Problem Statement:

The primary objective of this study is to investigate the progressive improvements in neural networks by comparing the performance of a Fully Connected Neural Network (FCNN) using NumPy arrays, an Artificial Neural Network (ANN) with additional optimizations, and a Convolutional Neural Network (CNN) leveraging tensors. The analysis examines key advancements such as the use of tensors, optimizers, additional layers, initialization strategies, and training epochs. Through these improvements, accuracy increased significantly from a modest 10% to an impressive 75%.

Introduction to Models and Architectures

**Model 1: Fully Connected Neural Network (FCNN) - Numpy Based**

The FCNN in Model 1 serves as the baseline, using NumPy arrays to process input data. This model flattens the 32x32x3 image input into a 1D vector of 3072 features, which leads to the loss of spatial information. With a simple fully connected architecture and ReLU activation functions, the model achieves a relatively low accuracy of ~10%. Despite being a rudimentary approach, it serves as a foundational starting point for comparison.

**Model 2: Improved FCNN - Numpy with Tensors**

Building on Model 1, Model 2 introduces more advanced features, including optimized activation functions (LeakyReLU and ReLU) and regularization techniques like dropout. Although still using NumPy arrays to flatten the input images, these changes help reduce overfitting and improve generalization. The model demonstrates a significant improvement in performance, reaching an accuracy of ~51%.

**Model 3: Convolutional Neural Network (CNN)**

In Model 3, the CNN architecture fully leverages tensors, preserving the spatial dimensions of the input data (32x32x3). Convolutional layers are introduced to capture hierarchical spatial features, while max-pooling layers reduce the spatial dimensions. Batch normalization and dropout are also employed to regularize the model. The CNN model represents the most advanced approach, achieving an impressive accuracy of ~75%.

Architectures Layers

| **Feature** | **FCNN(NumPy Based)** | | | **ImprovedFCNN (NumPy with Tensors)** | | **Convolutional Neural Network (CNN)** |
| --- | --- | --- | --- | --- | --- | --- |
| Model Type | SimpleDense Network | | | Dense Network with Dropout & ReLU | | Convolutional Neural Network |
| Input Shape | (32, 32, 3) -> Flattened to (3072,) | | | (32, 32, 3) -> Flattened to (3072,) | | Maintains spatial dimensions (32, 32, 3) |
| Data Preprocessing | None | | | Normalization + One-Hot Encoding | | Normalization + One-Hot Encoding |
| Regularization | None | | | Dropout (20%) | | Dropout (30%) + Batch Normalization |
| Activation Function | ReLU | | | ReLU + LeakyReLU | | ReLU |
| Filter Size | | N/A | N/A | | 3x3 | |
| Activation | | ReLU | ReLU + LeakyReLU | | ReLU | |
| Parameters | | ~3M | ~3M | | ~500K | |
| Dropout | | No | Yes (20%) | | Yes (30%) | |
| Batch Normalization | | No | No | | Yes | |
| Padding | | None | None | | Same (preserves size) | |
| Stride | | None | None | | 1 (reduces spatial size  through pooling) | |
| Optimizer | SGD | | | Adam | | Adam |
| Model Architecture | FullyConnected Layers | | | Fully Connected Layers | | Convolutional Layers + Dense Layers |
| Parameters | ~3 Million | | | ~3 Million | | ~500K (much fewer due to weight sharing) |
| LearningRate Scheduling | No | | | No | | No |
| Training Duration | Slower | | | Faster due to Adam | | Fastest due to spatial convolution |
| Accuracy (Test) | ~10% | | | ~51% | | ~75% |
| Loss (Test) | High | | | Lower | | Lowest |

Flaws and Positives

| **Model** | **Flaws** | **Positives** |
| --- | --- | --- |
| FCNN (NumPy Based) | - Ignores spatial relationships in image data | - Simple architecture for beginners |
|  | - Overfitting due to lack of regularization | - Quick to implement and test basic ideas |
| Improved FCNN | - Still does not utilize spatial structure | - Regularization reduces overfitting |
|  | - High parameter count | - Adam optimizer improves learning efficiency |
| CNN | - Needs more computational resources | - Preserves spatial information |
|  | - Slightly more complex to implement | - Significantly reduces parameters with weight sharing |

