

NOVEMBER 21, 2025

# PORTFOLIO OF EVIDENCE

PDAN8412

RICK JACKSON

**ST10458509**

Varsity College Cape Town

## Table of Contents

<b>INTRODUCTION.....</b>	<b>2</b>
<b>DATASET EVALUATION AND JUSTIFICATION.....</b>	<b>2</b>
<b>PLANNING.....</b>	<b>3</b>
<b>ANALYSIS AND RESULTS.....</b>	<b>5</b>
<b>CONCLUSION.....</b>	<b>13</b>
<b>REFERENCES.....</b>	<b>13</b>

## INTRODUCTION

A publishing house has tasked the creation of CNN Image Classification model for eventual development into facial recognition software. At this point, getting image recognition capabilities is a good start. This report explores the methods, results and interpretations of the dataset as well as creating an Image Classification model that is evaluated for precision and accuracy, among other metrics.

In Machine Learning, Image classification is a vital task where the goal is to assign a label to an image based on its contents. Convolutional Neural Networks (CNNs) are designed to analyse and interpret images. These CNNs are very good at detecting shapes, textures and patterns, by breaking down the image into smaller parts and learning from these details. It processes these patterns in multiple layers, where it is able to increasingly identify complex features. This is useful in the classifying images of objects, scenes or animals (GeekforGeeks, 2024).

## DATASET EVALUATION and JUSTIFICATION

The publishing house did specify that an image classifier model is needed but it does not need to identify human faces at this point.

The CIFAR-10 Image Dataset has been selected for this task.

The CIFAR-10 dataset is a widely used dataset for image classification. It contains 60,000 colour images (32×32) across 10 classes (e.g. airplane, cat, dog, automobile). This dataset was chosen for its suitability to the task, containing real-world images and is large enough to sufficiently train a CNN model.

All images are uniform in size (32x32 RGB images) and each class is **already balanced** and contains **5000 images each**. It has already been **pre-split into training and testing sets** which will allow focus on the effective data exploration methods and model training and evaluation required for this task.

The images contained in the dataset in the various classes are captured using a **variety of angles, lighting, backgrounds**. This is excellent for model data diversification, making the model more robust.

CIFAR dataset can be found at: <https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz>

The downloadable version of the CIFAR-10 dataset hosted by the institute who compiled, has been pickled. This method was approached with weariness due to the nature of how Pickling occurs in Python.

Pickling allows the user the ability to save and transmit for later use. It is Python module used for serializing and de-serializing a Python object structure. This means it takes the scattered data on the disk in chunks and translates this data into series that can be saved and converted back for later use. Pickling is a way to convert a Python object (list, dictionary, etc.) into a character stream. It provides a facility to convert any Python object to a byte stream. This Byte stream contains all essential information about the object so that it can be reconstructed, or "unpickled" and get back into its original form in any Python (GeekforGeeks, 2024).

Although future work emphasizes facial recognition, CIFAR-10 provides a sound foundation for building and evaluating the performance of CNN architectures on general image-recognition tasks.

## PLANNING

For all of this to work, the actual data stream format of pickle is really a very simple programming language, which is evaluated when you load a pickle stream to actually create the deserialized object. This makes it possible to inject custom code into a generated pickle object to make it run arbitrary Python code at the time of loading (GeeksforGeeks, 2024). There is no practical way to verify that the pickled code doesn't contain anything malicious, making its use risky.

The CIFAR Dataset has been downloaded directly from the PyTorch library to avoid undergoing 'de-pickling'.

The code for **unpickling** has been included for perusal but was not used for this notebook.

The dataset has been imported via PyTorch to not only make analysis less complex, it also avoids any potential security issues.

**STEP 1: Exploratory Data Analysis** - The dataset is loaded and inspected. This will allow for insights into the shape and structure of the data being dealt with.

The dataset is visualized, sample data is viewed, and dataset is checked for cleanliness (no missing values etc.)

Pixel density is checked to see if normalization is necessary. Class balances are checked.

