Cifar10 Neural Network Classifier

Nikolaos Karapatsias 10661 (Department of Electrical and Computing Engineering AUTH)

This document provides a simple neural network model solving the classification problem of Cifar10 database using Tensorflow. The Cifar-10 database is a widely used benchmark dataset in machine learning and computer vision, designed for object recognition tasks. It consists of 60,000 32x32 color images across 10 distinct classes: airplanes (0), automobiles (1), birds (2), cats (3), deer (4), dogs (5), frogs (6), horses (7), ships (8), and trucks (9). The dataset is divided into 50,000 training images and 10,000 test images, making it a standard choice for evaluating classification algorithms. Each class is represented equally, with 6,000 images per category, offering a balanced dataset for training and testing.

I. INTRODUCTION

A. The main model

The model is a Convolutional neural network (knwon also as CNN) which consists of 3 convolutional layers and 2 Pooling layers. An overview of the model is shown below:

Layer (type)	Output Shape	Param #
conv2d_12 (Conv2D)	(None, 32, 32, 32)	416
conv2d_13 (Conv2D)	(None, 32, 32, 128)	65,664
dropout_4 (Dropout)	(None, 32, 32, 128)	0
max_pooling2d_8 (MaxPooling2D)	(None, 10, 10, 128)	0
conv2d_14 (Conv2D)	(None, 10, 10, 256)	295,168
dropout_5 (Dropout)	(None, 10, 10, 256)	0
max_pooling2d_9 (MaxPooling2D)	(None, 5, 5, 256)	0
flatten_4 (Flatten)	(None, 6400)	0
dense_4 (Dense)	(None, 10)	64,010

- First Convolutional layer:
  - filters: 32
  - kernels size: 3x3
  - activation function: leakyRelu
- Second Convolutional layer:
  - filters: 128
  - kernels size: 4x4
  - activation function: leakyRelu
- First Pooling layer:
  - kernels size: 3x3
- Third Convolutional layer:
  - filters: 256
  - kernels size: 3x3
  - activation function: leakyRelu
- Second Pooling layer:
  - kernels size: 2x2
- Output Dense layer:
  - -Perceptrons: 10
  - -activation function: Softmax

B. Features

1) Dropout layers: Added two Dropout layers, before every Pooling layer, with 20% and 10% of dropping rate respectively. That means that our model zero the 20% or 10% of the neurons of the previous layer. In that way we are preventing our model to overfit

2) Dynamic learning rate: Added a dynamic learning rate, which is reduced after 7th epoch to also reduce the possibility of our model to overfit.

3) Data augmentation: Used data augmentation where we either rotate, swift or flip randomly some of our training dataset images.

4) Leaky Relu: After some experiments with other activation functions we concluded that Relu is the ideal one. The goal of using Leaky Relu instead of normal Relu is to prevent some some neurons to get a zero value by giving them a really small negative number. In that way they can continue learning and not remain zero.

II. THE CODE

A. Loading the dataset

Firstly, we need to load the Cifar10 which is already included in tensorflow framework as shown below:

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data() #Loading Cifar10 dataset from tensorflow libraries
```

B. Transforming

From there, we are transforming our data into a desired form. More specifically, we know that each pixel gets a value between 0 and 250 so we are dividing both x arrays (train and test) in order to prevent having really high weights in our model. Now that both arrays contain elements with values between 0 and 1, we are transforming the y_train arrays in one hot decoding. One-hot decoding refers to converting categorical labels into a one-hot encoded format, where each label is represented as a binary vector of size 10 (for 10 classes), with a 1 in the position corresponding to the class and 0s elsewhere

```
x_train = x_train / 255.0  # Normalize x_train elements to get values from 0 to 1
y_train = np.eye(10)[y_train.squeeze()] # Convert y_train to one hot coding

x_test = x_test / 255.0  # Normalize x_test elements to get values from 0 to 1
y_test = np.eye(10)[y_test.squeeze()] # Convert y_test to one hot coding
```

C. Setting the data augmentation

In that step, we are creating an instance of ImageDataGenerator to perform real-time data augmentation on images by applying transformations like random rotations (up to 15 degrees), width and height shifts (up to 10% of the image size), and horizontal flipping, enhancing the dataset's diversity. The datagen.fit(x_train) step computes the statistics (like mean or variance) needed for standardization based on the training data.

