**CNN for Image Classification**

Luke Henry, Joelle Godwin, and Rachel Verhoef

Shippensburg University

[lh2565@engr.ship.edu](mailto:lh2565@engr.ship.edu) [jg2219@engr.ship.edu](mailto:jg2219@engr.ship.edu) [rv0736@engr.ship.ed](mailto:rv0736@ship.edu)

**ABSTRACT**

We utilize a Convolutional Neural Network (CNN) to classify a set of images. The network extracts defining features of the input image through convolutional and max pooling layers. Then, the image is processed through a fully-connected layer to combine the extracted features to produce a final, classified image. The objective of this project is to explore the use of a CNN and understand the factors that aid the classification of the input images. Results showed the loss and accuracy of the classified images and how well the model performed.

**KEY WORDS**

CNN \* Image Classification \* PyTorch

1. **Introduction**

Convolutional Neural Networks (CNNs) are used widely for tasks in image classification. The process of assigning a label or a category to an image based on its content is referred to as image classification. CNNs are designed to automatically extract and learn features from images, to then be used to make predictions about the image’s class. The architecture of a CNN typically consists of multiple layers, including convolutional layers, pooling layers, and fully connected layers. Convolutional layers perform the task of extracting features in the image, such as edges or corners. Its output is then passed through the pooling layers to reduce the size of the feature maps while preserving their important features. The fully connected layer finally takes the extracted features from these previous layers and uses them to form a prediction about the image’s class. A softmax layer is typically connected to these layers to produce a probability distribution over the possible classes. During training, a large dataset of labeled images is presented to the CNN, and learns to adjust the weights of its neurons to minimize the difference between its predicted output and the true label of each image. After the CNN has been trained, it can then be used to classify new, unseen images by applying the same set of operations to the input image, producing a predicted class label. All in all, for image classification, CNNs are powerful tools to use as they can automatically learn to extract useful features from images without the need for human intervention for feature engineering.

1. **Background**

For a better understanding of what ideologies are being applied to this project, the following provides background information that aided our own understanding and motivated our research on the topic of Convolutional Neural Networks (CNN) for the task of image classification.

**2.1 Convolutional Neural Networks (CNNs)**

Convolutional neural networks (CNNs) are comparable to traditional artificial neural networks (ANNs) in that they comprise of neurons that self-optimize through learning [1]. Where each neuron receives an input and performs an operation which is the basis for countless ANNs. CNNs are typically utilized for classification and computer vision tasks. They provide a more scalable approach to image classification and object recognition tasks, leveraging principles from linear algebra, specifically matrix multiplication, to identify patterns within an image. Include three types of layers: Convolutional layer, pooling layer, and a fully-connected layer. A simplified CNN architecture for image classification is illustrated in Fig. 1.

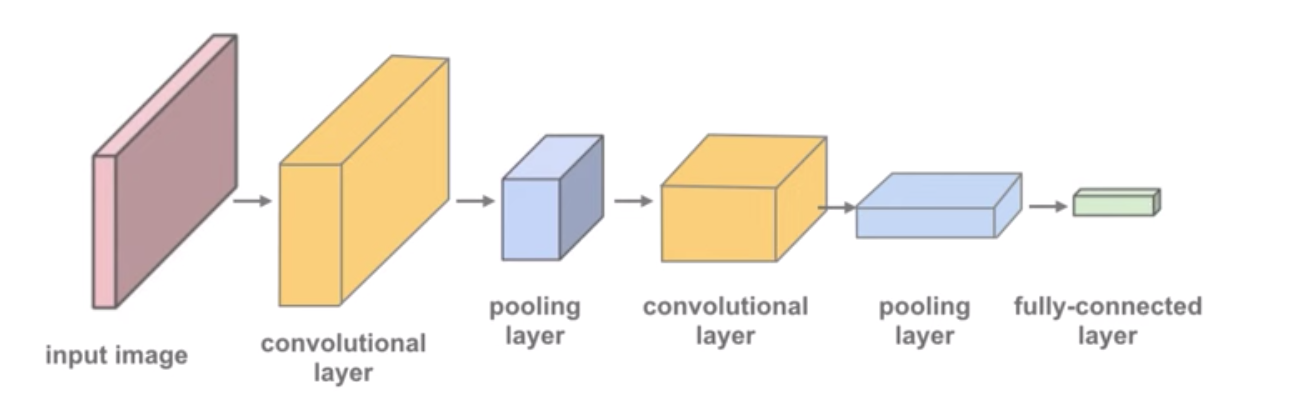


Figure 1: A simple CNN architecture

**2.1.1 Convolutional Layer**

The convolutional layer determines the output of neurons, or feature maps, which are connected to local regions of the input through the calculation of the scalar product between their weights and the region connected to the input volume [1]. The parameters of this layer focus on the use of learnable kernels, usually small and odd numbered in spatial dimensionality but spreads the entire depth of the input [1]. For each value in the kernel, the scalar product is calculated. Looking at Fig. 2, the network will learn of the kernels that signal when a specific feature, given at a spatial position, of the input, properly known as activations.