Impact of Key Improvements of CNN Architecture

| **Improvement** | **Impact** |
| --- | --- |
| Dropout | Reduces overfitting by randomly disabling neurons during training. |
| Batch Normalization | Stabilizes training, accelerates convergence, and allows higher learning rates. |
| ReLU Activation | Solves vanishing gradient problem and improves gradient flow. |
| LeakyReLU | Avoids "dying ReLU" problem, keeping neurons active even with negative gradients. |
| Convolutional Layers | Captures spatial relationships, reduces parameter count via weight sharing. |
| Padding ("same") | Maintains spatial dimensions after convolution, ensuring better spatial feature extraction. |
| Stride in Convolution | Down samples spatial dimensions, reducing tensor size while preserving essential features. |
| Adam Optimizer | Adaptive learning rate improves convergence speed compared to vanilla SGD. |

Architectural Code/Output Section

**Model 1: Fully Connected Neural Network (FCNN) - Numpy Based**

#Model 1 architecture with FCNN using Numpy not DL model : custom function named as "Ld"

import numpy as np

import matplotlib.pyplot as plt

# ReLU activation and its derivative

def Ld\_relu(x):

    return np.maximum(0, x)  # ReLU activation

def Ld\_relu\_derivative(x):

    return np.where(x > 0, 1, 0)  # Derivative of ReLU

# Softmax activation (for output layer)

def Ld\_softmax(x):

    exp\_x = np.exp(x - np.max(x, axis=1, keepdims=True))  # Stability trick (subtract max)

    return exp\_x / np.sum(exp\_x, axis=1, keepdims=True)  # Softmax function

# Cross-Entropy loss

def Ld\_cross\_entropy\_loss(predictions, labels):

    m = len(labels)

    # For multi-class classification, cross-entropy loss

    return -np.mean(np.log(predictions[np.arange(m), labels]))

# Train-Test Split

def Ld\_train\_test\_split(X, y, test\_size=0.2):

    indices = np.random.permutation(len(X))

    test\_size = int(len(X) \* test\_size)

    return X[indices[test\_size:]], X[indices[:test\_size]], y[indices[test\_size:]], y[indices[:test\_size]]

# Forward propagation for the updated model with two hidden layers

def Ld\_forward(X, weights1, bias1, weights2, bias2, weights3, bias3):

    a1 = Ld\_relu(np.dot(X, weights1) + bias1)  # First hidden layer

    a2 = Ld\_relu(np.dot(a1, weights2) + bias2)  # Second hidden layer

    a3 = Ld\_softmax(np.dot(a2, weights3) + bias3)  # Output layer

    return a1, a2, a3

# Backward propagation for the updated model with two hidden layers

def Ld\_backward(X, y, a1, a2, a3, weights2, weights3):

    m = X.shape[0]

    d\_z3 = a3

    d\_z3[np.arange(m), y] -= 1  # Gradient for softmax layer

    d\_z3 /= m

    d\_weights3 = np.dot(a2.T, d\_z3)

    d\_bias3 = np.sum(d\_z3, axis=0, keepdims=True)

    d\_a2 = np.dot(d\_z3, weights3.T)

    d\_z2 = d\_a2 \* Ld\_relu\_derivative(a2)

    d\_weights2 = np.dot(a1.T, d\_z2)

    d\_bias2 = np.sum(d\_z2, axis=0, keepdims=True)

    d\_a1 = np.dot(d\_z2, weights2.T)

    d\_z1 = d\_a1 \* Ld\_relu\_derivative(a1)

    d\_weights1 = np.dot(X.T, d\_z1)

    d\_bias1 = np.sum(d\_z1, axis=0, keepdims=True)

    return d\_weights1, d\_bias1, d\_weights2, d\_bias2, d\_weights3, d\_bias3

# Training the model

def Ld\_train(X\_train, y\_train, X\_test, y\_test, input\_size, hidden\_size1, hidden\_size2, output\_size, epochs=100, lr=0.01):

