Development of an image classification algorithm for CIFAR-10 dataset

Priyanka N Silveri

Introduction

CIFAR-10 is an image dataset consisting of 60,000 images of 32x32 pixel resolution composed of 10 classes. They are divided into 50,000 training images and 10,000 test images. The goal of this project is to develop a convolutional neural network (CNN) capable of accurately classifying an image from the dataset.

The training dataset, consisting of 50,000 samples, is split into training images and validation images. Since the given test dataset contains 10,000 images, the training set is set to 40,000 and the validation set is set to 10,000 to match the test set.

Since this is a large dataset, and the model randomly chooses the weights, it may take a while to reach an optimal solution. Hence, a large epoch is used to ensure the model sees the training data multiple times to reach the best solution, and multiple learning rates are used to see how quickly the model converges.

The model summary will offer us insight to the computational complexity of the models. Based on the complexity of the project, the models will be designed to only have a manageable number of trainable parameters (max: 100,000).

Convolutional Neural Network (CNN)

Designing the Model

Below is a snippet of the code that designs the CNN that conducts multi-class classification.

```python
# CNN Model
from functools import partial
DefaultConv2D = partial(keras.layers.Conv2D, kernel_size=3,
activation='relu',padding="SAME")
model = keras.models.Sequential([
  DefaultConv2D(filters=16, kernel_size=7, input_shape=[32, 32, 3]),
  keras.layers.BatchNormalization(),
  DefaultConv2D(filters=16),
  keras.layers.BatchNormalization(),
  keras.layers.MaxPooling2D(pool_size=2),
  DefaultConv2D(filters=32),
  keras.layers.BatchNormalization(),
  DefaultConv2D(filters=32),
  keras.layers.BatchNormalization(),
  keras.layers.MaxPooling2D(pool_size=2),
  DefaultConv2D(filters=64),
  keras.layers.BatchNormalization(),
  keras.layers.MaxPooling2D(pool_size=2),
  keras.layers.Flatten(),
  keras.layers.Dense(units=32, activation='relu'),
  keras.layers.Dropout(0.5),
  keras.layers.Dense(units=16, activation='relu'),
  keras.layers.Dropout(0.5),
  keras.layers.Dense(units=10, activation='softmax'),
  ])
```

In this code, we start by using the partial() function to define a thin wrapper around the Conv2D class, called DefaultConv2Din order to avoid repeating the same hyperparameter values. With a **3x3 kernel**, a **ReLu activation function** with “SAME” padding. “SAME” padding adds an outer dimension of 0’s to the input array before running the kernel through the input.

The first layer uses a kernel size of 16, but no stride because the input images are not very large. It also sets input_shape= [32, 32, 3], which means the images are 32× 32 pixels, with 3 channels (i.e. RGB). Next, we normalize the values in the batch, then repeat those two processes.

before adding a max pooling layer, which divides each spatial dimension by a factor of two (since pool_size=2).

Then we repeat the same structure three times: three convolutional layers, normalizing the outputs after the layer, followed by a max pooling layer. Note that the number of filters grows as we climb up the CNN towards the output layer (it is initially 16, then 64).

Next is an MLP. After flattening the output of the 5th layer, there are two dense layers with 32 and 16 neurons. To reduce overfitting, drop 50% of the data after every dense layer.

Finally, we add a Dense output layer with 10 neurons (one per class), using the softmax activation function (because the classes are exclusive).

In summary, the designed CNN has 8 layers: one to reshape the inputs to a 32x32 size with 3 channels, followed by 4 with growing filter size and batch normalization after each convolutional layer, then 2 dense layers also of increasing neuron size each using ReLu activations, lastly followed by one output layer of 10 neurons for classification using softmax activation due to the exclusivity of the classes.

Compiling and Training

The CNN is now fitted to the training data, using the validation data. I chose a learning rate of 0.001 because I wanted the weights to be updated slowly. I also initialized the epochs to be 100 to make sure the CNN adequately learns the model even with such a low learning rate.

CNN uses “nadam” as the optimizer. The name "Nadam" is derived from the combination of two other optimizers, namely "Nesterov accelerated gradient" (NAG) and "Adaptive Moment Estimation" (Adam).