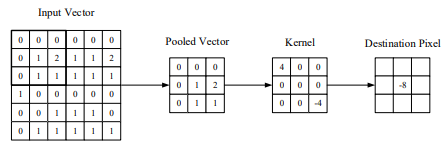


Figure 2: A visual representation of a convolutional layer.

The center element, depicted in Fig. 3, of the kernel is placed over the input vector, to which is then calculated and replaced with a weighted sum of itself and any nearby pixels [1]. This layer also has the ability to significantly reduce the complexity of the model through the optimization of its output, through hyperparameters, depth, stride, and setting zero-padding. The depth of the output can be manually set through the number of neurons to the same region of the input. To define the stride, we have to set the depth around the spatial dimensionality of the input in order to place the receptive field. Zero-padding is the simple process of padding the border of the input, and is an effective method to give further control as to the dimensionality of the output volumes [1]. Included in this layer is the rectified linear unit (ReLU) which aims to apply an ‘elementwise’ activation function, such as sigmoid, to the output of the activation produced by the previous layer.

**2.1.2 Pooling Layer**

Furthermore, the pooling layers are used to down-sample the feature maps that are produced from the previous convolution operations [5]. Taking the larger feature maps and shrinking them to a lower size, while simultaneously preserving the dominant features. Similar to the convolution operation, it is performed by specifying the pooled region size and the stride of the operation. There are different types of pooling techniques that can be used in separate layers. Such as max pooling, min pooling, average pooling, etc. Max pooling is one of the more popular and mostly used pooling methods.

**2.1.3 Fully-Connected Layer**

The fully connected layer contains neurons of which are directly connected to the neurons in the two adjacent layers, without being connected to any layers within them [1]. It is the last layer before the output layer, also known as the ‘dense’ layer. This layer is used to convert the features extracted by the convolutional and pooling layers into class scores, probabilities, or other values. Receiving a one-dimensional vector of features that have been flattened from the output of the previous layers. The amount of neurons in this layer is typically much smaller than those in previous layers. Each neuron in this layer calculates a weighted sum of the inputs from the previous layer, adds a bias term, and applies an activation function, such as ReLU (rectified linear unit) or the sigmoid function, to produce the output of the neuron. Weights and biases are learned during the training process, using backpropagation and gradient descent. To which the output of this layer is passed to the actual output layer, producing the final product of the network.

**2.2 PyTorch**

PyTorch is an open-source machine learning framework used for training deep neural networks. It provides a way to build simple neural networks and train them efficiently [4]. For example, torch.nn is a module for building neural networks. This abstraction makes it easier to define and train deep learning models, compared to lower-level frameworks like TensorFlow. PyTorch provides a number of modules and functions specifically designed for building and training CNNs, the most important modules include:

1. **torch.nn.Conv2d():** Implements a 2-dimensional convolutional layer, takes a tensor representing an image as input and applies a set of learnable filters to the image to produce a set of output feature maps.
2. **torch.nn.MaxPool2d():** Implements a 2-dimensional max pooling operation, used to downsample the output of a convolutional layer. It reduces the spatial resolution of the feature maps while preserving the most important features.
3. **torch.nn.Linear():** Implements a fully connected layer used to produce a final output prediction based on the extracted features from the convolutional layers.

Apart from these modules, PyTorch provides a number of other tools for working with CNNs, including loss functions, optimizers, and data loaders for loading and preprocessing image data. We took advantage of this wide range of tools to build our CNN.

1. **Build CNN Model**

Using the information gathered, we developed a model for a convolutional neural network that properly classified a set of images. Our first task was to gather a relevant data set to process through our CNN.

**3.1 Dataset**

The data set we chose was the CIFAR-10 data set [2], which consists of 60000 total 32x32 images. The images consist of 10 different categories: airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. Using this data set we can apply a number of tools, modules, and base classes from PyTorch to build our CNN.

