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.

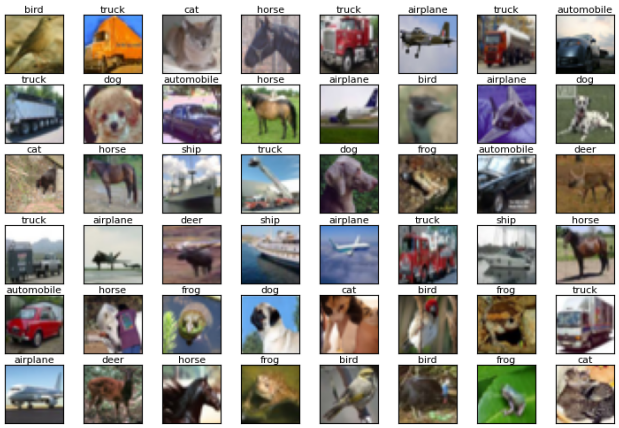

Figure 1: First 48 images of the training data set.

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

Normalization is crucial for training effective neural networks by ensuring features have comparable scales. This improves training efficiency and model effectiveness. Pixel values in the images, ranging from 0 to 255 for each RGB channel, are normalized through standardization (subtracting the mean and dividing by the standard deviation).

```
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

The ten classes are encoded using one-hot encoding to make them computable by the model. This approach is suitable for the small number of classes and avoids implying an unintended order.

```
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 reduce overfitting and enhance the model's generalization ability, data augmentation creates modified versions of the original images through rotations, translations, flips, zooms, shears, and color shifts. 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 data augmentation pipeline applies these transformations dynamically to the images as they are loaded during training.

3 Model Architecture

Two architectures, AlexNet and a minimalistic versions of VGG16, were trained from scratch using the CIFAR-10 dataset. These architectures use convolutional layers, batch normalization, max pooling, and dropout layers to prevent overfitting and reduce 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).

In addition to the scratch-trained networks, an ImageNet pre-trained VGG16 network was used for transfer learning. The pre-trained weights were frozen, and 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 versus GPU usage was slightly tested and was found out, unsurprisingly, that GPU performance was 3-4 times faster than CPU performance.

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 crucial step in making the pretrained network usable was resizing the input images, as the model was originally trained on ImageNet, where images typically have a size of 224×224. It was observed that without resizing, training accuracy failed to surpass 10%. Additionally, training performance was highly sensitive to the choice of trainable layers. The optimal combination of layers is the one reported in this study. Below is an example of trainable layers that did not yield good results:

```
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 extremely slow, making it difficult to complete full training runs. The reported values are preliminary, as they correspond to the last recorded epoch before the model was interrupted due to excessive runtime. Over the course of 10 epochs (approximately 150 minutes), the model achieved a maximum accuracy of around 70%.

Another critical factor was determining which layers of the pretrained base model to freeze. Ultimately, all pretrained layers were frozen, but several unsuccessful attempts involved freezing only the deeper layers, which led to suboptimal performance.

Data normalization played a particularly important role in training performance across all models. Simply dividing by the maximum pixel value (255) resulted in poor performance. Instead, standardizing the data yielded significantly better results, as reflected in the accuracy plots.

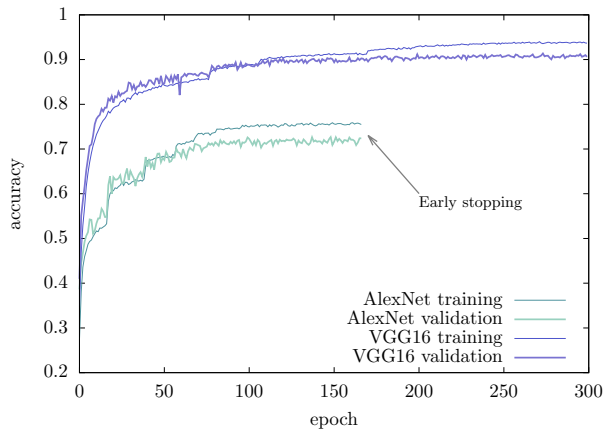

Figure 5: Training and validation accuracies of AlexNet and VGG16 scratch models.

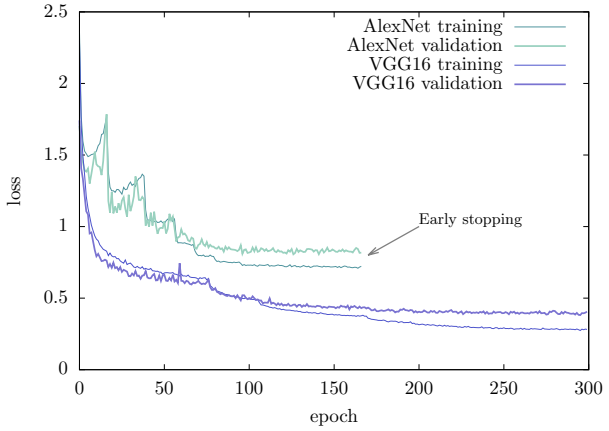

Figure 6: Training and validation losses of AlexNet and VGG16 scratch models.

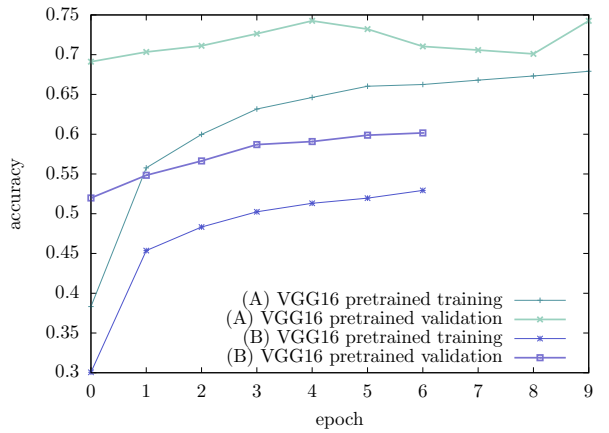

Figure 7: Training and validation accuracies of two models using a pretrained VGG16. Model (B) is a failed attempt that uses a Global Average Pooling followed by a Dropout layer after the base model. Model (A) is an improved version that Flattens the output of the base model and then connects to a Dense 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 using the VGG16 model trained from scratch. These images were preprocessed in the same way as the training set. However, the model failed to correctly classify all three as dogs. According to its predictions, only one image was identified as a dog, while the others were misclassified as a deer and 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

These models were computationally expensive to train, especially on local machines. External assistance proved invaluable, as it allowed multiple contributors to experiment with different configurations on their own computing resources. This collaborative approach led to valuable insights. A special thanks to Fausto Morales for his support.