CIFAR-10 Image Classification

Enrique Gómez Cruz

March 7, 2025

Abstract

This work investigates the performance of VGG16 and AlexNet architectures for image classification using the CIFAR-10 dataset. Models were trained locally on a MacBook Air M3, comparing scratch implementations and transfer learning using an ImageNet pre-trained VGG16 model. Results demonstrate a significant performance advantage for the scratch-trained VGG16, achieving 90% validation accuracy, compared to 72% for scratch-trained AlexNet. The transfer learning approach with the pre-trained VGG16 model, however, yielded a lower accuracy of 70%. Both neural networks were trained using identical settings, including batch size and learning rate. The analysis highlights the effectiveness of VGG16 for this task and underscores the challenges of applying pre-trained models without careful adaptation.

1 CIFAR-10

The CIFAR-10 dataset consists of 60,000 color images, each with dimensions of 32×32 pixels, divided into 10 distinct classes. Each class contains 6,000 images.

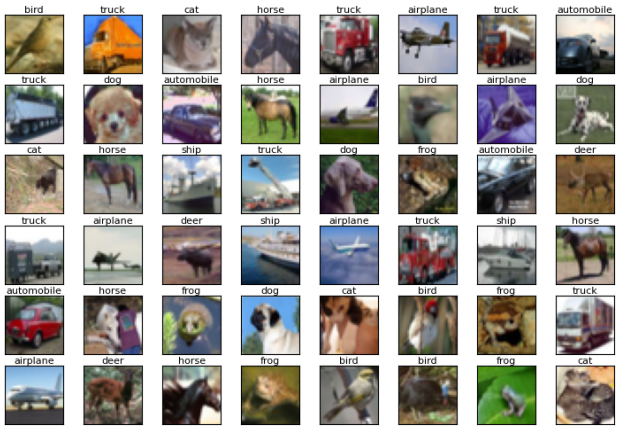

Figure 1: First 48 images of the training data set.

The dataset can be directly loaded using the Keras library. The `load_data()` function returns the training and test sets.

```
from tensorflow.keras.datasets import cifar10
(X_train, y_train), (X_test, y_test) =
↪  cifar10.load_data()
```

To optimize hyperparameters and provide an unbiased evaluation of the model during training, a validation set is created by partitioning 10% of the training data. A fixed random seed (`random_state=0`) ensures reproducibility.

```
from sklearn.model_selection import
↪  train_test_split

X_train, X_valid, y_train, y_valid =
↪  train_test_split(X_train, y_train,
↪  test_size=0.1, random_state=0)

print('Training:   ', X_train.shape)
print('Validation: ', X_valid.shape)
print('Testing:    ', X_test.shape)
```

2 Data Preprocessing

2.1 Normalization

Neural networks work best if the data they are trained on have features with comparable scales. Normalization improves the model's training efficiency and effectiveness. The pixel values of the images are between 0 and 255 for each RGB channel, their values are normalized by rescaling (min-max normalization) or standardization (substracting the mean and then dividing by the standard deviation). Both methods were separately tried.

```
import numpy as np
# Convert pixel values to float32
X_train = X_train.astype('float32')
X_test  = X_test.astype('float32')
X_valid = X_valid.astype('float32')

# Standardize the data
mean = np.mean(X_train)
std  = np.std(X_train) + 1e-7 # (small value avoids
↪  division by zero)
X_train = (X_train - mean) / std
```

```
X_test  = (X_test - mean) / std
X_valid = (X_valid - mean) / std
```

2.2 Label encoding

Each of the 10 classes need to be encoded so that they become computable by the model. Since the total number of classes is fairly small, one-hot encoding is a valid choice, it also does not imply an intrinsic order that would otherwise be unnecessarily learned if the classes where, naively, given a random integer value.

```
from keras.utils import to_categorical

# Encode each class into a one-hot vector.
y_train = to_categorical(y_train, 10)
y_valid = to_categorical(y_valid, 10)
y_test  = to_categorical(y_test, 10)
```