    # Initialize weights with Xavier initialization

    weights1 = np.random.randn(input\_size, hidden\_size1) \* np.sqrt(1. / input\_size)

    bias1 = np.zeros((1, hidden\_size1))

    weights2 = np.random.randn(hidden\_size1, hidden\_size2) \* np.sqrt(1. / hidden\_size1)

    bias2 = np.zeros((1, hidden\_size2))

    weights3 = np.random.randn(hidden\_size2, output\_size) \* np.sqrt(1. / hidden\_size2)

    bias3 = np.zeros((1, output\_size))

    # Lists to store the loss and accuracy for plotting

    loss\_history = []

    accuracy\_history = []

    # Train the model for a number of epochs

    for epoch in range(epochs):

        a1, a2, a3 = Ld\_forward(X\_train, weights1, bias1, weights2, bias2, weights3, bias3)

        loss = Ld\_cross\_entropy\_loss(a3, y\_train)

        d\_weights1, d\_bias1, d\_weights2, d\_bias2, d\_weights3, d\_bias3 = Ld\_backward(X\_train, y\_train, a1, a2, a3, weights2, weights3)

        # Update weights and biases using gradient descent

        weights1 -= lr \* d\_weights1

        bias1 -= lr \* d\_bias1

        weights2 -= lr \* d\_weights2

        bias2 -= lr \* d\_bias2

        weights3 -= lr \* d\_weights3

        bias3 -= lr \* d\_bias3

        # Track loss and accuracy for each epoch

        loss\_history.append(loss)

        accuracy = Ld\_compute\_accuracy(X\_test, y\_test, weights1, bias1, weights2, bias2, weights3, bias3)

        accuracy\_history.append(accuracy)

        # Print the loss and accuracy at each epoch

        if epoch % 10 == 0:

            print(f"Epoch {epoch}, Loss: {loss:.4f}, Accuracy: {accuracy:.2f}%")

    return weights1, bias1, weights2, bias2, weights3, bias3, loss\_history, accuracy\_history

# Accuracy function

def Ld\_compute\_accuracy(X\_test, y\_test, weights1, bias1, weights2, bias2, weights3, bias3):

    \_, \_, a3 = Ld\_forward(X\_test, weights1, bias1, weights2, bias2, weights3, bias3)

    predictions = np.argmax(a3, axis=1)

    accuracy = np.mean(predictions == y\_test) \* 100

    return accuracy

# LdModel summary function

def LdModel\_summary(input\_size, hidden\_size1, hidden\_size2, output\_size):

    print(f"Model Architecture Summary:")

    print(f"Input Layer: {input\_size} units")

    print(f"Hidden Layer 1: {hidden\_size1} units")

    print(f"Hidden Layer 2: {hidden\_size2} units")

    print(f"Output Layer: {output\_size} units")

    # Number of parameters for each layer

    params\_layer1 = input\_size \* hidden\_size1 + hidden\_size1  # Weights + biases

    params\_layer2 = hidden\_size1 \* hidden\_size2 + hidden\_size2  # Weights + biases

    params\_layer3 = hidden\_size2 \* output\_size + output\_size  # Weights + biases

    total\_params = params\_layer1 + params\_layer2 + params\_layer3

    print(f"Total parameters: {total\_params}")

    print(f"Layer 1 (Input to Hidden 1): {params\_layer1} parameters")

    print(f"Layer 2 (Hidden 1 to Hidden 2): {params\_layer2} parameters")

    print(f"Layer 3 (Hidden 2 to Output): {params\_layer3} parameters")

# Example usage with CIFAR-10 data

input\_size = 32 \* 32 \* 3  # CIFAR-10 image size (32x32x3 channels)

hidden\_size1 = 128

hidden\_size2 = 64

output\_size = 10  # CIFAR-10 classes

# Simulated data: 1000 training samples

X = np.random.randn(1000, input\_size)

y = np.random.randint(0, output\_size, 1000)

