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

September 27, 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.

[ ]:
```
import tensorflow as tf
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.preprocessing.image import ImageDataGenerator
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)

# Optionally apply data augmentation
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
datagen.fit(x_train)
```

```
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)
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071            14s
0us/step
Training data shape (50000, 32, 32, 3)
Training labels shape (50000, 10)
Testing data shape (10000, 32, 32, 3)
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.

[ ]:
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatten, Dense,␣
↪Dropout

# Define the CNN model
model = Sequential()

# First Conv Layer
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Second Conv Layer
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten the feature map and feed it to a fully connected layer
model.add(Flatten())
model.add(Dense(128, activation='tanh'))

# Output Layer
```

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

model.summary()
```

```
/usr/local/lib/python3.10/dist-
packages/keras/src/layers/convolutional/base_conv.py:107: 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 #

 conv2d (Conv2D)                    (None, 30, 30, 32)                        ␣
↪896

 max_pooling2d (MaxPooling2D)       (None, 15, 15, 32)                        ␣
↪   0

 conv2d_1 (Conv2D)                  (None, 13, 13, 64)                     ␣
↪18,496

 max_pooling2d_1 (MaxPooling2D)     (None, 6, 6, 64)                          ␣
↪   0

 flatten (Flatten)                  (None, 2304)                              ␣
↪   0

 dense (Dense)                      (None, 128)                           ␣
↪295,040

 dense_1 (Dense)                    (None, 10)                              ␣
↪1,290
```

Total params: 315,722 (1.20 MB)

Trainable params: 315,722 (1.20 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.

[ ]:
```
# 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.

[ ]:
```
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.

[ ]:
```
# Train the model
history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    validation_data=(x_test, y_test),
                    epochs=50)
```

```
Epoch 1/50
```

```
/usr/local/lib/python3.10/dist-
packages/keras/src/trainers/data_adapters/py_dataset_adapter.py:121:
UserWarning: Your `PyDataset` class should call `super().__init__(**kwargs)` in
its constructor. `**kwargs` can include `workers`, `use_multiprocessing`,
`max_queue_size`. Do not pass these arguments to `fit()`, as they will be
ignored.
  self._warn_if_super_not_called()
```

```
782/782             43s 49ms/step -
accuracy: 0.3785 - loss: 1.7097 - val_accuracy: 0.5883 - val_loss: 1.1803
Epoch 2/50
782/782             35s 44ms/step -
accuracy: 0.5586 - loss: 1.2429 - val_accuracy: 0.6195 - val_loss: 1.0866
Epoch 3/50
782/782             36s 45ms/step -
accuracy: 0.5961 - loss: 1.1418 - val_accuracy: 0.6363 - val_loss: 1.0450
Epoch 4/50
782/782             40s 44ms/step -
accuracy: 0.6201 - loss: 1.0777 - val_accuracy: 0.6655 - val_loss: 0.9568
Epoch 5/50
782/782             41s 44ms/step -
accuracy: 0.6407 - loss: 1.0203 - val_accuracy: 0.6858 - val_loss: 0.9212
Epoch 6/50
782/782             41s 44ms/step -
accuracy: 0.6572 - loss: 0.9819 - val_accuracy: 0.6989 - val_loss: 0.8664
Epoch 7/50
782/782             40s 43ms/step -
accuracy: 0.6701 - loss: 0.9464 - val_accuracy: 0.7124 - val_loss: 0.8402
Epoch 8/50
782/782             43s 45ms/step -
accuracy: 0.6806 - loss: 0.9066 - val_accuracy: 0.7050 - val_loss: 0.8509
Epoch 9/50
782/782             34s 43ms/step -
accuracy: 0.6886 - loss: 0.8839 - val_accuracy: 0.7203 - val_loss: 0.8147
Epoch 10/50
782/782             41s 44ms/step -
accuracy: 0.6980 - loss: 0.8568 - val_accuracy: 0.7259 - val_loss: 0.7944
```