First, we start by splitting the data set into training and validation sets in order to evaluate the model’s performance as it trains and adjust its hyperparameters. We simply used the **random\_split()** tool to accomplish this task. We also used another tool from PyTorch, called **DataLoader()**, to aid in loading and preprocessing the data, it takes in a dataset object as input and a number of other optional arguments to use for loading the data in parallel. Once we had successfully loaded, preprocessed, split, and trained our data we then were able to create the classes and layers of our model.

**3.2 Create Classes**

We utilized a tutorial [3] to aid our creation of our CNN. Using the tools we previously explored in PyTorch, we can create our classes that build our Convolutional Neural Network. The following breaks down what our classes do and how they work.

**ImageClassificationBase:** Provides the framework for implementing image classification models. Provides a number of useful methods and attributes that can be overridden or extended to implement a specific model architecture. The methods we used from this class include:

1. **\_\_init\_\_():** called when an instance of the model is created, initializes the models layers and parameters.
2. **forward():** implements the forward pass of the model, taking in a batch of images as input and applying the model’s layers to produce a set of output predictions.
3. **training\_step():** implements a single training step for the model, taking in a batch of training examples as input and updates the model’s parameters based on the loss computed by the criterion and the gradients computed by the backward pass.
4. **validation\_step():** implements a single validation step for the model, taking in a batch of validation examples and computes the validation loss and accuracy.
5. **validation\_epoch\_end():** used to compute and aggregate the validation metrics (such as accuracy, precision, recall, and F1 score) across all validation batches in the epoch. Provides a way to monitor the performance of the model on a validation set during training and to make decisions about the hyperparameters or other training settings based on that performance.

**Cifar10CnnModel:** This class is responsible for defining the layers within our CNN, our code depicted in Fig. 3.



Figure 3: Code illustrating the initialization of our CNN

It inherits information from the **ImageClassificationBase** class, providing a basic structure for defining image classification.

1. **\_\_init\_\_():** initializes our model
2. **super()\_\_init\_\_():** calls the initialization function of the parent **ImageClassificationBase** class to ensure that the model inherits all of its functionality.
3. **nn.Sequential():** this is the function that actually defines the architecture of our model. Which includes: six convolutional layers, each followed by a rectified linear unit (ReLU) activation function. After every 2 convolutions it is then followed by a max pooling layer. The convolution layers start with 3 input channels, 32 output channels, a kernel size of 3x3, a stride of 1, and a padding of 1. The last convolution layer differs by consisting of 256 input channels and 256 output channels. After the convolution and pooling layers, we flatten the output feature vector to then use it in our fully-connected layers.
4. **forward():** this method specifies the forward pass of the model, taking in a tensor ‘xb’ and applies the layers within the network to that tensor, returning an output tensor

**DeviceDataLoader:** This is a wrapper class that takes in a PyTorch data loader object and a device, either GPU or CPU, on which to load the data. Provides an iterable interface that yields batches of data after moving them to the device that was specified. It provides an easy way to load batches of data from a data loader object and move then to a specified device for training.

1. **\_\_init\_\_():** stores the data loader object and specified device as attributes of the class instance.
2. **\_\_iter\_\_():** the class instance is made iterable, where each batch of data from the data loader object is loaded to the specified device using the **to\_device()** function, moving the tensors to the device. Then the batches are yielded one at a time.
3. **\_\_len\_\_():** this returns the number of batches in the data loader.
4. **Results**

Before evaluating our results we first had to train our model using a previously specified training set then calculate the loss and accuracy of the trained model

**4.1 Train and Evaluate the Model**

We started off training the model one at a time and evaluating the accuracy of the trained model, depicted in Fig. 4. We used a **fit()** function to fit and train the model on the specified training set and evaluate it on the validation data set for a total of 8 epochs. To evaluate our trained model we simply used the **evaluate()** function to calculate the loss and accuracy percentages of the trained model using our validation data loader set.

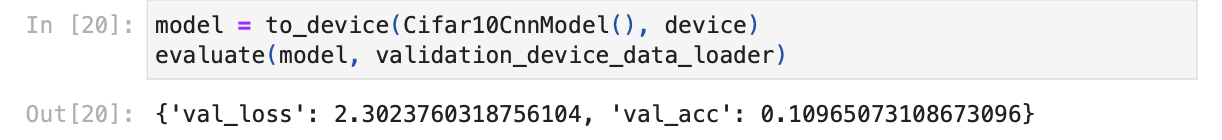


Figure 4: Code illustrating the initial loss and accuracy of the trained model

After initial training, we discovered that fitting the model, 10 consecutive times, took roughly 45 minutes, acquiring a 75% accuracy score, depicted in Fig. 5.