2.3 Data augmentation

To further help the model improve its ability to generalize, by reducing overfitting, the data is artificially expanded by creating modified versions of the images: rotated, translated, flipped, zoomed, sheared, color-shifted and brightness-shifted. Special care must be taken not to over emphasize the transformations since the images are already small (32×32) and may lose its characteristic feature. For example, by rotating an image too much the main object might go out of frame.

```
from tensorflow.keras.preprocessing.image import
↪  ImageDataGenerator

# Data augmentation pipeline
data_generator = ImageDataGenerator(
# Rotate images randomly by up to 15 degrees
rotation_range=15,

# Shift images horizontally by up to 12% of their
↪  width
width_shift_range=0.12,

# Shift images vertically by up to 12% of their
↪  height
height_shift_range=0.12,

# Randomly flip images horizontally
horizontal_flip=True,

# Zoom images in by up to 10%
zoom_range=0.1,

# Change brightness by up to 10%
brightness_range=[0.9,1.1],

# Shear intensity (shear angle in counter-clockwise
↪  direction in degrees)
shear_range=10,

# Channel shift intensity
channel_shift_range=0.1
)
```

The code above defines a pipeline that applies random transformations to the images at each epoch as they are loaded into the model during training.

This architectures share the common patter

3 Model Architecture

Two similar architectures were trained from scratch with the CIFAR-10 dataset: *a minimalistic version* of VGG16 and AlexNet. Both architectures share the common pattern of using a convolutional layer followed by batch normalization layer, a max pooling layer and a dropout layer. The combination of the three latter help in sum prevent overfitting while reducing computational complexity.

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 32, 32, 32)	896
batch_normalization (BatchNormalization)	(None, 32, 32, 32)	128
conv2d_1 (Conv2D)	(None, 32, 32, 32)	9,248
batch_normalization_1 (BatchNormalization)	(None, 32, 32, 32)	128
max_pooling2d (MaxPooling2D)	(None, 16, 16, 32)	0
dropout (Dropout)	(None, 16, 16, 32)	0
conv2d_2 (Conv2D)	(None, 16, 16, 64)	18,496
batch_normalization_2 (BatchNormalization)	(None, 16, 16, 64)	256
conv2d_3 (Conv2D)	(None, 16, 16, 64)	36,928
batch_normalization_3 (BatchNormalization)	(None, 16, 16, 64)	256
max_pooling2d_1 (MaxPooling2D)	(None, 8, 8, 64)	0
dropout_1 (Dropout)	(None, 8, 8, 64)	0
conv2d_4 (Conv2D)	(None, 8, 8, 128)	73,856
batch_normalization_4 (BatchNormalization)	(None, 8, 8, 128)	512
conv2d_5 (Conv2D)	(None, 8, 8, 128)	147,584
batch_normalization_5 (BatchNormalization)	(None, 8, 8, 128)	512
max_pooling2d_2 (MaxPooling2D)	(None, 4, 4, 128)	0
dropout_2 (Dropout)	(None, 4, 4, 128)	0
conv2d_6 (Conv2D)	(None, 4, 4, 256)	295,168
batch_normalization_6 (BatchNormalization)	(None, 4, 4, 256)	1,024
conv2d_7 (Conv2D)	(None, 4, 4, 256)	590,080
batch_normalization_7 (BatchNormalization)	(None, 4, 4, 256)	1,024
max_pooling2d_3 (MaxPooling2D)	(None, 2, 2, 256)	0
dropout_3 (Dropout)	(None, 2, 2, 256)	0
flatten (Flatten)	(None, 1024)	0
dense (Dense)	(None, 10)	10,250

Figure 2: VGG16 architecture with 1,186,346 total parameters (4.53 MB).

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 6, 6, 128)	46,592
batch_normalization (BatchNormalization)	(None, 6, 6, 128)	512
max_pooling2d (MaxPooling2D)	(None, 3, 3, 128)	0
conv2d_1 (Conv2D)	(None, 3, 3, 256)	819,456
batch_normalization_1 (BatchNormalization)	(None, 3, 3, 256)	1,024
max_pooling2d_1 (MaxPooling2D)	(None, 1, 1, 256)	0
conv2d_2 (Conv2D)	(None, 1, 1, 256)	590,080
batch_normalization_2 (BatchNormalization)	(None, 1, 1, 256)	1,024
conv2d_3 (Conv2D)	(None, 1, 1, 256)	65,792
batch_normalization_3 (BatchNormalization)	(None, 1, 1, 256)	1,024
conv2d_4 (Conv2D)	(None, 1, 1, 256)	65,792
batch_normalization_4 (BatchNormalization)	(None, 1, 1, 256)	1,024
max_pooling2d_2 (MaxPooling2D)	(None, 1, 1, 256)	0
flatten (Flatten)	(None, 256)	0
dense (Dense)	(None, 1024)	263,168
dropout (Dropout)	(None, 1024)	0
dense_1 (Dense)	(None, 1024)	1,049,600
dropout_1 (Dropout)	(None, 1024)	0
dense_2 (Dense)	(None, 10)	10,250

Figure 3: AlexNet architecture with 2,915,338 total parameters (11.12 MB).