```
# setting up our data augmentation parameters
datagen = ImageDataGenerator(
    rotation_range=15,
    width_shift_range=0.1,
    height_shift_range=0.1,
    horizontal_flip=True
)
datagen.fit(x_train)
```

D. Setting the layers and the model

Defining both layers and model as described before:

```
# Defining our model layers
Conv_1 = Conv2D(filters=32, kernel_size=(2, 2), activation=LeakyReLU(alpha=0.01),trainable=True, use_bias=True, padding="same")
Conv_2 = Conv2D(filters=128, kernel_size=(4, 4), activation=LeakyReLU(alpha=0.01),trainable=True, use_bias=True, padding="same")
Conv_3 = Conv2D(filters=256, kernel_size=(3, 3), activation=LeakyReLU(alpha=0.01),trainable=True, use_bias=True, padding="same")
Pool_a = MaxPooling2D(3, 3)
Pool_b = MaxPooling2D(2,2)
flat = Flatten()
Output_class = Dense(10, activation='softmax')

# Setting up our CNN model
CNN_model = Sequential([
    Input(shape=(32, 32, 3)),
    Conv_1,
    Conv_2,
    Dropout(0.2),
    Pool_a,
    Conv_3,
    Dropout(0.1),
    Pool_b,
    flat,
    Output_class
])
```

E. Choosing optimizer and loss function

After some testings we concluded that the best optimizer is Adam. Also we chose categorial crossentropy to calculate loss ,as we are using softmax in our output layer, with the formula: $L=-\sum Cy_ilog(pi)$ where i gets values from 1 till C=10 (our classes number)

```
optimizer = Adam(learning_rate=0.001) # Setting up our optimizer and the initial learning rate

CNN_model.compile(optimizer=optimizer, loss='categorical_crossentropy', metrics=['categorical_accuracy']) # Compile the model
```

F. Choosing learning rate

As we are planning to run our model for 20 epochs, we need to modify the learning rate to prevent training instabilities. So after some testings, we decided to decrease the learning rate by 5% in each step after the 7th epoch.

G. Training and testing the model

After all this process, our model is ready to be trained and make predictions

```
# Running the training and the testing process
history = CNN_model.fit(datagen.flow(x_train, y_train, batch_size=128),
                        validation_data=(x_test, y_test),
                        epochs=20,
                        callbacks=[lr_scheduler])

test_loss, test_acc = CNN_model.evaluate(x_test, y_test, verbose=2)

print(f"Test accuracy: {test_acc}") # Printing the final test accuracy of our model
```

III. EXECUTION TIME

The model takes around 40 minutes to execute. Training is mostly done in that time with am average of 120 seconds for each epoch to run, while testing part takes only 10 seconds.

IV. ACCURACY AND LOSS

Our model reaches an accuracy in test set of around 81,12% and an accuracy of 84,24% with 0,4567 loss for the training set as show in the following picture. We can also see that the loss is decreased at an acceptable step.The fact that both training and test set losses are decreasing means that we achieved a good balance between loss and learning rate. Also that indicates that we are far from both underfitting and overfitting.