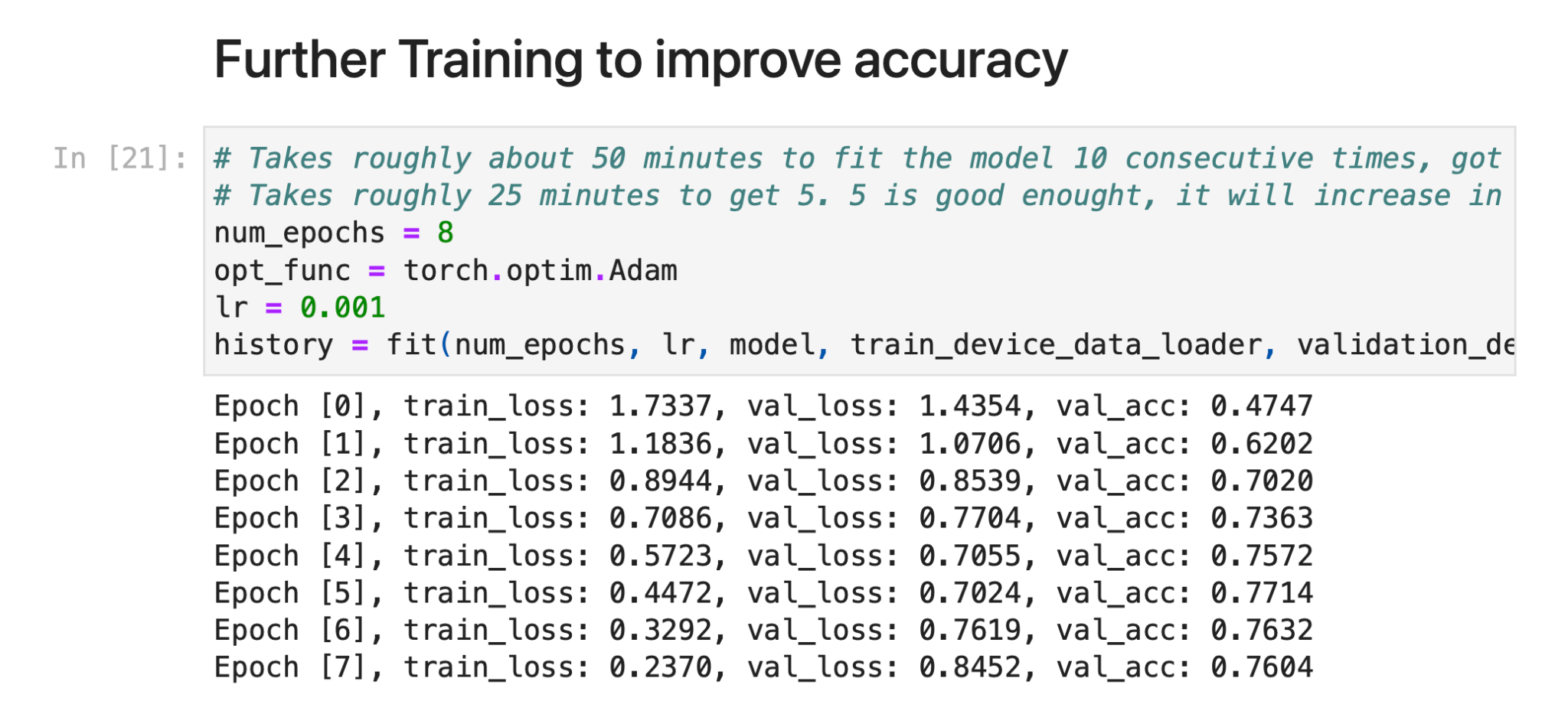


Figure 5: Code illustrating further training and accuracy evaluation

At its best, our model was able to reach an accuracy of around 78%. With further evaluation of a graph, illustrated in Fig. 6, it seemed that our model was unlikely to achieve an accuracy higher than 80% even after training for a long time.