Apart from the scratch trained neural networks, an ImageNet pre-trained VGG16 network was used as well. This pre-trained network was used as a base model for transfer learn on the CIFAR-10 dataset. The base model's weight were frozen and a additional trainable layers were added.

Layer (type)	Output Shape	Param #
resizing (Resizing)	(None, 224, 224, 3)	0
vgg16 (Functional)	(None, 7, 7, 512)	14,714,688
flatten (Flatten)	(None, 25088)	0
dense (Dense)	(None, 512)	12,845,568
dropout (Dropout)	(None, 512)	0
dense_1 (Dense)	(None, 10)	5,130

Figure 4: Pretrained VGG16 architecture with 27,565,386 total parameters (105.15 MB).

All neural networks were trained with a batch size of 64 for a maximum of 300 epochs, a learning rate of 5×10^{-4} for the Adam optimizer and the categorical crossentropy as the loss function. In case the training plateaus for 10 consecutive epochs, the learning rate is dynamically halved, also if that behaviour continuous for 40 epochs the model is stopped earlier.

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import
↪  ReduceLROnPlateau, EarlyStopping

# Compile model
model.compile(optimizer=Adam(learning_rate=0.0005),
↪  loss='categorical_crossentropy',
↪  metrics=['accuracy'])

# Callbacks during training
reduce_lr = ReduceLROnPlateau(monitor='val_loss',
↪  factor=0.5, patience=10, min_lr=0.00001)
early_stopping = EarlyStopping(monitor='val_loss',
↪  patience=40, restore_best_weights=True,
↪  verbose=1)

# Train the model
model.fit(data_generator.flow(X_train, y_train,
↪  batch_size=64), epochs=300,
↪  validation_data=data_generator.flow(X_valid,
↪  y_valid, batch_size=64), callbacks=[reduce_lr,
↪  early_stopping], verbose=2)
```

4 Results

The models were trained locally using a MacBook Air M3 with 16GB of RAM. Both Pytorch and Tensorflow support GPU via the Metal library. CPU vs GPU usage was slightly tested and found out, unsurprisingly, that GPU was 3-4 times faster than the CPU.

The scratch VGG16 network performed significantly better than the scratch Alexnet network, and was smaller, being a third of AlexNet's size. The pretrained network was the slowest to train and was the biggest of them all.

A critical step to have the pretrained network usable was to resize the initial input images since the model was trained on ImageNet that has images of size 224×224 on average. It was observed that if the images were not resized, then the training would fail to increase beyond 10%. The training performance was also sensible to the added trainable layers. The "right" combination of layers are the ones that are being reported. An example of trainable layers that didn't perform well are the following:

```
model = Sequential()