Nadam optimizer is a variant of the Adam optimizer and incorporates the NAG method to update the parameters. NAG is a method that helps accelerate the convergence of the gradient descent algorithm by estimating the optimal next step in the direction of the gradient. Adam optimizer, on the other hand, uses the concept of moving averages of past gradients to update the parameters.

```
opt = tf.keras.optimizers.Nadam(learning_rate=0.001)
model.compile(loss="sparse_categorical_crossentropy",
optimizer=opt,  metrics=["accuracy"])
#early_stopping = keras.callbacks.EarlyStopping(patience=5, min_delta=0.01,
restore_best_weights=True)
history = model.fit(train_X, train_y, epochs=100, validation_data=(valid_X,
valid_y))
```

Evaluation and Summary

The following code outputs a 2D array, with the final loss value and accuracy score after the total number of epochs is run to give insight to how good the model is at classification.

```
model.evaluate(test_X, test_y)
313/313 [==============================] - 1s 3ms/step - loss: 1.4629 - accuracy: 0.6877
[1.4628902673721313, 0.6876999735832214]
```

Hence this CNN model is 68.76% accurate with a loss of 1.46, which is not that bad score.

To find the number of trainable parameters used, we invoke the following line of code(and present the output).

```
model.summary()
```

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d (Conv2D)             (None, 32, 32, 16)        2368

 batch_normalization (Batch  (None, 32, 32, 16)        64
 Normalization)

 conv2d_1 (Conv2D)           (None, 32, 32, 16)        2320

 batch_normalization_1 (Bat  (None, 32, 32, 16)        64
 chNormalization)

 max_pooling2d (MaxPooling2  (None, 16, 16, 16)        0
 D)

 conv2d_2 (Conv2D)           (None, 16, 16, 32)        4640

 batch_normalization_2 (Bat  (None, 16, 16, 32)        128
 chNormalization)

 conv2d_3 (Conv2D)           (None, 16, 16, 32)        9248

 batch_normalization_3 (Bat  (None, 16, 16, 32)        128
 chNormalization)

 max_pooling2d_1 (MaxPoolin  (None, 8, 8, 32)          0
 g2D)

 conv2d_4 (Conv2D)           (None, 8, 8, 64)          18496

 batch_normalization_4 (Bat  (None, 8, 8, 64)          256
 chNormalization)

 max_pooling2d_2 (MaxPoolin  (None, 4, 4, 64)          0
 g2D)
 flatten (Flatten)           (None, 1024)              0

 dense (Dense)               (None, 32)                32800

 dropout (Dropout)           (None, 32)                0

 dense_1 (Dense)             (None, 16)                528

 dropout_1 (Dropout)         (None, 16)                0

 dense_2 (Dense)             (None, 10)                170

=================================================================
Total params: 71210 (278.16 KB)
Trainable params: 70890 (276.91 KB)
Non-trainable params: 320 (1.25 KB)
_________________________________________________________________
```

Data Augmentation

To make our model much more accurate we can add data augmentation on our data and then train it again. Calling model.fit() again on augmented data will continue training where it left off, after 100 epochs. The data is now going to be fitted with a batch size of 32, stretch by a factor of 0.05 (width and height), rotation, zoom, data format and flip the images horizontally, enhancing the model's ability to generalize to unseen data. Then call model.fit again for 100 epochs.

```
batch_size = 32
data_generator = tf.keras.preprocessing.image.ImageDataGenerator(
            rotation_range=10,
            zoom_range = 0.05,
            width_shift_range=0.05,
            height_shift_range=0.05,
            horizontal_flip=True,
            vertical_flip=False,
            data_format="channels_last"
)

# data augmenting by expanding by a factor of 0.05 and fliping it upside
down
train_generator = data_generator.flow(train_X, train_y, batch_size)
steps_per_epoch = train_X.shape[0] // batch_size

data_augmented = model.fit(train_generator,
validation_data=(test_X,test_y), epochs=100)
```

rotation_range=10 : rotate images by up to 10 degrees, enhancing the model's ability to generalize to variations in object orientation.
zoom_range=0.05 : random zooming in or out of images by up to 5%, enabling the model to learn features at different scales.
width_shift_range=0.05 : It shifts images horizontally by up to 5%, introducing translations and aiding in invariant feature learning.
height_shift_range=0.05 : It shifts images vertically by up to 5%, diversifying the dataset and improving the model's robustness to positional changes.
horizontal_flip=True : It flips images horizontally, the model's ability to learn invariant features to left-right orientation changes.

The resulting evaluation array is as follows with an accuracy of **71.48%,** which indeed is an improvement on the model.