```
Epoch 1/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 122s 305ms/step - categorical_accuracy: 0.3600 - loss: 1.7505 - val_categorical_accuracy: 0.5949 - val_loss: 1.1572 - learning_rate: 0.0010
Epoch 2/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 116s 295ms/step - categorical_accuracy: 0.6059 - loss: 1.1227 - val_categorical_accuracy: 0.6683 - val_loss: 0.9458 - learning_rate: 0.0010
Epoch 3/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 118s 302ms/step - categorical_accuracy: 0.6751 - loss: 0.9422 - val_categorical_accuracy: 0.7249 - val_loss: 0.8190 - learning_rate: 0.0010
Epoch 4/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 120s 307ms/step - categorical_accuracy: 0.7022 - loss: 0.8534 - val_categorical_accuracy: 0.7530 - val_loss: 0.7437 - learning_rate: 0.0010
Epoch 5/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 120s 308ms/step - categorical_accuracy: 0.7255 - loss: 0.7871 - val_categorical_accuracy: 0.7430 - val_loss: 0.7473 - learning_rate: 0.0010
Epoch 6/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 128s 328ms/step - categorical_accuracy: 0.7459 - loss: 0.7376 - val_categorical_accuracy: 0.7690 - val_loss: 0.6846 - learning_rate: 0.0010
Epoch 7/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 141s 360ms/step - categorical_accuracy: 0.7601 - loss: 0.6959 - val_categorical_accuracy: 0.7589 - val_loss: 0.6968 - learning_rate: 0.0010
Epoch 8/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 138s 353ms/step - categorical_accuracy: 0.7679 - loss: 0.6728 - val_categorical_accuracy: 0.7754 - val_loss: 0.6606 - learning_rate: 9.5000e-04
Epoch 9/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 137s 349ms/step - categorical_accuracy: 0.7717 - loss: 0.6503 - val_categorical_accuracy: 0.7938 - val_loss: 0.6259 - learning_rate: 9.0250e-04
Epoch 10/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 135s 345ms/step - categorical_accuracy: 0.7890 - loss: 0.6084 - val_categorical_accuracy: 0.7723 - val_loss: 0.6617 - learning_rate: 8.5737e-04
Epoch 11/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 140s 359ms/step - categorical_accuracy: 0.7944 - loss: 0.5990 - val_categorical_accuracy: 0.7805 - val_loss: 0.6570 - learning_rate: 8.1451e-04
Epoch 12/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 138s 354ms/step - categorical_accuracy: 0.8027 - loss: 0.5745 - val_categorical_accuracy: 0.7860 - val_loss: 0.6198 - learning_rate: 7.7378e-04
Epoch 13/20
...
Epoch 20/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 135s 344ms/step - categorical_accuracy: 0.8424 - loss: 0.4567 - val_categorical_accuracy: 0.8124 - val_loss: 0.5662 - learning_rate: 5.1334e-04
313/313 - 7s - 22ms/step - categorical_accuracy: 0.8124 - loss: 0.5662
Test accuracy: 0.8123999834060669
```

V. ACCURACY WITH DIFFERENT PARAMETERS

A. Choosing different featured maps number and pooling layers

Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 31, 31, 32)	384
conv2d_7 (Conv2D)	(None, 28, 28, 64)	32,768
dropout_4 (Dropout)	(None, 28, 28, 64)	0
max_pooling2d_4 (MaxPooling2D)	(None, 7, 7, 64)	0
conv2d_8 (Conv2D)	(None, 5, 5, 128)	73,728
dropout_5 (Dropout)	(None, 5, 5, 128)	0
max_pooling2d_5 (MaxPooling2D)	(None, 1, 1, 128)	0
flatten_2 (Flatten)	(None, 128)	0
dense_2 (Dense)	(None, 10)	1,290

We've replaced our convolutional layers with smaller amount of featured maps (32 64 and 128 respectively) while also increased the kernel dimensions of our pooling layers. The result is that we got a faster execution of around 16 minutes but with a lower accuracy in the test set of around 70.8% as shown below:

```
Epoch 1/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 50s 124ms/step - categorical_accuracy: 0.2752 - loss: 1.9539 - val_categorical_accuracy: 0.4797 - val_loss: 1.5202 - learning_rate: 0.0010
Epoch 2/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 46s 118ms/step - categorical_accuracy: 0.4676 - loss: 1.4942 - val_categorical_accuracy: 0.5508 - val_loss: 1.3403 - learning_rate: 0.0010
Epoch 3/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 45s 115ms/step - categorical_accuracy: 0.5306 - loss: 1.3375 - val_categorical_accuracy: 0.5733 - val_loss: 1.2623 - learning_rate: 0.0010
Epoch 4/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 47s 120ms/step - categorical_accuracy: 0.5652 - loss: 1.2447 - val_categorical_accuracy: 0.5984 - val_loss: 1.1786 - learning_rate: 0.0010
Epoch 5/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 47s 121ms/step - categorical_accuracy: 0.5936 - loss: 1.1720 - val_categorical_accuracy: 0.6426 - val_loss: 1.0942 - learning_rate: 0.0010
Epoch 6/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 47s 120ms/step - categorical_accuracy: 0.6138 - loss: 1.1249 - val_categorical_accuracy: 0.6385 - val_loss: 1.0603 - learning_rate: 0.0010
Epoch 7/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 49s 126ms/step - categorical_accuracy: 0.6196 - loss: 1.0909 - val_categorical_accuracy: 0.6563 - val_loss: 1.0197 - learning_rate: 0.0010
Epoch 8/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 47s 121ms/step - categorical_accuracy: 0.6419 - loss: 1.0386 - val_categorical_accuracy: 0.6307 - val_loss: 1.0816 - learning_rate: 9.5000e-04
Epoch 9/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 51s 131ms/step - categorical_accuracy: 0.6542 - loss: 1.0076 - val_categorical_accuracy: 0.6652 - val_loss: 0.9924 - learning_rate: 9.0250e-04
Epoch 10/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 49s 125ms/step - categorical_accuracy: 0.6680 - loss: 0.9700 - val_categorical_accuracy: 0.6306 - val_loss: 1.0810 - learning_rate: 8.5737e-04
Epoch 11/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 50s 128ms/step - categorical_accuracy: 0.6675 - loss: 0.9689 - val_categorical_accuracy: 0.6694 - val_loss: 0.9723 - learning_rate: 8.1451e-04
Epoch 12/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 47s 121ms/step - categorical_accuracy: 0.6720 - loss: 0.9513 - val_categorical_accuracy: 0.6711 - val_loss: 0.9613 - learning_rate: 7.7378e-04
Epoch 13/20
...
Epoch 20/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 47s 120ms/step - categorical_accuracy: 0.7050 - loss: 0.8592 - val_categorical_accuracy: 0.7080 - val_loss: 0.8692 - learning_rate: 5.1334e-04
313/313 - 2s - 7ms/step - categorical_accuracy: 0.7080 - loss: 0.8692
Test accuracy: 0.7080000042915344
```

B. Choosing different kernel size for each layer:

Layer (type)	Output Shape	Param #
conv2d_15 (Conv2D)	(None, 29, 29, 32)	1,536
conv2d_16 (Conv2D)	(None, 27, 27, 128)	36,864
dropout_10 (Dropout)	(None, 27, 27, 128)	0
max_pooling2d_10 (MaxPooling2D)	(None, 9, 9, 128)	0
conv2d_17 (Conv2D)	(None, 7, 7, 256)	294,912
dropout_11 (Dropout)	(None, 7, 7, 256)	0
max_pooling2d_11 (MaxPooling2D)	(None, 3, 3, 256)	0
flatten_5 (Flatten)	(None, 2304)	0
dense_5 (Dense)	(None, 10)	23,050

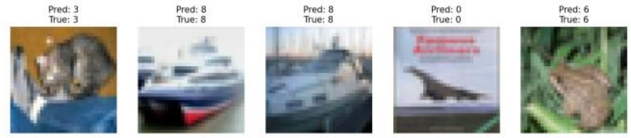
In this test example we've modified the size of the kernels to 4x4, 3x3, 3x3 in succession. We've achieved an accuracy of 76.2 % in the test set with an execution time around 22 minutes which is half of main models time!