X\_train, X\_test, y\_train, y\_test = Ld\_train\_test\_split(X, y)

# Print model summary

LdModel\_summary(input\_size, hidden\_size1, hidden\_size2, output\_size)

# Train the model

weights1, bias1, weights2, bias2, weights3, bias3, loss\_history, accuracy\_history = Ld\_train(X\_train, y\_train, X\_test, y\_test, input\_size, hidden\_size1, hidden\_size2, output\_size, epochs=100, lr=0.001)

# Plotting Loss and Accuracy

plt.figure(figsize=(10, 3))

# Loss plot

plt.subplot(1, 2, 1)

plt.plot(loss\_history, label="Loss")

plt.title("Loss over Epochs")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.legend()

# Accuracy plot

plt.subplot(1, 2, 2)

plt.plot(accuracy\_history, label="Accuracy")

plt.title("Accuracy over Epochs")

plt.xlabel("Epochs")

plt.ylabel("Accuracy (%)")

plt.legend()

plt.tight\_layout()

plt.show()

# Final Test Accuracy

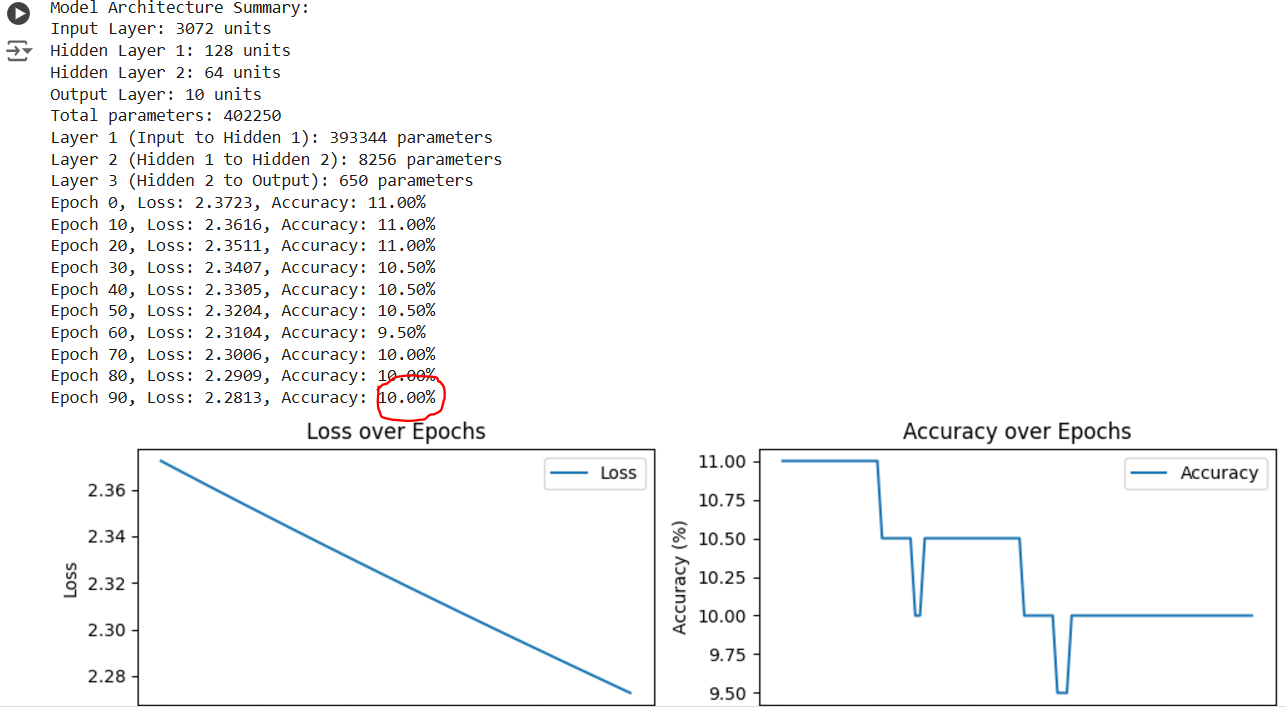
final\_accuracy = accuracy\_history[-1]

print(f"Final Test Accuracy: {final\_accuracy:.2f}%")