```
test_loss_aug, test_accuracy_aug = model.evaluate(test_X, test_y,
verbose=2)
313/313 - 1s - loss: 0.9627 - accuracy: 0.7148 - 835ms/epoch - 3ms/step
Test Loss after augmentation: 0.9627256989479065
Test Accuracy after augmentation: 0.7148000001907349
```

<u>Learning Curves</u>

The following are the training and testing accuracy of the CNN after data augmentation. Also, our code prints test loss and test accuracy after evaluation on test data.

```
test_loss_aug, test_accuracy_aug = model.evaluate(test_X, test_y,
verbose=2)

print(f'Test Loss after augmentation: {test_loss_aug}')
print(f'Test Accuracy after augmentation: {test_accuracy_aug}')

model.evaluate(test_X, test_y)
plt.plot(data_augmented.history['accuracy'], label='acc', color='red')
plt.plot(data_augmented.history['val_accuracy'], label='val_acc',
color='green')
plt.legend()
```

```
313/313 - 1s - loss: 0.9627 - accuracy: 0.7148 - 835ms/epoch - 3ms/step
Test Loss after augmentation: 0.9627256989479065
Test Accuracy after augmentation: 0.7148000001907349
<matplotlib.legend.Legend at 0x7e78ec41e260>
```

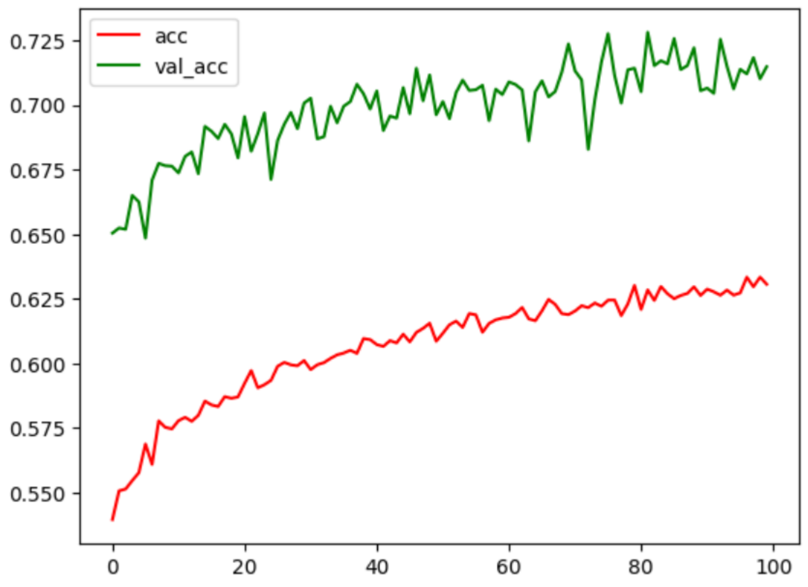


Fig 1: Performance Metrics and Training History: Test Loss and Accuracy After Data Augmentation

Evaluation on test data after augmentation:

