nndl-l-vinay-kumar-reddy-126-lab-3

October 25, 2024

#1. Data Preprocessing:#

Load the CIFAR-10 dataset.

Perform necessary data preprocessing steps:

Normalize pixel values to range between 0 and 1.

Convert class labels into one-hot encoded format.

Split the dataset into training and test sets (e.g., 50,000 images for training and 10,000 for testing).

Optionally, apply data augmentation techniques (such as random flips, rotations, or shifts) to improve the generalization of the model.

[2]:

```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from sklearn.model_selection import train_test_split

# Load the CIFAR-10 dataset
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert class labels to one-hot encoded format
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Flatten the input data for ANN (from 32x32x3 to 3072)
x_train = x_train.reshape(-1, 32*32*3)
x_test = x_test.reshape(-1, 32*32*3)

print("Training data shape", x_train.shape)
print("Training labels shape", y_train.shape)
print("Testing data shape", x_test.shape)
print("Testing labels shape", y_test.shape)
```

```
Training data shape (50000, 3072)
```

```
Training labels shape (50000, 10)
Testing data shape (10000, 3072)
Testing labels shape (10000, 10)
```

Normalization: Scale pixel values to a range between 0 and 1.

One-Hot Encoding: Convert the class labels into one-hot encoded format for multi-class classification.

Data Splitting: Split the dataset into training (50,000) and testing (10,000) images.

#2. Network Architecture Design:#

Design a feedforward neural network to classify the images.

Input Layer: The input shape should match the 32x32x3 dimensions of the CIFAR-10 images.

Hidden Layers: Use appropriate layers.

Output Layer: The final layer should have 10 output neurons (one for each class) with a softmax activation function for multi-class classification.

[3]:
```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout

# Define the ANN model
model = Sequential()

# Input layer (3072 features from the 32x32x3 image)
model.add(Dense(512, activation='relu', input_shape=(32*32*3,)))

# Hidden layers with ReLU and Dropout for regularization
model.add(Dense(256, activation='relu'))
model.add(Dropout(0.5))  # Dropout to avoid overfitting

model.add(Dense(128, activation='tanh'))
model.add(Dropout(0.5))

# Output layer (10 classes with softmax for multi-class classification)
model.add(Dense(10, activation='softmax'))

# Display model summary
model.summary()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

```
Model: "sequential"
```

```
 Layer (type)                         Output Shape                     ␣
↪Param #

 dense (Dense)                        (None, 512)                    ␣
↪1,573,376

 dense_1 (Dense)                      (None, 256)                      ␣
↪131,328

 dropout (Dropout)                    (None, 256)                          ␣
↪  0

 dense_2 (Dense)                      (None, 128)                       ␣
↪32,896

 dropout_1 (Dropout)                  (None, 128)                          ␣
↪  0

 dense_3 (Dense)                      (None, 10)                         ␣
↪1,290


 Total params: 1,738,890 (6.63 MB)

 Trainable params: 1,738,890 (6.63 MB)

 Non-trainable params: 0 (0.00 B)
```

Input Layer: 32x32x3 (RGB image).

Convolutional Layers: To detect patterns like edges, colors, or textures.

Pooling Layers: To downsample the image and reduce complexity.

Fully Connected Layers: To classify the extracted features into categories.

Output Layer: 10 neurons with softmax activation for multi-class classification.

Justification

Convolutional layers help in automatically learning filters for feature extraction.

Pooling layers reduce the number of parameters and computational load.

Fully connected layers consolidate the extracted features into final class scores.

#3. Activation Functions

ReLU (Rectified Linear Unit) is efficient for preventing the vanishing gradient problem during backpropagation by allowing faster learning.

tanh ensures that the values are centered around zero, which can improve convergence in some cases.

```
[4]: # No change needed in the previous code as ReLU is already used.
```

Role in Backpropagation:

ReLU: ReLU mitigates the vanishing gradient problem (which is common with Sigmoid and Tanh) because its gradient does not saturate (except for the zero output case).ReLU deactivates neurons when the input is negative (output is 0), making the model sparse and more computationally efficient.

tanh: can be useful in cases where the input data is centered around zero, but it may suffer from the vanishing gradient problem in deeper layers.

#4. Loss Function and Optimizer

The most suitable loss function for multi-class classification is categorical crossentropy. You could compare this with:

Mean Squared Error (MSE): Not ideal for classification but used to compare performance.

Sparse Categorical Crossentropy: Another variant of cross-entropy when the labels are integers.

Use Adam optimizer due to its adaptive learning rate and ability to handle sparse gradients.