```
Epoch 1/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 69s 174ms/step - categorical_accuracy: 0.3205 - loss: 1.8550 - val_categorical_accuracy: 0.5238 - val_loss: 1.3999 - learning_rate: 0.0010
Epoch 2/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 71s 183ms/step - categorical_accuracy: 0.5106 - loss: 1.3742 - val_categorical_accuracy: 0.6139 - val_loss: 1.1536 - learning_rate: 0.0010
Epoch 3/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 64s 163ms/step - categorical_accuracy: 0.5844 - loss: 1.2003 - val_categorical_accuracy: 0.6304 - val_loss: 1.0998 - learning_rate: 0.0010
Epoch 4/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 64s 165ms/step - categorical_accuracy: 0.6257 - loss: 1.0882 - val_categorical_accuracy: 0.6573 - val_loss: 1.0036 - learning_rate: 0.0010
Epoch 5/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 69s 177ms/step - categorical_accuracy: 0.6455 - loss: 1.0264 - val_categorical_accuracy: 0.6809 - val_loss: 0.9275 - learning_rate: 0.0010
Epoch 6/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 66s 169ms/step - categorical_accuracy: 0.6607 - loss: 0.9806 - val_categorical_accuracy: 0.6824 - val_loss: 0.9362 - learning_rate: 0.0010
Epoch 7/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 68s 174ms/step - categorical_accuracy: 0.6788 - loss: 0.9349 - val_categorical_accuracy: 0.7153 - val_loss: 0.8530 - learning_rate: 0.0010
Epoch 8/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 79s 201ms/step - categorical_accuracy: 0.6931 - loss: 0.8931 - val_categorical_accuracy: 0.7119 - val_loss: 0.8482 - learning_rate: 9.5000e-04
Epoch 9/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 69s 176ms/step - categorical_accuracy: 0.7049 - loss: 0.8634 - val_categorical_accuracy: 0.7184 - val_loss: 0.8223 - learning_rate: 9.0250e-04
Epoch 10/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 66s 168ms/step - categorical_accuracy: 0.7155 - loss: 0.8268 - val_categorical_accuracy: 0.7380 - val_loss: 0.7676 - learning_rate: 8.5737e-04
Epoch 11/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 67s 172ms/step - categorical_accuracy: 0.7229 - loss: 0.8048 - val_categorical_accuracy: 0.7363 - val_loss: 0.7748 - learning_rate: 8.1451e-04
Epoch 12/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 66s 169ms/step - categorical_accuracy: 0.7351 - loss: 0.7820 - val_categorical_accuracy: 0.7511 - val_loss: 0.7419 - learning_rate: 7.7378e-04
Epoch 13/20
...
Epoch 20/20
391/391 ━━━━━━━━━━━━━━━━━━━━ 75s 191ms/step - categorical_accuracy: 0.7696 - loss: 0.6803 - val_categorical_accuracy: 0.7622 - val_loss: 0.7097 - learning_rate: 5.1334e-04
313/313 - 4s - 13ms/step - categorical_accuracy: 0.7622 - loss: 0.7097
Test accuracy: 0.7621999979019165
```

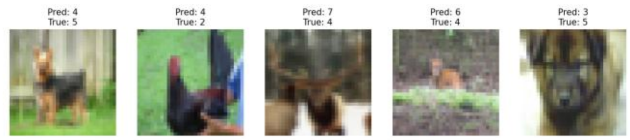
C. Conclusion

By modifying our models parameters such as kernels size, or number of produced featured maps or pooling layers mapping we can achieve various accuracies with different training times. For example in the second experiment above we achieved to train our model in half the time of our main one in exchange of loosing nearly 8% of accuracy on training set

VI. .EXAMPLES OF RIGHT AND WRONG CLASSIFICATION



In the above image we can see that our model predicts in the right way the 5 images. For example the first image is a cat and predicts correctly a cat represented with the number 3 as the fourth class of the database



The above image is a sample of a false classification where for example our model predicts a deer (4) while the correct answer is dog (5)

VII. VS KNN & NCC

As we proved on our previous project, 1NN and 3NNgot 35.39% and 33.03% of accuracy respectively and NCC got 27.74% all of them in an execution time way less than a second for the same dataset (Cifar10). On the other side, our model achieved an accuracy of 81,12% with a way worst execution time around 40 minutes