

vansh-152-lab3

September 27, 2024

```
[1]: # Import required libraries
import tensorflow as tf
from tensorflow.keras.datasets import cifar10 # Import CIFAR-10 dataset
      ↳ directly from Keras
from tensorflow.keras.utils import to_categorical # To convert labels to
      ↳ one-hot encoding
from tensorflow.keras.preprocessing.image import ImageDataGenerator # For
      ↳ image augmentation

# Load CIFAR-10 dataset
# CIFAR-10 dataset is divided into 50,000 training images and 10,000 test
      ↳ images.
# Each image has dimensions of 32x32 with 3 color channels (RGB), and there are
      ↳ 10 classes (e.g., airplanes, cars).
(x_train, y_train), (x_test, y_test) = cifar10.load_data()

# Normalize pixel values to range between 0 and 1
# Pixel values range from 0 to 255, so dividing by 255 scales them to be
      ↳ between 0 and 1, which helps neural networks train faster and more
      ↳ accurately.
x_train = x_train.astype('float32') / 255.0
x_test = x_test.astype('float32') / 255.0

# Convert class labels to one-hot encoded format
# Each class label is converted to a binary vector with 1 for the true class
      ↳ and 0 for others.
# Example: If a class label is 3, it becomes [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]
      ↳ (for 10 classes).
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# Optionally apply data augmentation
# Data augmentation is a technique to artificially increase the size of the
      ↳ training set by applying random transformations like flipping and rotating
      ↳ images.
# This helps the model generalize better by seeing slightly altered versions of
      ↳ the original images.
```

```

datagen = ImageDataGenerator(
    horizontal_flip=True,          # Randomly flip the images horizontally
    rotation_range=15,            # Rotate images by up to 15 degrees
    width_shift_range=0.1,        # Shift images horizontally by up to 10% of
    ↪the width
    height_shift_range=0.1        # Shift images vertically by up to 10% of the
    ↪height
)

# Apply the augmentations only to the training data (not test data).
datagen.fit(x_train)

```

Downloading data from <https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz>
 170498071/170498071 42s
 Ous/step

```

[2]: # Import necessary libraries from Keras
from tensorflow.keras.models import Sequential # Sequential model allows us to
    ↪build a model layer by layer
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D #
    ↪Import different types of layers to build the model

# Build the feedforward neural network using the Sequential API
model = Sequential()

# First convolutional layer
# Conv2D creates a convolutional layer with 32 filters of size 3x3, using ReLU
    ↪activation function.
# The input_shape parameter specifies the dimensions of the input images (32x32
    ↪pixels, 3 color channels).
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))

# First max pooling layer
# MaxPooling2D reduces the spatial dimensions of the feature maps (outputs of
    ↪Conv2D) by taking the maximum value over a 2x2 pool size.
# This helps to down-sample the feature maps and reduce the number of
    ↪parameters.
model.add(MaxPooling2D(pool_size=(2, 2)))

# Second convolutional layer
# Adding another convolutional layer with 64 filters to learn more complex
    ↪features.
# The filters will convolve over the output of the previous layer.
model.add(Conv2D(64, (3, 3), activation='relu'))

# Second max pooling layer

```

```

# Again down-sampling the feature maps to reduce the spatial dimensions.
model.add(MaxPooling2D(pool_size=(2, 2)))

# Third convolutional layer
# Adding a third convolutional layer with 128 filters for even more complex
    ↪ feature extraction.
model.add(Conv2D(128, (3, 3), activation='relu'))

# Third max pooling layer
# Down-sampling the feature maps again.
model.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten layer
# Flatten converts the 3D output from the previous layer (height, width,
    ↪ filters) into a 1D vector.
# This step is necessary before passing the data to fully connected (Dense)
    ↪ layers.
model.add(Flatten())

# Fully connected layer
# Dense layer with 256 neurons. The ReLU activation function is applied to
    ↪ introduce non-linearity.
model.add(Dense(256, activation='relu'))

# Output layer
# The output layer has 10 neurons (one for each class) with a softmax
    ↪ activation function.
# Softmax is used for multi-class classification, converting raw output values
    ↪ into probabilities that sum to 1.
model.add(Dense(10, activation='softmax'))

# Compile the model
# The compile method configures the model for training.
# We use the Adam optimizer, categorical_crossentropy as the loss function
    ↪ (suitable for multi-class classification),
# and we want to monitor accuracy during training.
model.compile(optimizer='adam', loss='categorical_crossentropy',
    ↪ metrics=['accuracy'])