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END PROGRAM – ARCHITECTURE 1\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Output of Architecture 1**



**Model 2: Improved FCNN - Numpy with Tensors**

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense, Dropout, Flatten, LeakyReLU, ReLU

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping

# Load CIFAR-10 dataset

(X\_train, y\_train), (X\_test, y\_test) = cifar10.load\_data()

# Normalize the data to [0, 1]

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

# One-hot encoding of labels

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Define the model

def build\_model(input\_shape=(32, 32, 3), dropout\_rate=0.2):

    model = Sequential()

    # Flatten input image to vector

    model.add(Flatten(input\_shape=input\_shape))

    # Hidden Layer 1 with Dropout and LeakyReLU activation

    model.add(Dense(512))

    model.add(Dropout(dropout\_rate))

    model.add(LeakyReLU(alpha=0.1))

    # Hidden Layer 2 with Dropout and ReLU activation

    model.add(Dense(256))

    model.add(Dropout(dropout\_rate))

    model.add(ReLU())

    # Hidden Layer 3 with Dropout

    model.add(Dense(128))

    model.add(Dropout(dropout\_rate))

    model.add(ReLU())

    # Output layer

    model.add(Dense(10, activation='softmax'))

    # Compile the model

    model.compile(optimizer=Adam(learning\_rate=0.001), loss='categorical\_crossentropy', metrics=['accuracy'])

    return model

# Build and summarize the model

model = build\_model()

# Early stopping callback

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

# Train the model

history = model.fit(X\_train, y\_train, validation\_data=(X\_test, y\_test), epochs=50, batch\_size=64,

                    callbacks=[early\_stopping], verbose=1)

# Plot accuracy and loss

plt.figure(figsize=(12, 5))

# Accuracy plot

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.title('Accuracy over epochs')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

# Loss plot

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.title('Loss over epochs')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