```
Epoch 11/50
782/782              33s 42ms/step -
accuracy: 0.7064 - loss: 0.8284 - val_accuracy: 0.7291 - val_loss: 0.7814
Epoch 12/50
782/782              41s 42ms/step -
accuracy: 0.7132 - loss: 0.8197 - val_accuracy: 0.7434 - val_loss: 0.7466
Epoch 13/50
782/782              41s 43ms/step -
accuracy: 0.7224 - loss: 0.7970 - val_accuracy: 0.7402 - val_loss: 0.7497
Epoch 14/50
782/782              33s 42ms/step -
accuracy: 0.7269 - loss: 0.7832 - val_accuracy: 0.7468 - val_loss: 0.7236
Epoch 15/50
782/782              35s 44ms/step -
accuracy: 0.7310 - loss: 0.7567 - val_accuracy: 0.7408 - val_loss: 0.7563
Epoch 16/50
782/782              41s 44ms/step -
accuracy: 0.7361 - loss: 0.7532 - val_accuracy: 0.7640 - val_loss: 0.6895
Epoch 17/50
782/782              34s 44ms/step -
accuracy: 0.7430 - loss: 0.7396 - val_accuracy: 0.7611 - val_loss: 0.6926
Epoch 18/50
782/782              34s 43ms/step -
accuracy: 0.7421 - loss: 0.7387 - val_accuracy: 0.7361 - val_loss: 0.7801
Epoch 19/50
782/782              42s 44ms/step -
accuracy: 0.7438 - loss: 0.7204 - val_accuracy: 0.7373 - val_loss: 0.7828
Epoch 20/50
782/782              33s 42ms/step -
accuracy: 0.7502 - loss: 0.7192 - val_accuracy: 0.7603 - val_loss: 0.7000
Epoch 21/50
782/782              42s 44ms/step -
accuracy: 0.7596 - loss: 0.6918 - val_accuracy: 0.7552 - val_loss: 0.7208
Epoch 22/50
782/782              33s 42ms/step -
accuracy: 0.7617 - loss: 0.6819 - val_accuracy: 0.7611 - val_loss: 0.6935
Epoch 23/50
782/782              42s 43ms/step -
accuracy: 0.7606 - loss: 0.6802 - val_accuracy: 0.7723 - val_loss: 0.6516
Epoch 24/50
782/782              33s 42ms/step -
accuracy: 0.7649 - loss: 0.6694 - val_accuracy: 0.7591 - val_loss: 0.7067
Epoch 25/50
782/782              40s 42ms/step -
accuracy: 0.7673 - loss: 0.6605 - val_accuracy: 0.7651 - val_loss: 0.7072
Epoch 26/50
782/782              42s 43ms/step -
accuracy: 0.7712 - loss: 0.6562 - val_accuracy: 0.7609 - val_loss: 0.7087
```

```
Epoch 27/50
782/782             33s 42ms/step -
accuracy: 0.7713 - loss: 0.6455 - val_accuracy: 0.7710 - val_loss: 0.6702
Epoch 28/50
782/782             34s 43ms/step -
accuracy: 0.7679 - loss: 0.6548 - val_accuracy: 0.7804 - val_loss: 0.6454
Epoch 29/50
782/782             41s 42ms/step -
accuracy: 0.7745 - loss: 0.6422 - val_accuracy: 0.7627 - val_loss: 0.6975
Epoch 30/50
782/782             34s 43ms/step -
accuracy: 0.7791 - loss: 0.6330 - val_accuracy: 0.7689 - val_loss: 0.6674
Epoch 31/50
782/782             33s 42ms/step -
accuracy: 0.7788 - loss: 0.6307 - val_accuracy: 0.7788 - val_loss: 0.6442
Epoch 32/50
782/782             36s 46ms/step -
accuracy: 0.7812 - loss: 0.6219 - val_accuracy: 0.7630 - val_loss: 0.7104
Epoch 33/50
782/782             39s 44ms/step -
accuracy: 0.7810 - loss: 0.6240 - val_accuracy: 0.7587 - val_loss: 0.7069
Epoch 34/50
782/782             35s 44ms/step -
accuracy: 0.7839 - loss: 0.6139 - val_accuracy: 0.7815 - val_loss: 0.6476
Epoch 35/50
782/782             40s 44ms/step -
accuracy: 0.7882 - loss: 0.6122 - val_accuracy: 0.7662 - val_loss: 0.6955
Epoch 36/50
782/782             35s 44ms/step -
accuracy: 0.7848 - loss: 0.6082 - val_accuracy: 0.7638 - val_loss: 0.6929
Epoch 37/50
782/782             42s 45ms/step -
accuracy: 0.7870 - loss: 0.6069 - val_accuracy: 0.7549 - val_loss: 0.7552
Epoch 38/50
782/782             41s 45ms/step -
accuracy: 0.7912 - loss: 0.5979 - val_accuracy: 0.7681 - val_loss: 0.6914
Epoch 39/50
782/782             34s 44ms/step -
accuracy: 0.7913 - loss: 0.5988 - val_accuracy: 0.7785 - val_loss: 0.6559
Epoch 40/50
782/782             34s 43ms/step -
accuracy: 0.7926 - loss: 0.5890 - val_accuracy: 0.7747 - val_loss: 0.6715
Epoch 41/50
782/782             35s 45ms/step -
accuracy: 0.7946 - loss: 0.5882 - val_accuracy: 0.7784 - val_loss: 0.6568
Epoch 42/50
782/782             35s 45ms/step -
accuracy: 0.7965 - loss: 0.5842 - val_accuracy: 0.7760 - val_loss: 0.6656
```