```

/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-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)
```

```
[7]: # Import necessary libraries from Keras
from tensorflow.keras.models import Sequential # Sequential model allows us to
    ↳ build a model layer by layer
from tensorflow.keras.layers import Dense, Flatten, Conv2D, MaxPooling2D #
    ↳ Import different types of layers to build the model

# Initialize the Sequential model
model = Sequential()

# First convolutional layer
# Conv2D creates a convolutional layer with 32 filters of size 3x3.
# The ReLU (Rectified Linear Unit) activation function is used to introduce
    ↳ non-linearity.
# The input_shape parameter specifies the dimensions of the input images: (32
    ↳ pixels height, 32 pixels width, 3 color channels).
model.add(Conv2D(32, (3, 3), activation='relu', input_shape=(32, 32, 3)))

# First max pooling layer
# MaxPooling2D reduces the spatial dimensions of the feature maps produced by
    ↳ the previous layer.
# Here, we use a pool size of 2x2, which means we take the maximum value from
    ↳ every 2x2 block of pixels.
model.add(MaxPooling2D(pool_size=(2, 2)))

# Second convolutional layer
# Adding another convolutional layer with 64 filters to learn more complex
    ↳ features from the input images.
# The filters will convolve over the output of the previous layer.
model.add(Conv2D(64, (3, 3), activation='relu'))

# Second max pooling layer
# Again down-sampling the feature maps to reduce their size and complexity.
model.add(MaxPooling2D(pool_size=(2, 2)))

# Third convolutional layer
# Adding a third convolutional layer with 128 filters for extracting even more
    ↳ complex features.
model.add(Conv2D(128, (3, 3), activation='relu'))

# Third max pooling layer
# Another down-sampling step to further reduce the spatial dimensions of the
    ↳ feature maps.
model.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten layer
```

```

# Flatten converts the 3D output from the last max pooling layer (height,
    width, channels) into a 1D vector.
# This transformation is necessary to connect the convolutional part of the
    network to the fully connected part.
model.add(Flatten())

# Fully connected layer
# Dense layer with 256 neurons. The ReLU activation function is applied to
    introduce non-linearity.
# This layer will learn to combine the features extracted by the convolutional
    layers.
model.add(Dense(256, activation='relu'))

# Output layer
# The output layer has 10 neurons, corresponding to the 10 classes in the
    CIFAR-10 dataset.
# The softmax activation function is used to convert the raw output values into
    probabilities for each class.
model.add(Dense(10, activation='softmax'))

# Compile the model
# The compile method configures the model for training.
# We use the Adam optimizer, which adjusts the learning rate dynamically and is
    generally effective for deep learning tasks.
# Categorical crossentropy is used as the loss function, which is appropriate
    for multi-class classification problems.
# We also specify to track accuracy during training and evaluation.
model.compile(optimizer='adam', loss='categorical_crossentropy',
    metrics=['accuracy'])

```

/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-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)
```

```

[4]: # Import the SGD optimizer from Keras
from tensorflow.keras.optimizers import SGD

# Initialize the SGD optimizer
# Setting the learning rate to 0.01. The learning rate controls how much to
    change the model in response to the estimated error each time the model
    weights are updated.
# The momentum parameter is set to 0.9, which helps accelerate SGD in the
    relevant direction and dampens oscillations.

```

```
sgd = SGD(learning_rate=0.01, momentum=0.9)

# Compile the model using the SGD optimizer
# In this step, we configure the model for training. We specify the optimizer
↳ (SGD), the loss function (categorical_crossentropy), and metrics to track
↳ (accuracy).
model.compile(optimizer=sgd, loss='categorical_crossentropy',
↳ metrics=['accuracy'])

# Check if the model is not converging (i.e., not improving accuracy)
# If the model's performance is not improving (e.g., accuracy stagnating), we
↳ may need to adjust the learning rate.
# In this case, we are reducing the learning rate to 0.001 to make smaller
↳ adjustments to the weights during training.
model.optimizer.learning_rate = 0.001
```

```
[8]: # Train the model using backpropagation
# The fit method is used to train the model on the training data (x_train,
↳ y_train) for a specified number of epochs.
# datagen.flow(x_train, y_train, batch_size=64) generates batches of augmented
↳ image data for training.
# Validation data (x_test, y_test) is provided to evaluate the model's
↳ performance on unseen data after each epoch.
history = model.fit(datagen.flow(x_train, y_train, batch_size=64),
                    validation_data=(x_test, y_test),
                    epochs=50, # Number of times to iterate over the training
↳ dataset
                    verbose=1) # Verbosity mode; 1 means progress will be
↳ displayed during training
```