# Evaluate model on the test set

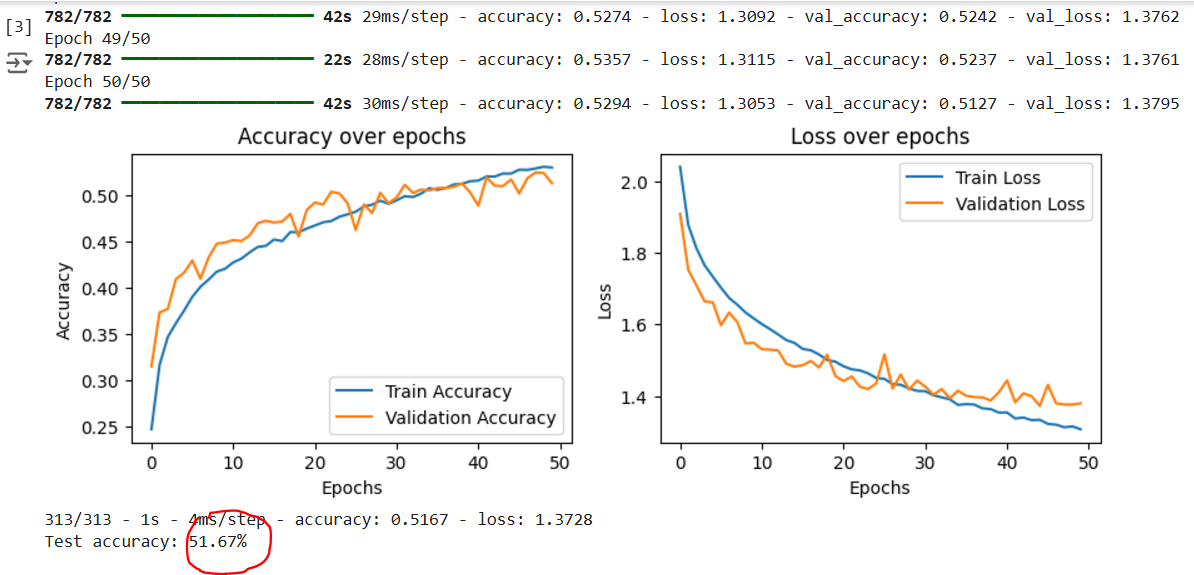
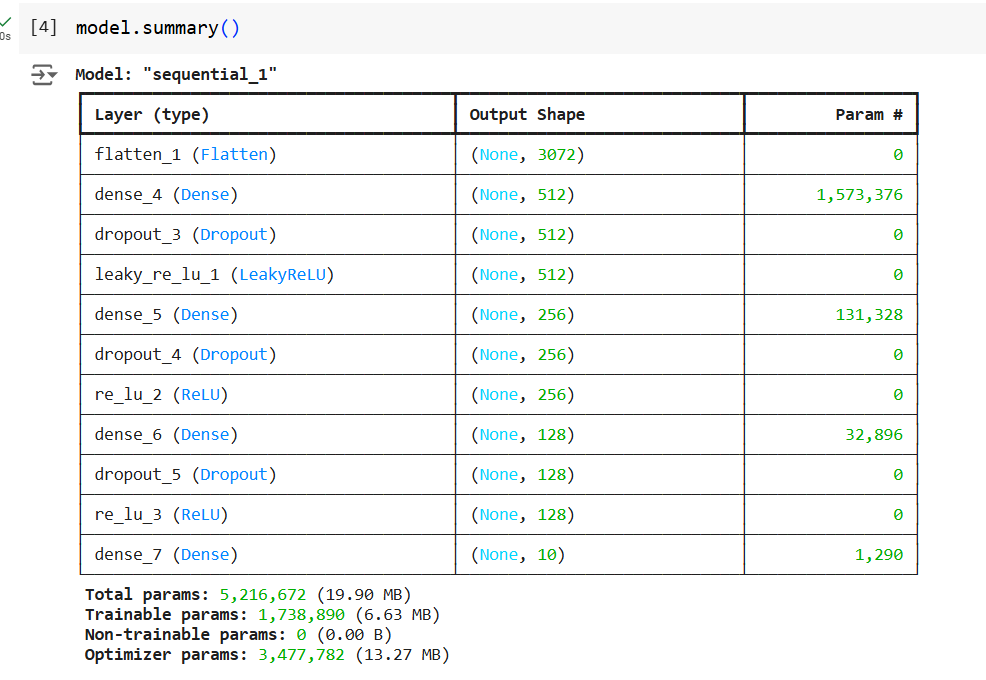
test\_loss, test\_acc = model.evaluate(X\_test, y\_test, verbose=2)

print(f"Test accuracy: {test\_acc \* 100:.2f}%")

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END PROGRAM – ARCHITECTURE 2\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**Output of Architecture 2:**



**Model 3: Convolutional Neural Network (CNN)**

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import cifar10

from tensorflow.keras.utils import to\_categorical

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense, Dropout, BatchNormalization

from tensorflow.keras.optimizers import Adam

from tensorflow.keras.callbacks import EarlyStopping

# Load CIFAR-10 dataset

(X\_train, y\_train), (X\_test, y\_test) = cifar10.load\_data()

# Normalize the data to [0, 1]

X\_train = X\_train / 255.0

X\_test = X\_test / 255.0

# One-hot encoding of labels

y\_train = to\_categorical(y\_train, 10)

y\_test = to\_categorical(y\_test, 10)

# Define the CNN model

def build\_cnn\_model(input\_shape=(32, 32, 3), dropout\_rate=0.3):

    model = Sequential()

    # Convolutional Layer 1

    model.add(Conv2D(32, (3, 3), activation='relu', input\_shape=input\_shape, padding='same'))

    model.add(BatchNormalization())

    model.add(MaxPooling2D(pool\_size=(2, 2)))

    # Convolutional Layer 2

    model.add(Conv2D(64, (3, 3), activation='relu', padding='same'))

    model.add(BatchNormalization())

    model.add(MaxPooling2D(pool\_size=(2, 2)))

