevant details instead of general patterns. This leads to high accuracy on training data but poor performance on unseen data.
2. **Underfitting –** Happens when a model is too simple to capture underlying patterns in the data. It results in poor performance on both training and test data due to insufficient learning.
3. **L1 Regularization (Lasso) –** Adds the sum of absolute values of weights to the loss function, forcing some weights to become zero. This helps in feature selection and prevents overfitting.

4. **L2 Regularization (Ridge)** – Adds the sum of squared weights to the loss function, reducing the impact of large weights. This helps in smoothing the model and improving generalization.
5. **Dropout –** A regularization technique where random neurons are "dropped" (set to zero) during training. This prevents reliance on specific neurons, improving model robustness and reducing overfitting.
6. **K-Fold Cross-Validation –** A technique for evaluating model performance by dividing the dataset into K subsets (folds). The model is trained on K-1 folds and tested on the remaining fold, repeating the process K times. This ensures a more reliable performance assessment and reduces bias due to dataset splitting.

## Code:

```
import numpy as np
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout
from tensorflow.keras.regularizers import l2
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import KFold

# Load and preprocess CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0  # Normalize
y_train, y_test = to_categorical(y_train, 10), to_categorical(y_test, 10)  # One-hot encode labels

# Function to create CNN model with regularization and dropout
def create_model():
        model = Sequential([
            Conv2D(32, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(1e-4), input_shape=(32, 32, 3)),
            MaxPooling2D((2, 2)),
            Dropout(0.5),
            Conv2D(64, (3, 3), activation='relu', padding='same', kernel_regularizer=l2(1e-4)),
            MaxPooling2D((2, 2)),
            Dropout(0.5),
            Flatten(),
            Dense(128, activation='relu', kernel_regularizer=l2(1e-4)),
            Dropout(0.5),
            Dense(10, activation='softmax')
        ])
         model.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])
         return model

# Perform 5-Fold Cross-Validation
kf = KFold(n_splits=5, shuffle=True, random_state=42)
for fold, (train_idx, val_idx) in enumerate(kf.split(x_train)):
    model = create_model()
    model.fit(x_train[train_idx], y_train[train_idx], epochs=5, batch_size=64, verbose=1,
              validation_data=(x_train[val_idx], y_train[val_idx]))

# Train final model on full training set and evaluate on test set
final_model = create_model()
final_model.fit(x_train, y_train, epochs=10, batch_size=64, verbose=1)
test_loss, test_acc = final_model.evaluate(x_test, y_test, verbose=0)
print(f'Test Loss: {test_loss:.4f}, Test Accuracy: {test_acc*100:.2f}%')
```

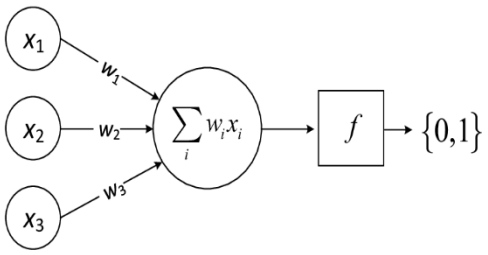
## Output

```
625/625 ━━━━━━━━━━━━━━━━━━━━ 6s 4ms/step - accuracy: 0.4237 - loss: 1.6205 - val_accuracy: 0.5309 - val_loss: 1.4025
Epoch 3/5
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.4696 - loss: 1.5136 - val_accuracy: 0.5675 - val_loss: 1.3224
Epoch 4/5
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.4972 - loss: 1.4557 - val_accuracy: 0.5970 - val_loss: 1.2743
Epoch 5/5
625/625 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.5212 - loss: 1.4078 - val_accuracy: 0.5978 - val_loss: 1.2635
Epoch 1/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.2626 - loss: 2.0176
Epoch 2/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 5ms/step - accuracy: 0.4564 - loss: 1.5439
Epoch 3/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 4s 4ms/step - accuracy: 0.5010 - loss: 1.4484
Epoch 4/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.5265 - loss: 1.3906
Epoch 5/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.5454 - loss: 1.3564
Epoch 6/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.5667 - loss: 1.3244
Epoch 7/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.5746 - loss: 1.3035
Epoch 8/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.5814 - loss: 1.2808
Epoch 9/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 5s 4ms/step - accuracy: 0.5903 - loss: 1.2663
Epoch 10/10
782/782 ━━━━━━━━━━━━━━━━━━━━ 3s 4ms/step - accuracy: 0.5925 - loss: 1.2614
Test Loss: 1.1075, Test Accuracy: 66.70%
```