```
model.evaluate(test_X, test_y)
313/313 [==============================] - 1s 3ms/step - loss: 0.9627 - accuracy: 0.7148
[0.9627256989479065, 0.7148000001907349]
```

Next is the cross-entropy loss over time, as well as the classification accuracy over the final 100 epochs. The plot shows the training and validation loss over epochs to visualize the model's performance during training.

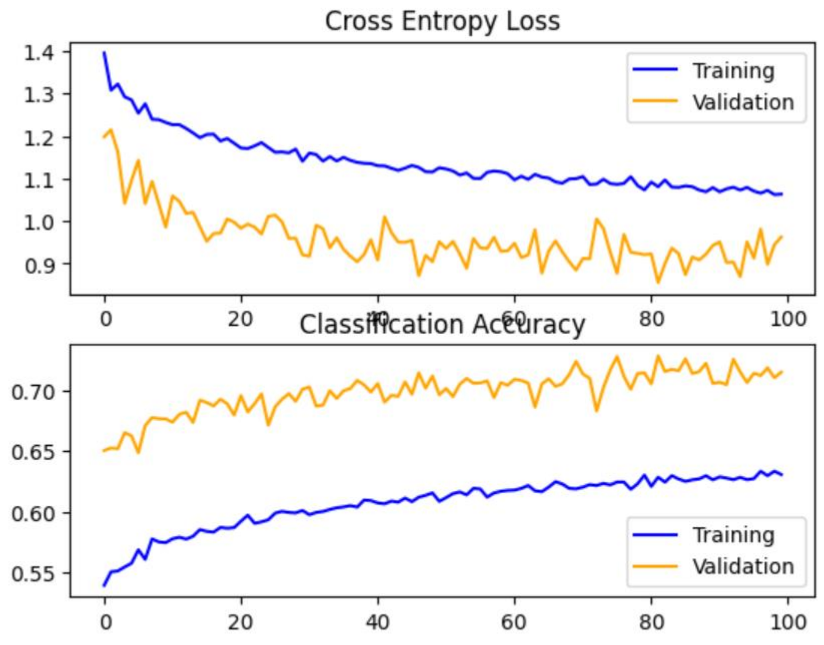


Fig 2: Training Progress: Loss and Accuracy Visualization

Validation Process

In the code, the validation process is implemented using the validation_data parameter in the fit() function during model training. The validation_data parameter is set to a tuple (valid_X, valid_y), where valid_X represents the validation images and valid_y represents the corresponding labels. During training, after each epoch, the model evaluates its performance on the validation dataset by computing the loss and accuracy metrics. This helps monitor how well the model generalizes to unseen data and whether it's overfitting. While the code does not include automated early stopping, we can manually monitor the validation loss values printed during training.

I have tried to validate images from the validation set. The predictions are made using the trained model, and we compare the predicted labels with the actual labels to assess the model's performance. We use the trained model (model) to predict labels for the images in the validation set (valid_X). The predictions are stored in val_predictions. The actual labels (valid_y) are initially in the shape (10000, 1). To compare them with the predicted labels, we flatten them to shape (10000,) using the flatten() method and store them in valid_y_flat.

The plot_images function is defined to plot images along with their predicted and actual labels. we call the plot_images function to plot the first 20 images from the validation dataset (valid_X). Each image is accompanied by its true label and the label predicted by the model.

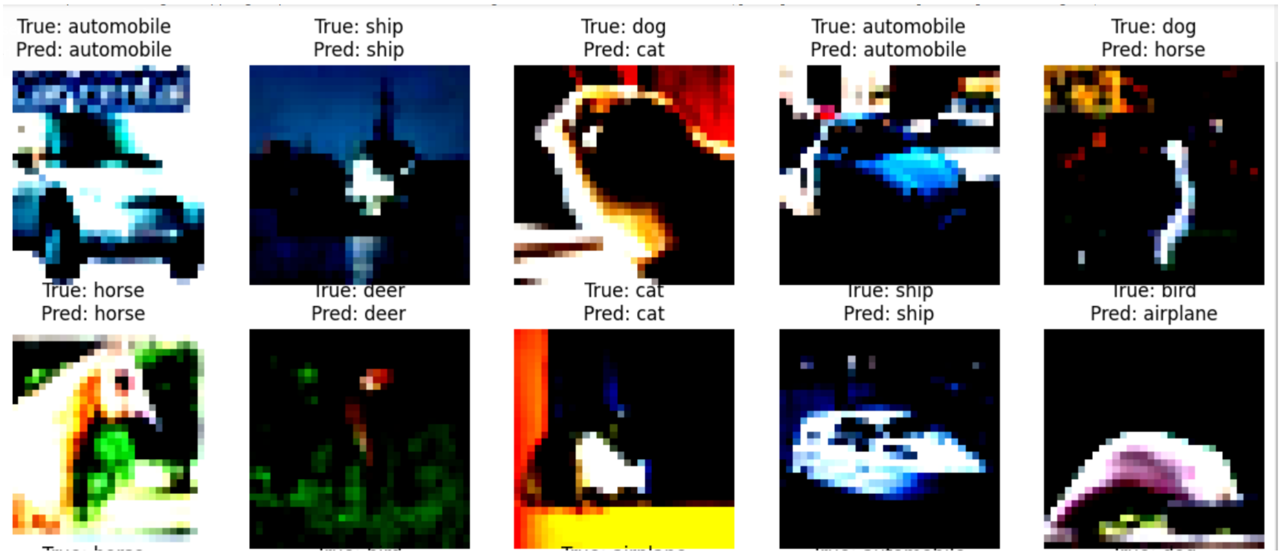
```
# CIFAR-10 class names
class_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog',
'frog', 'horse', 'ship', 'truck']

# Function to plot images along with their predicted and actual labels
def plot_images(images, true_labels, pred_labels, class_names,
num_images=20):
    plt.figure(figsize=(15, 15))
    for i in range(num_images):
        plt.subplot(5, 5, i + 1)
        plt.imshow(images[i])
        true_label = class_names[true_labels[i]]
        pred_label = class_names[pred_labels[i]]
        plt.title(f"True: {true_label}\nPred: {pred_label}")
        plt.axis('off')
    plt.show()
# Validate images from the validation set
val_predictions = model.predict(valid_X)
val_pred_labels = np.argmax(val_predictions, axis=1)