Epoch 1/50

4/782 13s 17ms/step - accuracy:

0.1100 - loss: 2.3135

/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-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 12s 15ms/step -

accuracy: 0.2987 - loss: 1.8766 - val_accuracy: 0.4723 - val_loss: 1.4941

Epoch 2/50

782/782 12s 15ms/step -

accuracy: 0.5035 - loss: 1.3743 - val_accuracy: 0.5834 - val_loss: 1.1530

Epoch 3/50

782/782 12s 15ms/step -
 accuracy: 0.5611 - loss: 1.2298 - val_accuracy: 0.6120 - val_loss: 1.1126
 Epoch 4/50
 782/782 14s 18ms/step -
 accuracy: 0.6029 - loss: 1.1204 - val_accuracy: 0.6491 - val_loss: 0.9990
 Epoch 5/50
 782/782 12s 16ms/step -
 accuracy: 0.6305 - loss: 1.0501 - val_accuracy: 0.6567 - val_loss: 0.9604
 Epoch 6/50
 782/782 12s 16ms/step -
 accuracy: 0.6433 - loss: 1.0107 - val_accuracy: 0.6647 - val_loss: 0.9656
 Epoch 7/50
 782/782 12s 16ms/step -
 accuracy: 0.6650 - loss: 0.9582 - val_accuracy: 0.6625 - val_loss: 0.9636
 Epoch 8/50
 782/782 13s 16ms/step -
 accuracy: 0.6798 - loss: 0.9159 - val_accuracy: 0.6918 - val_loss: 0.8764
 Epoch 9/50
 782/782 13s 16ms/step -
 accuracy: 0.6842 - loss: 0.8914 - val_accuracy: 0.7042 - val_loss: 0.8570
 Epoch 10/50
 782/782 13s 17ms/step -
 accuracy: 0.6939 - loss: 0.8727 - val_accuracy: 0.7076 - val_loss: 0.8277
 Epoch 11/50
 782/782 14s 17ms/step -
 accuracy: 0.7039 - loss: 0.8445 - val_accuracy: 0.6978 - val_loss: 0.8660
 Epoch 12/50
 782/782 14s 17ms/step -
 accuracy: 0.7105 - loss: 0.8244 - val_accuracy: 0.7135 - val_loss: 0.8169
 Epoch 13/50
 782/782 12s 16ms/step -
 accuracy: 0.7168 - loss: 0.8080 - val_accuracy: 0.7270 - val_loss: 0.7841
 Epoch 14/50
 782/782 12s 16ms/step -
 accuracy: 0.7279 - loss: 0.7758 - val_accuracy: 0.7332 - val_loss: 0.7637
 Epoch 15/50
 782/782 12s 16ms/step -
 accuracy: 0.7333 - loss: 0.7645 - val_accuracy: 0.7253 - val_loss: 0.7898
 Epoch 16/50
 782/782 12s 16ms/step -
 accuracy: 0.7345 - loss: 0.7507 - val_accuracy: 0.7463 - val_loss: 0.7280
 Epoch 17/50
 782/782 12s 16ms/step -
 accuracy: 0.7387 - loss: 0.7421 - val_accuracy: 0.7452 - val_loss: 0.7366
 Epoch 18/50
 782/782 13s 16ms/step -
 accuracy: 0.7443 - loss: 0.7316 - val_accuracy: 0.7484 - val_loss: 0.7403
 Epoch 19/50