```
Epoch 43/50
782/782              33s 42ms/step -
accuracy: 0.7975 - loss: 0.5793 - val_accuracy: 0.7728 - val_loss: 0.6825
Epoch 44/50
782/782              35s 44ms/step -
accuracy: 0.7973 - loss: 0.5833 - val_accuracy: 0.7760 - val_loss: 0.6863
Epoch 45/50
782/782              34s 43ms/step -
accuracy: 0.7924 - loss: 0.5902 - val_accuracy: 0.7847 - val_loss: 0.6396
Epoch 46/50
782/782              41s 43ms/step -
accuracy: 0.7962 - loss: 0.5807 - val_accuracy: 0.7741 - val_loss: 0.6779
Epoch 47/50
782/782              41s 44ms/step -
accuracy: 0.7958 - loss: 0.5760 - val_accuracy: 0.7804 - val_loss: 0.6706
Epoch 48/50
782/782              35s 44ms/step -
accuracy: 0.8036 - loss: 0.5666 - val_accuracy: 0.7882 - val_loss: 0.6367
Epoch 49/50
782/782              34s 43ms/step -
accuracy: 0.8031 - loss: 0.5595 - val_accuracy: 0.7724 - val_loss: 0.6885
Epoch 50/50
782/782              42s 44ms/step -
accuracy: 0.8036 - loss: 0.5647 - val_accuracy: 0.7821 - val_loss: 0.6704
```

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.

[ ]:
```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))
```

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

```
313/313               1s 2ms/step -
accuracy: 0.7777 - loss: 0.6823
313/313               1s 2ms/step
              precision    recall  f1-score   support

           0       0.76      0.86      0.81      1000
           1       0.84      0.89      0.86      1000
           2       0.84      0.60      0.70      1000
           3       0.67      0.55      0.60      1000
           4       0.75      0.76      0.76      1000
           5       0.77      0.63      0.70      1000
           6       0.79      0.88      0.83      1000
           7       0.77      0.86      0.81      1000
           8       0.87      0.86      0.87      1000
           9       0.76      0.91      0.83      1000

    accuracy                           0.78     10000
   macro avg       0.78      0.78      0.78     10000
weighted avg       0.78      0.78      0.78     10000

[[861  30  14   3   7   1   4   7  34  39]
 [  9 892   2   2   1   1   1   2  12  78]
 [ 82  13 601  40  80  48  65  46   6  19]
 [ 29  24  38 550  63  93  77  56  27  43]
 [ 22   3  29  33 762  14  40  78  14   5]
 [ 22  12  13 139  39 634  42  60  13  26]
 [ 12   7   9  34  24   5 880   7  11  11]
 [ 14   8   8  19  31  26   4 864   5  21]
 [ 59  26   1   3   1   1   5   1 864  39]
 [ 16  50   2   2   2   0   2   4   9 913]]
```

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.