```
[5]: # Compile the model
model.compile(optimizer='adam', loss='categorical_crossentropy',␣
 ↪metrics=['accuracy'])
```

Effect of Optimizer & Learning Rate:

Adam adjusts the learning rate dynamically, leading to faster convergence.

If the model isn't converging, reduce the learning rate to allow for finer updates.

#5. Training the Model:

Implement backpropagation to update the weights and biases of the network during training.

Train the model for a fixed number of epochs (e.g., 50 epochs) and monitor the training and validation accuracy.

```
[6]: # Train the model (batch size of 64 and 50 epochs)
history = model.fit(x_train, y_train, batch_size=64, epochs=50,␣
 ↪validation_data=(x_test, y_test))
```

```
Epoch 1/50
782/782             11s 8ms/step -
accuracy: 0.1552 - loss: 2.6320 - val_accuracy: 0.3099 - val_loss: 1.8767
Epoch 2/50
782/782             3s 3ms/step -
```

```
accuracy: 0.3168 - loss: 1.8655 - val_accuracy: 0.3550 - val_loss: 1.7682
Epoch 3/50
782/782            2s 3ms/step -
accuracy: 0.3337 - loss: 1.8257 - val_accuracy: 0.3673 - val_loss: 1.7387
Epoch 4/50
782/782            3s 3ms/step -
accuracy: 0.3409 - loss: 1.7992 - val_accuracy: 0.3733 - val_loss: 1.7248
Epoch 5/50
782/782            2s 3ms/step -
accuracy: 0.3614 - loss: 1.7763 - val_accuracy: 0.3783 - val_loss: 1.7143
Epoch 6/50
782/782            3s 3ms/step -
accuracy: 0.3620 - loss: 1.7554 - val_accuracy: 0.3723 - val_loss: 1.7151
Epoch 7/50
782/782            3s 3ms/step -
accuracy: 0.3685 - loss: 1.7403 - val_accuracy: 0.3816 - val_loss: 1.6966
Epoch 8/50
782/782            2s 3ms/step -
accuracy: 0.3773 - loss: 1.7181 - val_accuracy: 0.4048 - val_loss: 1.6604
Epoch 9/50
782/782            3s 4ms/step -
accuracy: 0.3838 - loss: 1.7164 - val_accuracy: 0.4040 - val_loss: 1.6692
Epoch 10/50
782/782            4s 3ms/step -
accuracy: 0.3877 - loss: 1.6961 - val_accuracy: 0.4022 - val_loss: 1.6640
Epoch 11/50
782/782            2s 3ms/step -
accuracy: 0.3951 - loss: 1.6878 - val_accuracy: 0.3919 - val_loss: 1.6895
Epoch 12/50
782/782            2s 3ms/step -
accuracy: 0.3949 - loss: 1.6804 - val_accuracy: 0.4141 - val_loss: 1.6362
Epoch 13/50
782/782            3s 3ms/step -
accuracy: 0.4059 - loss: 1.6702 - val_accuracy: 0.4128 - val_loss: 1.6388
Epoch 14/50
782/782            2s 3ms/step -
accuracy: 0.4036 - loss: 1.6684 - val_accuracy: 0.4139 - val_loss: 1.6243
Epoch 15/50
782/782            2s 3ms/step -
accuracy: 0.4089 - loss: 1.6511 - val_accuracy: 0.4078 - val_loss: 1.6748
Epoch 16/50
782/782            2s 3ms/step -
accuracy: 0.4052 - loss: 1.6561 - val_accuracy: 0.4153 - val_loss: 1.6438
Epoch 17/50
782/782            2s 3ms/step -
accuracy: 0.4157 - loss: 1.6421 - val_accuracy: 0.4179 - val_loss: 1.6238
Epoch 18/50
782/782            3s 3ms/step -
```