**8b. Design a single unit perceptron for classification of iris dataset without using predefined models**

**Aim:**

**Description:**




A perceptron is a simple linear classifier that makes predictions based on a linear decision boundary. The goal of this experiment is to implement a Single Unit Perceptron from scratch to classify the Iris dataset without using predefined machine learning models. The perceptron will handle binary classification by distinguishing between two classes: *Iris-setosa* and *Iris-versicolor*.

The perceptron algorithm adjusts weights iteratively using the Perceptron Learning Rule until it converges or reaches a maximum number of iterations. The experiment will demonstrate:

1. **Loading and preprocessing the data.**
2. **Designing the perceptron algorithm.**
3. **Training the perceptron on the dataset.**
4. **Evaluating its performance on the test set.**

**Libraries Used:**

- **NumPy (np):** Supports numerical operations like creating arrays, matrix multiplication, and weight initialization.
- **Scikit-learn (sklearn.datasets):** Loads the **Iris dataset** for training and testing the perceptron model.
- **Matplotlib (plt):** Visualizes the **decision boundary** and data points to assess classification performance.

**Operations Performed:**

1. **Loading the Iris Dataset** – The first **100 samples** (two features: **sepal length & width**) are selected for **binary classification**.
2. **Data Preparation** – Features and target labels are formatted for training the perceptron.
3. **Defining Learning Parameters** – Learning rate (**0.1**) and iterations (**100**) are set for model training.
4. **Initializing Weights** – Random weights are assigned and updated during training.
5. **Activation Function** – A **threshold function** classifies inputs as **0 or 1** based on weighted sums.
6. **Training the Perceptron** – The model iterates over the data, updating weights using **gradient descent** while tracking errors.
7. **Plotting Decision Boundary** – The trained model's **classification performance** is visualized using **Matplotlib**.

**Program:**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris

# Load Iris dataset (first 100 samples, first two features)
iris = load_iris()
X = iris.data[:100, :2]
y = iris.target[:100]

# Convert labels: 0 →-1, 1 → 1 (for perceptron learning)
y = np.where(y == 0,-1, 1)

# Initialize weights (including bias)
weights = np.random.rand(3)  # 2 for features + 1 for bias
learning_rate = 0.1
epochs = 100

# Step Activation Function
def activation(z):
   return 1 if z >= 0 else-1

# Train the perceptron
for _ in range(epochs):
   for i in range(len(X)):
      x_i = np.insert(X[i], 0, 1)  # Add bias term
      y_pred = activation(np.dot(weights, x_i))
      weights += learning_rate * (y[i]- y_pred) * x_i  # Update weights

# Plot decision boundary
x_min, x_max = X[:, 0].min(), X[:, 0].max()
y_min, y_max = X[:, 1].min(), X[:, 1].max()
xx = np.linspace(x_min, x_max, 100)
yy =-(weights[1] * xx + weights[0]) / weights[2]  # Decision boundary formula

plt.scatter(X[:, 0], X[:, 1], c=y, cmap='bwr', edgecolors='k')
plt.plot(xx, yy, 'k--', label="Decision Boundary")
plt.xlabel("Sepal Length")
plt.ylabel("Sepal Width")
plt.legend()
```

```
plt.title("Perceptron Classification of Iris Dataset")
plt.show()
```

**Output:**