**STEP 2: Feature Selection** – For CNNs, feature selection is not necessary as the nature of CNNs training allows it to detect patterns in the data on its own,

The images were normalized anyway.

**STEP 3: Train the Model –**

Simple CNN Architecture is created with convolutional, pooling, and fully connected layers.

**Setting up the Training Loop**

**Hyperparameters:**

Learning rate: 0.001, Batch size: 64, **Epochs: 30**, Optimizer: Adam, Loss: **Categorical Crossentropy**

**Plotting Losses on Curve**

**STEP 4: Evaluate the Model –**

The model is evaluated on the following metrics:

Accuracy, Confusion Matrix, Precision, Recall, and F-1 score.

The Loss and Accuracy Curve plot will also be used as an evaluation metric.

## ANALYSIS and RESULTS

### STEP 1: EXPLORATORY DATA ANALYSIS

The dataset is loaded into the Jupyter notebook via PyTorch, both the training set and the testing set.

#### Loading Training Data using PyTorch

```
import torchvision.transforms as transforms, torchvision, matplotlib.pyplot as plt

trainset = torchvision.datasets.CIFAR10(root='/Users/rickjackson/Downloads',
                                       train=True,
                                       download=True,
                                       transform=transform)

trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=4,
                                          shuffle=True)

images, labels = next(iter(trainloader))
```

Training data now loaded via PyTorch.

#### Loading Test Data using PyTorch ¶

```
# transform is defined using existing transform already defined above.

# e.g. transform = transforms.Compose([...])

# Loading the CIFAR-10 test set
testset = torchvision.datasets.CIFAR10(
    root='/Users/rickjackson/Downloads',
    train=False,      # <-- differentiator to get test set.
    download=True,
    transform=transform
)

# Create DataLoader for test set
testloader = torch.utils.data.DataLoader(
    testset,
    batch_size=4,     # same as trainloader
    shuffle=False     # no shuffling required for test data
)
```

Test data now loaded via PyTorch

The differentiator to get the test data was to change train=false.

The **dataset structure is inspected** for shape and structure:

```
[19]: # Show shape of one image
img, label = trainset[0]
print("Image shape:", img.shape)
print("Label:", label)
```

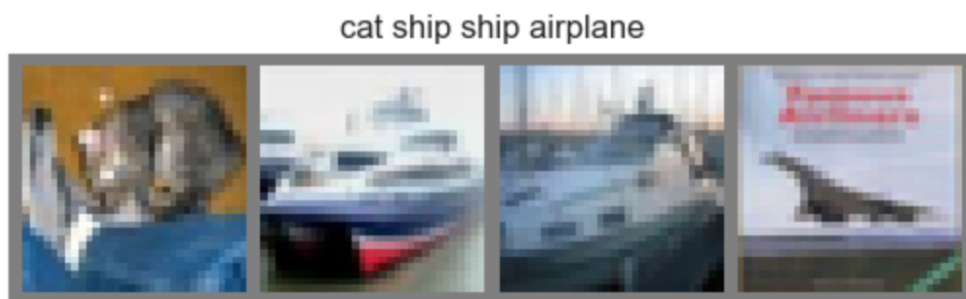
Image shape: torch.Size([3, 32, 32])  
Label: 6

The above line corroborates with the dimensions described by the dataset compilers. Image dimensions match the 32 X 32 X 3, which is expected of CIFAR-10 Image dataset.

Example images are gleaned from the dataset for further visual representation of the dataset.

```
[51]: # Display a batch of test images
test_images, test_labels = next(iter(testloader))
img_test = torchvision.utils.make_grid(test_images)
img_test = img_test * 0.5 + 0.5 # unnormalize for display

plt.imshow(img_test.permute(1, 2, 0))
plt.title(' '.join(testset.classes[label.item()] for label in test_labels))
plt.axis('off')
plt.show()
```



Class balances are checked to ensure no data class is underrepresented in the training.