    # Convolutional Layer 3

    model.add(Conv2D(128, (3, 3), activation='relu', padding='same'))

    model.add(BatchNormalization())

    model.add(MaxPooling2D(pool\_size=(2, 2)))

    # Flatten Layer

    model.add(Flatten())

    # Fully Connected Layer 1

    model.add(Dense(128, activation='relu'))

    model.add(Dropout(dropout\_rate))

    # Fully Connected Layer 2

    model.add(Dense(64, activation='relu'))

    model.add(Dropout(dropout\_rate))

    # Output Layer

    model.add(Dense(10, activation='softmax'))

    # Compile the model

    model.compile(optimizer=Adam(learning\_rate=0.001), loss='categorical\_crossentropy', metrics=['accuracy'])

    return model

# Build and summarize the model

cnn\_model = build\_cnn\_model()

cnn\_model.summary()

# Early stopping callback

early\_stopping = EarlyStopping(monitor='val\_loss', patience=5, restore\_best\_weights=True)

# Train the CNN model

history = cnn\_model.fit(

    X\_train, y\_train,

    validation\_data=(X\_test, y\_test),

    epochs=50,

    batch\_size=64,

    callbacks=[early\_stopping],

    verbose=1

)

# Plot accuracy and loss

plt.figure(figsize=(12, 5))

# Accuracy plot

plt.subplot(1, 2, 1)

plt.plot(history.history['accuracy'], label='Train Accuracy')

plt.plot(history.history['val\_accuracy'], label='Validation Accuracy')

plt.title('Accuracy over epochs')

plt.xlabel('Epochs')

plt.ylabel('Accuracy')

plt.legend()

# Loss plot

plt.subplot(1, 2, 2)

plt.plot(history.history['loss'], label='Train Loss')

plt.plot(history.history['val\_loss'], label='Validation Loss')

plt.title('Loss over epochs')

plt.xlabel('Epochs')

plt.ylabel('Loss')

plt.legend()

plt.show()

# Evaluate the model on the test set

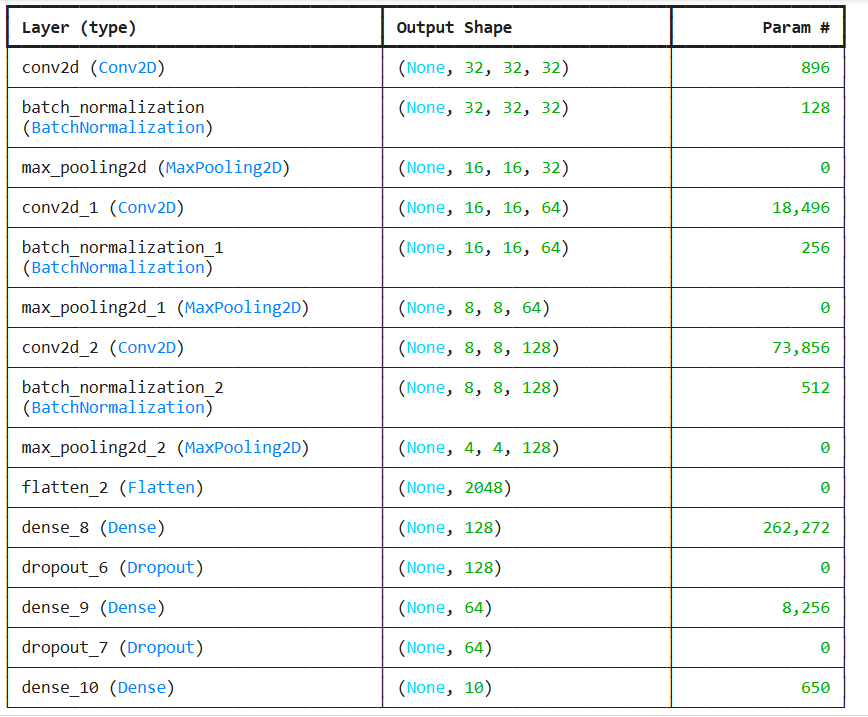
test\_loss, test\_acc = cnn\_model.evaluate(X\_test, y\_test, verbose=2)

print(f"Test accuracy: {test\_acc \* 100:.2f}%")