```
accuracy: 0.4130 - loss: 1.6383 - val_accuracy: 0.4208 - val_loss: 1.6243
Epoch 19/50
782/782             3s 3ms/step -
accuracy: 0.4124 - loss: 1.6362 - val_accuracy: 0.4262 - val_loss: 1.6074
Epoch 20/50
782/782             5s 3ms/step -
accuracy: 0.4175 - loss: 1.6288 - val_accuracy: 0.4240 - val_loss: 1.6126
Epoch 21/50
782/782             3s 3ms/step -
accuracy: 0.4145 - loss: 1.6290 - val_accuracy: 0.4254 - val_loss: 1.5939
Epoch 22/50
782/782             2s 3ms/step -
accuracy: 0.4208 - loss: 1.6191 - val_accuracy: 0.4254 - val_loss: 1.6119
Epoch 23/50
782/782             3s 3ms/step -
accuracy: 0.4202 - loss: 1.6188 - val_accuracy: 0.4384 - val_loss: 1.5773
Epoch 24/50
782/782             2s 3ms/step -
accuracy: 0.4231 - loss: 1.6113 - val_accuracy: 0.4421 - val_loss: 1.5734
Epoch 25/50
782/782             2s 3ms/step -
accuracy: 0.4260 - loss: 1.6054 - val_accuracy: 0.4206 - val_loss: 1.6191
Epoch 26/50
782/782             3s 3ms/step -
accuracy: 0.4204 - loss: 1.6215 - val_accuracy: 0.4220 - val_loss: 1.6222
Epoch 27/50
782/782             3s 3ms/step -
accuracy: 0.4280 - loss: 1.6083 - val_accuracy: 0.4305 - val_loss: 1.5984
Epoch 28/50
782/782             3s 3ms/step -
accuracy: 0.4291 - loss: 1.5953 - val_accuracy: 0.4320 - val_loss: 1.5823
Epoch 29/50
782/782             5s 3ms/step -
accuracy: 0.4258 - loss: 1.6027 - val_accuracy: 0.4364 - val_loss: 1.5797
Epoch 30/50
782/782             3s 3ms/step -
accuracy: 0.4298 - loss: 1.5961 - val_accuracy: 0.4288 - val_loss: 1.5952
Epoch 31/50
782/782             2s 3ms/step -
accuracy: 0.4295 - loss: 1.5993 - val_accuracy: 0.4325 - val_loss: 1.6086
Epoch 32/50
782/782             4s 5ms/step -
accuracy: 0.4334 - loss: 1.5843 - val_accuracy: 0.4437 - val_loss: 1.5594
Epoch 33/50
782/782             2s 3ms/step -
accuracy: 0.4351 - loss: 1.5852 - val_accuracy: 0.4344 - val_loss: 1.5834
Epoch 34/50
782/782             2s 3ms/step -
```

```
accuracy: 0.4342 - loss: 1.5794 - val_accuracy: 0.4422 - val_loss: 1.5607
Epoch 35/50
782/782            3s 4ms/step -
accuracy: 0.4366 - loss: 1.5766 - val_accuracy: 0.4275 - val_loss: 1.5992
Epoch 36/50
782/782            2s 3ms/step -
accuracy: 0.4330 - loss: 1.5863 - val_accuracy: 0.4488 - val_loss: 1.5587
Epoch 37/50
782/782            3s 4ms/step -
accuracy: 0.4375 - loss: 1.5689 - val_accuracy: 0.4350 - val_loss: 1.6005
Epoch 38/50
782/782            2s 3ms/step -
accuracy: 0.4388 - loss: 1.5768 - val_accuracy: 0.4476 - val_loss: 1.5720
Epoch 39/50
782/782            2s 3ms/step -
accuracy: 0.4368 - loss: 1.5705 - val_accuracy: 0.4393 - val_loss: 1.5753
Epoch 40/50
782/782            3s 3ms/step -
accuracy: 0.4363 - loss: 1.5720 - val_accuracy: 0.4476 - val_loss: 1.5504
Epoch 41/50
782/782            2s 3ms/step -
accuracy: 0.4373 - loss: 1.5691 - val_accuracy: 0.4114 - val_loss: 1.6363
Epoch 42/50
782/782            3s 3ms/step -
accuracy: 0.4417 - loss: 1.5624 - val_accuracy: 0.4354 - val_loss: 1.5883
Epoch 43/50
782/782            5s 3ms/step -
accuracy: 0.4459 - loss: 1.5598 - val_accuracy: 0.4395 - val_loss: 1.5676
Epoch 44/50
782/782            3s 3ms/step -
accuracy: 0.4401 - loss: 1.5594 - val_accuracy: 0.4401 - val_loss: 1.5761
Epoch 45/50
782/782            3s 3ms/step -
accuracy: 0.4470 - loss: 1.5567 - val_accuracy: 0.4349 - val_loss: 1.5849
Epoch 46/50
782/782            3s 3ms/step -
accuracy: 0.4470 - loss: 1.5523 - val_accuracy: 0.4449 - val_loss: 1.5622
Epoch 47/50
782/782            2s 3ms/step -
accuracy: 0.4447 - loss: 1.5530 - val_accuracy: 0.4425 - val_loss: 1.5735
Epoch 48/50
782/782            2s 3ms/step -
accuracy: 0.4484 - loss: 1.5519 - val_accuracy: 0.4499 - val_loss: 1.5479
Epoch 49/50
782/782            2s 3ms/step -
accuracy: 0.4513 - loss: 1.5453 - val_accuracy: 0.4449 - val_loss: 1.5692
Epoch 50/50
782/782            2s 3ms/step -
```