# Preprocessing layer
model.add(Input(shape=X_train.shape[1:]))
model.add(Resizing(224, 224))

# VGG16 pretrained model
model.add(base_model)

# Trainable layers THAT DIDN'T PERFORM WELL
model.add(GlobalAveragePooling2D())
model.add(Dropout(0.5))
model.add(Dense(10, activation=('softmax')))
```

Table 1: CNN training performance.

CNN	Δt (minutes)	Val. Acc. %
Scratch VGG16	249	90
Scratch AlexNet	108	72
Pretrained VGG16	150	70

The pretrained model was so slow that it was hard to completely trained it. The reported values are preliminary in that sense that they were the last values on the epoch were the model was interrupted (it was taking too long!). The model ran 10 epochs for about 150 minutes, reaching an accuracy of no more than 70%.

Other impactful aspect was that of which layers to freeze of the pretrained base model. At the end all the pretrained layers were frozen, but several unsuccesfull tries (performance wise) consisted of only freezing the deeper layers.

The normalization of the data was particularly relevant for the training performance of all the models. It was the case that dividing only by the maximum value (255) was made the models perform poorly. Instead, the data was standardised, the good results are shown in the accuracy plots.

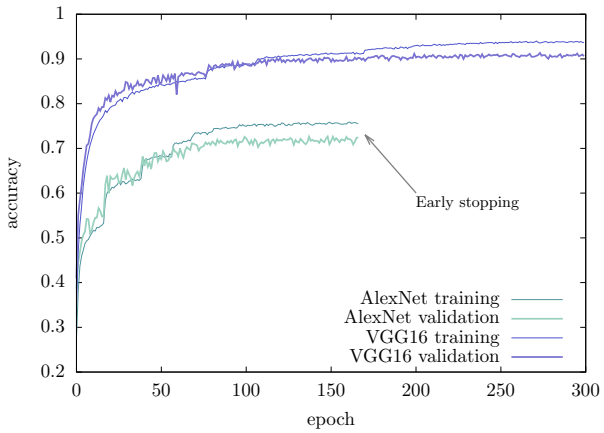

Figure 5: Training and validation accuracies of AlexNet and VGG16 scratch models.

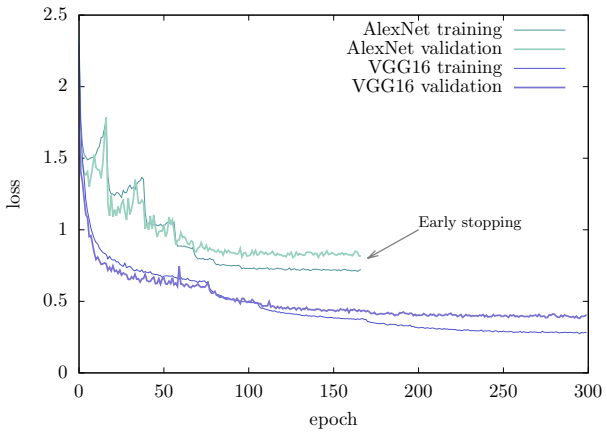

Figure 6: Training and validation losses of AlexNet and VGG16 scratch models.

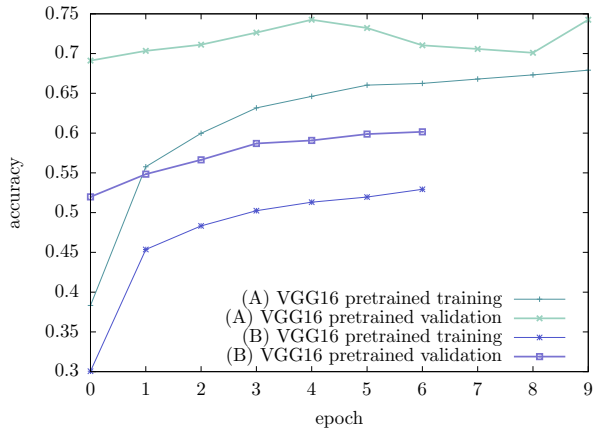

Figure 7: Training and validation accuracies of two models using a pretrained VGG16. Model (B) is a failed attempt that used a Global Average Pooling followed by a Dropout layer after the base model. Model (A) is an improved version that Flattens the output of the base model and then connects to a Dense layer followed by a Dropout.

The scratch models were very fast to evaluate and predict the test set, taking around 3 seconds. On the other hand, the pretrained model took around 100 seconds.

Table 2: CNN testing performance.

CNN	Δt (seconds)	Val. Acc. %
Scratch VGG16	4	90
Scratch AlexNet	3	74
Pretrained VGG16	100	77

4.1 Testing an out-of-set image

Three out-of-set images were tested with the scratch VGG16 model. The images had to be preprocessed the same way the training set was. The model failed to recognize all three images as dogs. According to the model the top dog is indeed a dog, the middle dog is a deer and the bottom dog is in reality a horse.

(a) A dog.

(b) A deer.

(c) ...and a horse.

Figure 8: Out-of-set pictures of dogs classified with the scratch VGG16 model.

5 Acknowlegedments

This models were so heavy to train, specially locally, that is why help from outside was very helpful, given that everybody could try different configurations on their own compute time. Helpful ideas came about this way, and a special thanks to Fausto Morales for the help.