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*END PROGRAM – ARCHITECTURE 3\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

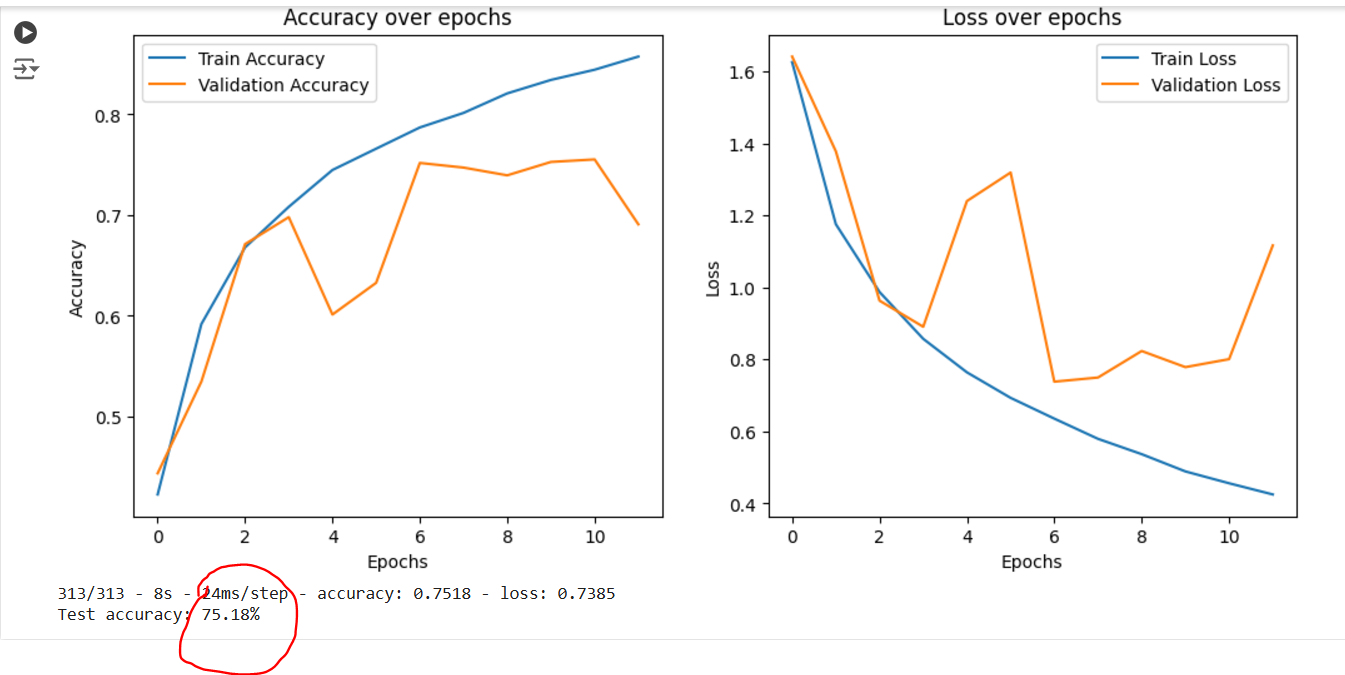
**Output of Architecture 3 using CNN**



**Total params:** 365,322 (1.39 MB)

**Trainable params:** 364,874 (1.39 MB)

**Non-trainable params:** 448 (1.75 KB)



Future Improvements and Conclusion

Future work can focus on incorporating advanced architectures such as Residual Networks (ResNet) or Inception Networks, which excel in deeper learning tasks. Additionally, exploring transfer learning with pre-trained models (e.g., VGG, MobileNet) and hyperparameter tuning may further improve classification accuracy and reduce training time. This study highlights the transformative impact of leveraging tensors, convolutional layers, and modern optimizations in neural networks. The transition from FCNNs (~10% accuracy) to CNNs (~75% accuracy) underscores the importance of preserving spatial information, incorporating advanced layers, and optimizing training techniques to enhance model performance in image classification tasks.