```
accuracy: 0.4503 - loss: 1.5412 - val_accuracy: 0.4500 - val_loss: 1.5541
```

Backpropagation & Learning Rate:

Backpropagation updates the weights in each layer by calculating the gradient of the loss with respect to the weights and adjusting them using the learning rate.

The learning rate determines how large these weight updates are. If it's too high, the model may overshoot optimal points; if too low, it might converge slowly.

#6. Model Evaluation: After training, evaluate the performance of your model on the test set.

Calculate accuracy, precision, recall, F1-score, and the confusion matrix to understand the model's classification performance.

[7]:
```python
from sklearn.metrics import classification_report, confusion_matrix

# Evaluate the model
test_loss, test_acc = model.evaluate(x_test, y_test)

# Get predictions
y_pred = model.predict(x_test)
y_pred_classes = y_pred.argmax(axis=1)
y_true = y_test.argmax(axis=1)

# Classification report
print(classification_report(y_true, y_pred_classes))

# Confusion matrix
print(confusion_matrix(y_true, y_pred_classes))
```

```
313/313              1s 1ms/step -
accuracy: 0.4525 - loss: 1.5387
313/313              1s 3ms/step
              precision    recall  f1-score   support

           0       0.45      0.55      0.50      1000
           1       0.58      0.56      0.57      1000
           2       0.36      0.17      0.23      1000
           3       0.30      0.35      0.32      1000
           4       0.42      0.32      0.36      1000
           5       0.41      0.32      0.36      1000
           6       0.48      0.52      0.50      1000
           7       0.43      0.60      0.50      1000
           8       0.50      0.67      0.57      1000
           9       0.52      0.46      0.49      1000

    accuracy                           0.45     10000
   macro avg       0.45      0.45      0.44     10000
weighted avg       0.45      0.45      0.44     10000
```

```
[[549  32  28  34  19  11  14  64 204  45]
 [ 68 559  11  42  14  12  13  31 110 140]
 [142  32 166 102 148  76 138 142  36  18]
 [ 46  28  46 350  55 164 106 118  46  41]
 [ 80  14  84  89 318  52 144 173  35  11]
 [ 40  20  56 212  45 317  86 145  53  26]
 [ 16  11  31 188  86  49 515  57  22  25]
 [ 70  23  21  58  58  70  24 596  27  53]
 [144  45   9  33   7  10   8  15 668  61]
 [ 55 192   5  47   2  17  15  59 146 462]]
```

How to Improve Performance:

Data Augmentation: Introduce variations in the data to reduce overfitting.

More Complex Architectures: Add more layers or filters to improve feature extraction.

#7. Optimization Strategies **Early Stopping**: Stop training when validation accuracy no longer improves.

Learning Rate Scheduling: Gradually decrease the learning rate to allow finer convergence.

Weight Initialization: Start with weights near zero, but not zero, to ensure symmetry breaking and efficient learning.

Weight Initialization Importance:

Poor initialization can cause vanishing/exploding gradients.

Techniques like He initialization for ReLU layers can help achieve faster convergence.