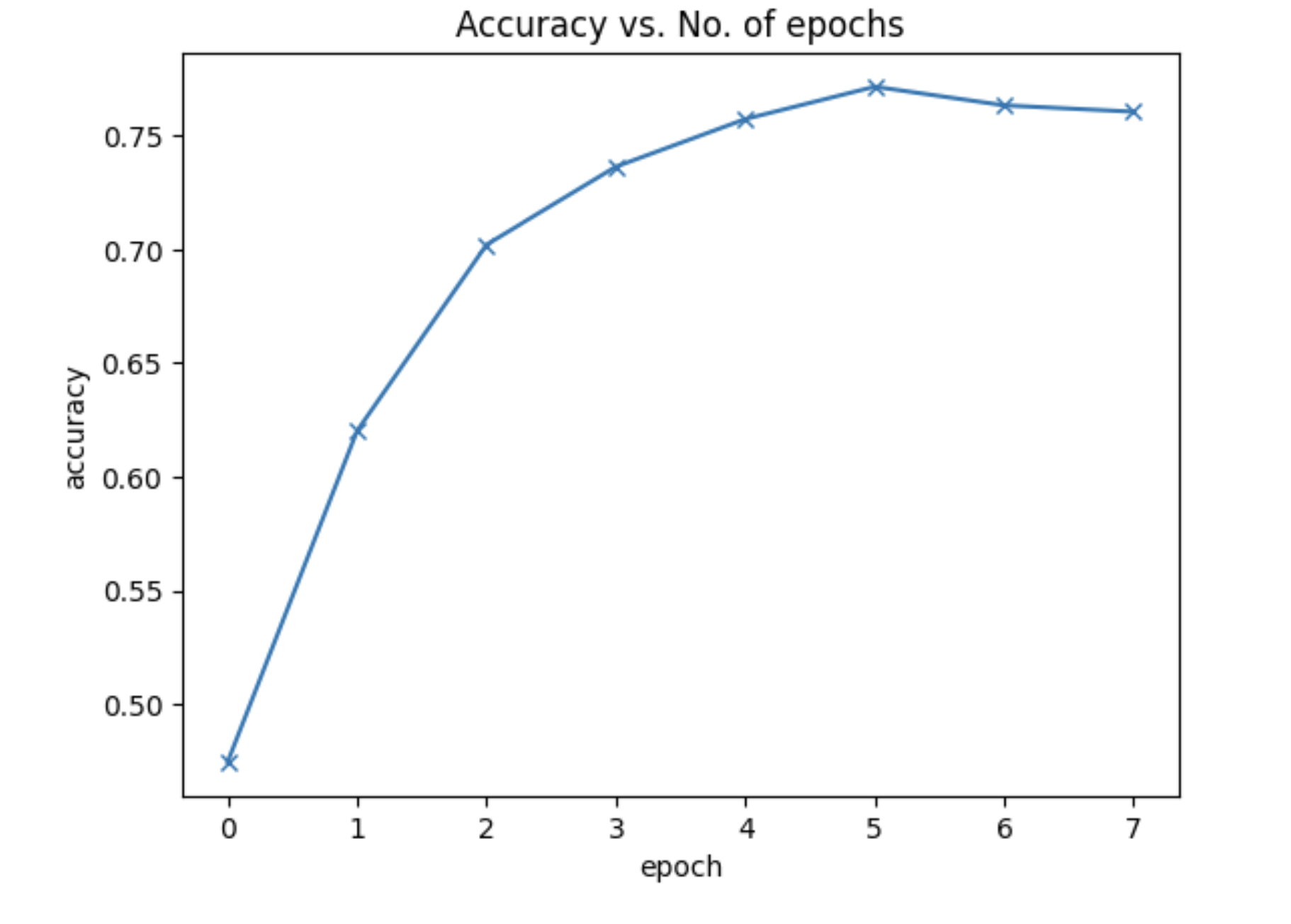


Figure 6: Graph depicting accuracy vs. the number of epochs

It seemed that initially both the training and validation accuracies decrease over time. This can be defined as overfitting and is one of the main reasons why many machine learning models obtain rather terrible results on real-world data. This occurs because the model, in an attempt to minimize the loss, starts to learn patterns that are unique to the training data, sometimes even memorizing specific training examples. Due to this, the model does not generalize well to previously unseen data.

1. **Image Recognition Test**

Our image recognition tests utilized the Convolutional Neural Network to identify and recognize complex patterns in images. Our pool of photos, illustrated in Fig. 7, consisted of a wide range of various pictures, featuring animals and cars. This dataset has images that have different angles, lighting, and colors in order to accurately train the model in familiarizing itself with a myriad of different concepts.