Here is the code snippet used to perform this visualization:

```
targets = [label for _, label in trainset]

plt.figure(figsize=(8,5))

sns.countplot(
    x=targets,
    hue=targets,      # required to use a palette as seaborn.countplot() is changing its API.
    palette="viridis",
    legend=False      # hide redundant legend
)

plt.title("Training Set Class Distribution")
plt.xlabel("Class Index")
plt.ylabel("Count")
plt.show()

print("Classes:", trainset.classes)
```

I was required to add the targets instance due to seaborn, countplot() changing the API. To keep this code reproducible, the palette is hard-coded.

The classes are balanced, according to the dataset compilers, but it is checked to get its visual representation.

The CIFAR-10 dataset is then checked for any missing values using the following code snippet:

```
missing_count = 0
corrupted_count = 0

for i in range(50000): # check entire training set - because confirmed above 5000 images in each class.
    img, _ = raw_trainset[i]
    img_np = np.array(img)

    if img_np is None:
        missing_count += 1

    # Corrupted Files = all pixel values are likely to show up as more identical in a corrupted file,
    # no diversification in pixel values does not make an image.
    if np.all(img_np == img_np[0,0,0]):
        corrupted_count += 1

print("Missing images:", missing_count)
print("Corrupted images:", corrupted_count)
```

```
Missing images: 0
Corrupted images: 0
```

The dataset compilers and PyTorch assures there are no missing or corrupted values, but the check is performed anyway.

**STEP 2: Feature Selection** – CNNs do not require feature selection as they perform this task themselves. The images were normalized upon import. Images were subsequently unnormalized for display purposes, without touching the actual dataset so re-normalization is not required.

```
# Transform includes ToTensor + normalization

transform = transforms.Compose([
    transforms.ToTensor(), # converts 0-255 PIL image to [0,1] tensor
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)) # scale to [-1,1]
])
```

### STEP 3: Train the Model – Building the CNN model

Inspired by the code for building a CNN model from scratch from GeeksforGeeks using TensorFlow. The code has been modified to PyTorch CNN as Kera cannot be used due to PyTorch import.

This task is being performed on an Apple MacBook Pro running Apple Silicon M2 chip which has GPU acceleration.

To take advantage of this and get this over with faster, this code snippet is used to

```
device = torch.device("mps" if torch.backends.mps.is_available() else "cpu")  
print("Running on:", device)  
Running on: mps
```

A MacBook Pro M2 provides an excellent advantage when training PyTorch models, because it supports **Apple's MPS (Metal Performance Shaders)** backend.

MPS is stable for forward/backward training.

## Setting up the Training Loop

The CNN architecture consists of three convolutional blocks, each containing two convolutional layers with batch normalization and ReLU activation, followed by a max-pooling layer to reduce spatial dimensions.

After the convolutional blocks, the feature maps are flattened and passes through a fully connected dense layer. Dropout is applied after flattening and after the dense layer to reduce overfitting.

The final layer outputs 10 logits corresponding to the CIFAR-10 classes, and the PyTorch CrossEntropyLoss function is used to compute the loss, which internally applies the softmax activation.

## Setting up the Training Loop

```
epochs = 30
train_losses = []
train_acc = []

for epoch in range(epochs):
    model.train()

    running_loss = 0.0
    correct = 0
    total = 0

    # tqdm progress bar for the batches
    loop = tqdm(trainloader, leave=True)
    loop.set_description(f'Epoch [{epoch+1}/{epochs}]')

    for images, labels in loop:
        images, labels = images.to(device), labels.to(device)

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        _, predicted = outputs.max(1)
        total += labels.size(0)
        correct += predicted.eq(labels).sum().item()
```

```
    # Update tqdm bar description
    loop.set_postfix({
        "loss": f"{running_loss / (total/labels.size(0)):.4f}",
        "acc": f"{100 * correct / total:.2f}%"
    })

    epoch_loss = running_loss / len(trainloader)
    epoch_acc = 100 * correct / total

    train_losses.append(epoch_loss)
    train_acc.append(epoch_acc)

    print(f"\nEpoch {epoch+1}/{epochs} | Loss: {epoch_loss:.4f} | Acc: {epoch_acc:.2f}%\n")
```

```
Epoch [1/30]: 100%|█| 12500/12500 [03:27<00:00, 60.32it/s, loss=0.7697, acc=73.7
```

```
Epoch 1/30 | Loss: 0.7697 | Acc: 73.76%
```

```
Epoch [2/30]: 80%|██| 9943/12500 [02:50<00:46, 55.50it/s, loss=0.6681, acc=77.17
```

Training loss and accuracy were monitored and plotted to identify convergence behaviour and possible overfitting.