782/782 13s 16ms/step -
 accuracy: 0.7495 - loss: 0.7156 - val_accuracy: 0.7491 - val_loss: 0.7348
 Epoch 20/50
 782/782 12s 16ms/step -
 accuracy: 0.7503 - loss: 0.7124 - val_accuracy: 0.7520 - val_loss: 0.7329
 Epoch 21/50
 782/782 13s 17ms/step -
 accuracy: 0.7546 - loss: 0.6996 - val_accuracy: 0.7524 - val_loss: 0.7145
 Epoch 22/50
 782/782 14s 18ms/step -
 accuracy: 0.7529 - loss: 0.6961 - val_accuracy: 0.7539 - val_loss: 0.7091
 Epoch 23/50
 782/782 14s 17ms/step -
 accuracy: 0.7585 - loss: 0.6834 - val_accuracy: 0.7581 - val_loss: 0.7275
 Epoch 24/50
 782/782 14s 18ms/step -
 accuracy: 0.7651 - loss: 0.6682 - val_accuracy: 0.7584 - val_loss: 0.7038
 Epoch 25/50
 782/782 14s 18ms/step -
 accuracy: 0.7634 - loss: 0.6695 - val_accuracy: 0.7484 - val_loss: 0.7485
 Epoch 26/50
 782/782 15s 19ms/step -
 accuracy: 0.7688 - loss: 0.6557 - val_accuracy: 0.7368 - val_loss: 0.8017
 Epoch 27/50
 782/782 14s 17ms/step -
 accuracy: 0.7727 - loss: 0.6496 - val_accuracy: 0.7586 - val_loss: 0.7049
 Epoch 28/50
 782/782 13s 16ms/step -
 accuracy: 0.7744 - loss: 0.6396 - val_accuracy: 0.7692 - val_loss: 0.6879
 Epoch 29/50
 782/782 14s 18ms/step -
 accuracy: 0.7735 - loss: 0.6434 - val_accuracy: 0.7592 - val_loss: 0.7296
 Epoch 30/50
 782/782 14s 18ms/step -
 accuracy: 0.7777 - loss: 0.6371 - val_accuracy: 0.7818 - val_loss: 0.6368
 Epoch 31/50
 782/782 15s 19ms/step -
 accuracy: 0.7822 - loss: 0.6165 - val_accuracy: 0.7618 - val_loss: 0.7105
 Epoch 32/50
 782/782 14s 17ms/step -
 accuracy: 0.7825 - loss: 0.6158 - val_accuracy: 0.7783 - val_loss: 0.6645
 Epoch 33/50
 782/782 15s 19ms/step -
 accuracy: 0.7825 - loss: 0.6186 - val_accuracy: 0.7520 - val_loss: 0.7572
 Epoch 34/50
 782/782 16s 21ms/step -
 accuracy: 0.7839 - loss: 0.6191 - val_accuracy: 0.7854 - val_loss: 0.6560
 Epoch 35/50

782/782 16s 20ms/step -
 accuracy: 0.7878 - loss: 0.5984 - val_accuracy: 0.7697 - val_loss: 0.7129
 Epoch 36/50
 782/782 16s 20ms/step -
 accuracy: 0.7902 - loss: 0.6030 - val_accuracy: 0.7710 - val_loss: 0.7058
 Epoch 37/50
 782/782 15s 20ms/step -
 accuracy: 0.7931 - loss: 0.6000 - val_accuracy: 0.7715 - val_loss: 0.6903
 Epoch 38/50
 782/782 15s 19ms/step -
 accuracy: 0.7902 - loss: 0.6006 - val_accuracy: 0.7822 - val_loss: 0.6699
 Epoch 39/50
 782/782 15s 19ms/step -
 accuracy: 0.7934 - loss: 0.5901 - val_accuracy: 0.7756 - val_loss: 0.6824
 Epoch 40/50
 782/782 15s 19ms/step -
 accuracy: 0.7951 - loss: 0.5829 - val_accuracy: 0.7731 - val_loss: 0.6983
 Epoch 41/50
 782/782 15s 19ms/step -
 accuracy: 0.7977 - loss: 0.5821 - val_accuracy: 0.7798 - val_loss: 0.6606
 Epoch 42/50
 782/782 15s 19ms/step -
 accuracy: 0.7977 - loss: 0.5783 - val_accuracy: 0.7805 - val_loss: 0.6820
 Epoch 43/50
 782/782 15s 19ms/step -
 accuracy: 0.8008 - loss: 0.5700 - val_accuracy: 0.7719 - val_loss: 0.7112
 Epoch 44/50
 782/782 15s 19ms/step -
 accuracy: 0.7998 - loss: 0.5789 - val_accuracy: 0.7780 - val_loss: 0.7005
 Epoch 45/50
 782/782 15s 19ms/step -
 accuracy: 0.8049 - loss: 0.5590 - val_accuracy: 0.7517 - val_loss: 0.7578
 Epoch 46/50
 782/782 15s 19ms/step -
 accuracy: 0.8035 - loss: 0.5635 - val_accuracy: 0.7794 - val_loss: 0.6748
 Epoch 47/50
 782/782 15s 19ms/step -
 accuracy: 0.8059 - loss: 0.5585 - val_accuracy: 0.7800 - val_loss: 0.6784
 Epoch 48/50
 782/782 16s 20ms/step -
 accuracy: 0.8044 - loss: 0.5622 - val_accuracy: 0.7840 - val_loss: 0.6617
 Epoch 49/50
 782/782 15s 20ms/step -
 accuracy: 0.8056 - loss: 0.5519 - val_accuracy: 0.7856 - val_loss: 0.6650
 Epoch 50/50
 782/782 15s 19ms/step -
 accuracy: 0.8067 - loss: 0.5543 - val_accuracy: 0.7843 - val_loss: 0.6625

```
[9]: # Evaluate the model on the test set
# The evaluate method computes the loss and accuracy of the model on the test
# data (x_test, y_test).
# It returns the test loss and test accuracy, which provide a measure of how
# well the model performs on unseen data.
test_loss, test_acc = model.evaluate(x_test, y_test, verbose=2)

# Import necessary libraries for performance evaluation
from sklearn.metrics import classification_report, confusion_matrix
import numpy as np

# Predicting on test data
# The predict method generates output predictions for the input samples in
# x_test.
y_pred = model.predict(x_test)

# Convert the predicted probabilities to class labels
# np.argmax() returns the indices of the maximum values along an axis.
# Here, we get the predicted class indices by finding the index of the highest
# predicted probability.
y_pred_classes = np.argmax(y_pred, axis=1)

# Convert the true labels from one-hot encoded format to class labels
y_true = np.argmax(y_test, axis=1)

# Confusion matrix and classification report
# The confusion matrix shows the number of correct and incorrect predictions
# for each class.
print(confusion_matrix(y_true, y_pred_classes))

# The classification report provides a summary of various classification
# metrics, including precision, recall, and F1-score for each class.
print(classification_report(y_true, y_pred_classes))
```