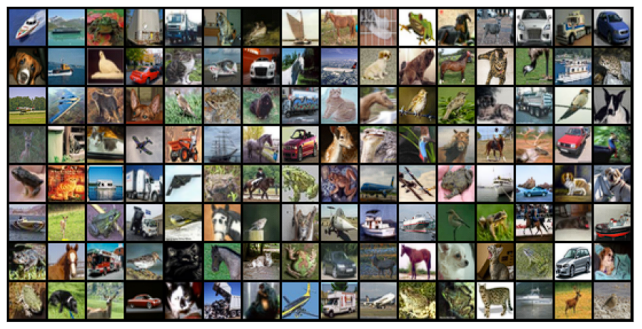


Figure 7: Photos used in the Image Recognition Test

Besides from the basic distinction of animals and cars within the dataset, our pool of photos became more specific. We wanted the model to not only know the difference between an animal and a car, but the difference between a cat and a dog, or a ship and a truck.

After putting the model through 10 iterations of randomly generated images from the dataset, it showed signs of higher accuracy. Within the first iteration, the model inaccurately predicted two images it was given, but those mistakes became nearly non-existent by the ninth and tenth iteration.

Naturally, the more epochs the model goes through, the higher its accuracy was. After feeding it 10 individual photos, it was able to correctly identify all of them.

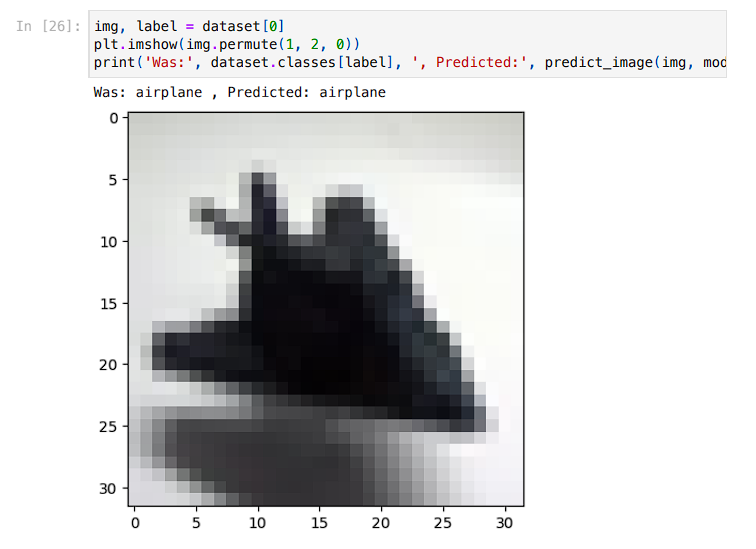


Figure 8: Code illustrating the models’ correct prediction of an airplane

1. **Overview of Findings**

In comparison to the previous labs that took place within this course, working with a Convolutional Neural Network (CNN), we found, was by far one of the more complex tasks. We discovered that it is a reliable machine learning algorithm, among many algorithms, that is able to work with images. However, as much as CNNs are reliable, it is quite costly. Further comparison to previous labs, this project consisted of the most hardware and time consuming calculations to attain a decent accuracy score. For example, fitting the model with 8 epochs took 30-45 minutes. As previously stated in our experience, there was a limit to about 80% accuracy in our model. Therefore, the ability for the model to predict every image is not guaranteed no matter how much time was dedicated to training.

1. **Conclusion**

The result of this project is a Convolutional Neural Network (CNN) that classifies a data set of images. The model created consisted of convolutional layers, pooling layers, and fully-connected layers. These layers collectively worked together to learn and classify images in a network. The specifics of this network are modeled based on the background information explored previously. The results showed that a network produces an average of, at most, 80% accuracy for each epoch. With more experimentation and knowledge on the inner workings of CNNs, the model could be improved to attain 100% accuracy.

**References**

[1]O'Shea, K., & Nash, R. (2015, December 2). *An introduction to Convolutional Neural Networks*. arXiv.org.

[2]Faressayah. (2023, February 2). *📷 CIFAR-10 images classification using CNNS (88%)*. Kaggle.

[3]Biswal, A. (2023, April 24). *Convolutional Neural Network tutorial [update]*. Simplilearn.com.

[4]*What is pytorch?* Built In. (n.d.). https://builtin.com/machine-learning/pytorch

[5]Ghosh, Anirudha & Sufian, A. & Sultana, Farhana & Chakrabarti, Amlan & De, Debashis. (2020, January). *Fundamental Concepts of Convolutional Neural Network*. 10.1007/978-3-030-32644-9\_36.