# Since valid_y is in shape (10000, 1), we need to flatten it to (10000,)
for comparison
valid_y_flat = valid_y.flatten()

# Plot the first 20 images from the validation dataset
plot_images(valid_X[:20], valid_y_flat[:20], val_pred_labels[:20],
class_names)
```

Out of the 20 validation images that we attempted to classify, the model accurately predicted the labels for 15 images. This means that the model successfully assigned the correct class labels to the majority of the images in this subset.



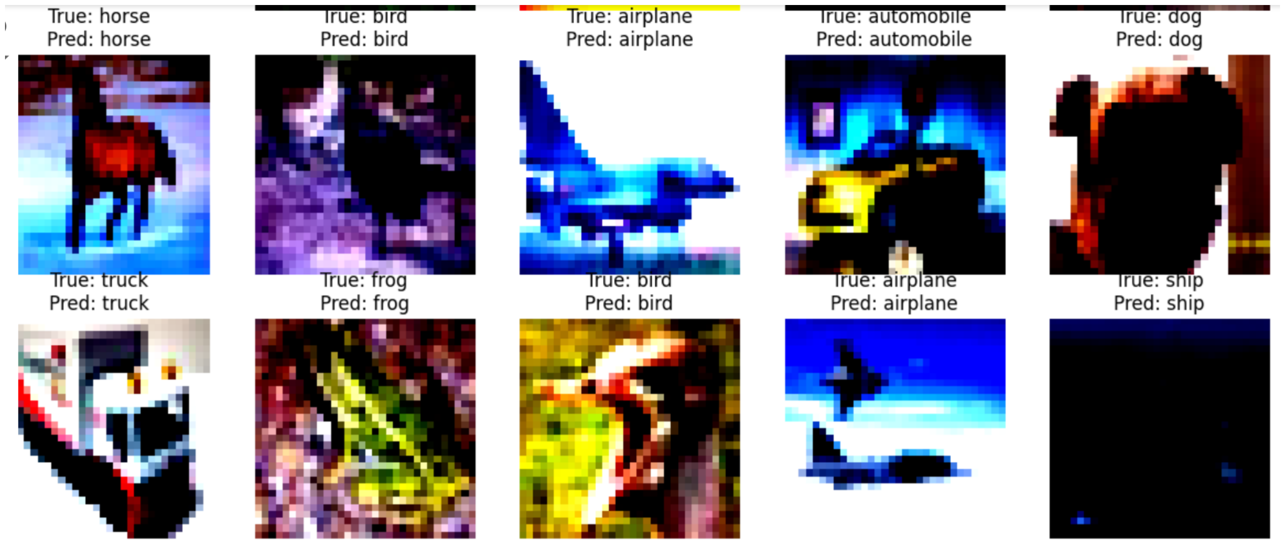


Fig 3: Classifies images using proposed model

Classification Report

The classification report presents a detailed assessment of the model's performance on the validation dataset, providing insights into its ability to classify images across different classes. Precision, recall, and F1-score metrics are reported for each class, indicating the model's accuracy, completeness, and balance in classification. While some classes show high precision and recall, such as automobile and ship, others show lower performance metrics, like cat and dog. Overall, the model achieves an accuracy of 72%, with macro and weighted averages of precision, recall, and F1-score indicating consistent performance across classes, with some variations.

```
from sklearn.metrics import classification_report
report = classification_report(valid_y_flat, val_pred_labels,
target_names=class_names)
print(report)
```

```
              precision    recall  f1-score   support

    airplane       0.84      0.78      0.81      1014
  automobile       0.91      0.88      0.89      1014
        bird       0.63      0.61      0.62       952
         cat       0.44      0.41      0.43      1016
        deer       0.84      0.58      0.68       997
         dog       0.69      0.45      0.55      1025
        frog       0.59      0.94      0.72       980
       horse       0.80      0.82      0.81       977
        ship       0.92      0.86      0.89      1003
       truck       0.72      0.92      0.81      1022

    accuracy                           0.72     10000
   macro avg       0.74      0.73      0.72     10000
weighted avg       0.74      0.72      0.72     10000
```

Conclusion

In conclusion, although the first CNN model seems to work well with 68.76% accuracy, the CNN did better after data augmentation. It improved by ~3% (to 71.48%) on the test data. The learning curves show that fitting wasn't an issue but that there were some noisy data points in the whole data set. Perhaps, a more sophisticated CNN model with more epochs and some more complex data augmentation techniques would improve the accuracy to the +85%.