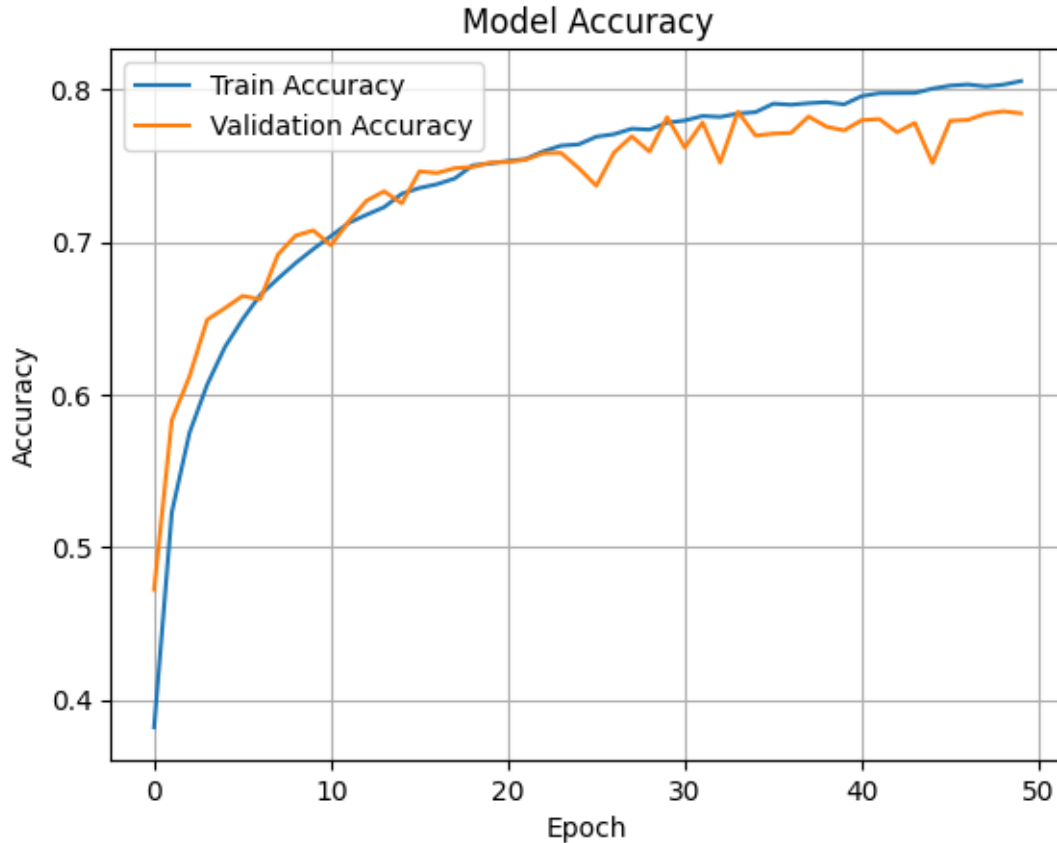
313/313 - 1s - 2ms/step - accuracy: 0.7843 - loss: 0.6625

```
313/313          1s 3ms/step
[[837  14  11  13   8   8   9  22  36  42]
 [  4 895   3   1   0   1   3   3   9  81]
 [ 57  11 617  49  43  79  63  61   6  14]
 [ 23  11  29 533  33 192  65  61  16  37]
 [ 13   4  34  43 662  43  74 115   4   8]
 [ 12   6  11  79  24 771  20  62   1  14]
 [  6   5  19  38   7  12 897   6   3   7]
 [ 13   7   8  10  16  36   8 881   0  21]
 [ 48  48   5   5   2   5   1   5 828  53]
 [ 18  30   1   1   1   4   3  10  10 922]]
          precision    recall  f1-score   support
```