## Plotting the Training Curves

```
plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
plt.plot(train_losses, label="Training Loss")
plt.title("Loss Curve")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()

plt.subplot(1,2,2)
plt.plot(train_acc, label="Training Accuracy")
plt.title("Accuracy Curve")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.legend()

plt.show()
```

This step will help in diagnosing potential biases in terms of overfitting or underfitting.

Monitoring loss functions are crucial in guiding the optimization process. The losses help determine how well the model learns and generalises.

The loss represents the discrepancy between the predicted output of the model and the actual target value.

During training, the model is attempting to minimize this loss by adjusting their weights (GeeksforGeeks, 2025) .

**Training loss** and **validation loss** are **two key metrics** used to monitor the model's performance and generalization ability.

**Training loss** refers to the **error** on the data the **model was trained on**.

**Validation loss** is the **error on unseen data**, used to **evaluate the model's** performance outside the training dataset (GeeksforGeeks, 2025).

**STEP 4: Evaluate the Model**

Plot training curves are monitored over training to evaluate performance.

```
plt.figure(figsize=(12,4))

plt.subplot(1,2,1)
plt.plot(train_losses, label="Training Loss")
plt.title("Loss Curve")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()

plt.subplot(1,2,2)
plt.plot(train_acc, label="Training Accuracy")
plt.title("Accuracy Curve")
plt.xlabel("Epoch")
plt.ylabel("Accuracy (%)")
plt.legend()

plt.show()
```

The above code is a snippet of the Training Loss and Accuracy Curve plot for the training of the model.

### Confusion Matrix and Classification Report

```
cm = confusion_matrix(true, preds)

plt.figure(figsize=(10,8))
sns.heatmap(cm, annot=False, cmap="Blues")
plt.title("Confusion Matrix")
plt.show()

print(classification_report(true, preds, target_names=classes))
```

## **CONCLUSION**

The task to create an image recognition model for the publishing house has been achieved using the CIFAR-10 image dataset and the employment of a deep layer CNN.

Through exploratory analysis, the data was adequately represented using plots and sample images to get a better view of the data.

Using the well-designed architecture, the model was trained on Apple Silicon M2 at 30 epochs with the resulting model having strong generalization abilities.

The Evaluation results confirms robust, multi-class performance. This model now serves as a solid foundation for the development of facial image recognition.

For future improvements, hyperparameter tuning at 10 epoch for speed and comparing to 30 epochs performance in terms of evaluation metrics and test whether a longer testing time did contribute to the overall accuracy score improvement.

## **REFERENCES**

GeeksforGeeks (2024) Understanding python pickling with example, GeeksforGeeks.  
Available at: <https://www.geeksforgeeks.org/python/understanding-python-pickling-example/>  
(Accessed: 20 November 2025).

GeeksforGeeks (2025a) CIFAR-10 image classification in tensorflow, GeeksforGeeks.  
Available at: <https://www.geeksforgeeks.org/deep-learning/cifar-10-image-classification-in-tensorflow/> (Accessed: 20 November 2025).

GeeksforGeeks (2025b) Image classification using CNN, GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/machine-learning/image-classifier-using-cnn/> (Accessed: 19 November 2025).

Krizhevsky, A., Nair, V. and Hinton, G. (2009) 'Chapter 3', in Learning Multiple Layers of Features from Tiny Images. Toronto: Department of Computer Science, University of Toronto, pp. 32–33. Available at: <https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf> (Accessed: 19 November 2025).