0	0.81	0.84	0.82	1000
1	0.87	0.90	0.88	1000
2	0.84	0.62	0.71	1000
3	0.69	0.53	0.60	1000
4	0.83	0.66	0.74	1000
5	0.67	0.77	0.72	1000
6	0.78	0.90	0.84	1000
7	0.72	0.88	0.79	1000
8	0.91	0.83	0.87	1000
9	0.77	0.92	0.84	1000
accuracy			0.78	10000
macro avg	0.79	0.78	0.78	10000
weighted avg	0.79	0.78	0.78	10000

```
[11]: import matplotlib.pyplot as plt

# Plot training & validation accuracy values
plt.plot(history.history['accuracy'], label='Train Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(loc='upper left')
plt.grid(True)
plt.show()
```



1. **Justification for Network Architecture:** The chosen network architecture includes three convolutional layers with 32, 64, and 128 filters respectively, followed by max-pooling layers. This setup helps in gradually learning more complex features at each level. The architecture ends with fully connected (dense) layers to consolidate the learned features and output predictions for the 10 CIFAR-10 classes. The number of neurons in the final dense layer (256) is chosen to balance model complexity and generalization. The softmax output layer with 10 neurons is appropriate for multi-class classification tasks like CIFAR-10.
2. **Impact of Optimizer and Learning Rate on Convergence:** The choice of optimizer and learning rate plays a critical role in the model's convergence. The Adam optimizer is used, which dynamically adjusts the learning rate to converge faster. The learning rate determines how much the model adjusts weights after each iteration. If the model isn't converging properly, you can lower the learning rate, as done in the document where the learning rate was reduced from 0.01 to 0.001, allowing for smaller, more stable weight updates.
3. **Backpropagation and Learning Rate:** Backpropagation updates the weights by computing the gradient of the loss function with respect to the weights. The learning rate controls how much to update these weights during each iteration. A larger learning rate may cause the model to overshoot optimal values, while a smaller rate can lead to slower convergence.
4. **Improving Low Model Accuracy:** To improve low accuracy, you can try various techniques such as:

- Implementing data augmentation (e.g., random flipping, rotating) to improve generalization.
 - Increasing the complexity of the model by adding more layers or neurons.
 - Experimenting with different optimizers or tuning hyperparameters such as learning rate.
 - Reducing overfitting by using dropout layer.
5. **Importance of Weight Initialization:** Weight initialization is crucial because poor initialization can lead to slow convergence or the model getting stuck in local minima. Proper initialization ensures that the model starts with reasonable weight values, allowing the gradients to propagate effectively during backpropagation.

0.0.1 Project Report: CIFAR-10 Image Classification

Hyperparameters Used:

- **Learning Rate:** Initially 0.01, reduced to 0.001 for better convergence.
- **Batch Size:** 64
- **Number of Epochs:** 50
- **Optimizer:** Adam and SGD (for experimentation)
- **Loss Function:** Categorical Crossentropy
- **Model Architecture:**
 - Convolutional layers: 32, 64, 128 filters
 - Dense layers: 256 neurons
 - Output Layer: 10 neurons (softmax activation)

Challenges Faced & Solutions:

1. **Overfitting:** The model showed signs of overfitting, which was mitigated by applying data augmentation techniques like random flipping and rotation.
2. **Non-convergence:** The model initially struggled with convergence. Lowering the learning rate from 0.01 to 0.001 improved stability and helped the model converge smoothly.
3. **Training Time:** To reduce the training time, batch size and the number of epochs were adjusted for faster iterations without sacrificing accuracy.

This report briefly covers key hyperparameters and addresses challenges faced during model training and tuning.