Algorithmic Analysis and CNN Implementation for Cat vs. Dog Classification in PyTorch

Le Duc Bach
University of Science and Technology of Hanoi

Abstract

This report provides a detailed account of the design and implementation of a Convolutional Neural Network (CNN) algorithm using the PyTorch library to address the image classification task of distinguishing between cats and dogs. The algorithm is structured around a standard machine learning workflow, encompassing input data preprocessing, CNN model architecture definition, model training, and performance evaluation. The "Dogs vs. Cats Redux: Kernels Edition" dataset from Kaggle served as the primary data source. The implemented CNN architecture features three convolutional layers, each succeeded by batch normalization, ReLU activation, and max pooling, and is finalized with two fully connected layers for classification. Image augmentation techniques were integral to the training data preparation phase to enhance the model's generalization capabilities. The algorithm was trained over 10 epochs employing the Adam optimizer and the CrossEntropyLoss function. Evaluation on the validation set yielded an accuracy of approximately 82.54

1 Introduction

The core objective of this project is to develop and train an image classification algorithm capable of accurately discriminating between images of cats and dogs. This task is a foundational problem in computer vision, offering a practical platform for understanding the principles of constructing and training deep learning models. The "Dogs vs. Cats Redux: Kernels Edition" dataset from Kaggle, a standard and widely utilized resource for this binary image classification challenge, was selected. The central methodology revolves around a Convolutional Neural Network (CNN). CNNs are a specialized class of deep neural networks that have demonstrated exceptional success in processing and analyzing visual data. Their architecture, inspired by the human visual cortex, enables them to automatically learn hierarchical spatial features from input images. The implementation of this CNN was executed using PyTorch, an open-source machine learn-

ing library favored for its dynamic computation graphs, user-friendly interface, and extensive support for GPU acceleration. This report will concentrate on a detailed analysis of the source code and algorithmic decisions throughout the implementation process, covering: Environment and Library Setup: Analysis of the Python libraries employed and their roles in the algorithm's implementation. Hyperparameter Configuration: Discussion of the significance and impact of hyperparameters on the algorithm's learning process. Data Preparation and Preprocessing in Code: Detailing how the PyTorch code handles data loading, transformations, and augmentations, and the algorithmic rationale behind these steps. CNN Model Architecture Definition in Code: Analyzing how each layer of the CNN is defined and interconnected using PyTorch modules. Loss Function and Optimizer Selection in Code: Explaining the choice and initialization of the loss function and optimizer. Training and Validation Loop Implementation in Code: Describing in detail how the source code executes the model training and evaluation process. Test Set Prediction and Submission File Generation: Analyzing the code for prediction and result exportation.

2 Algorithmic Analysis and Source Code Implementation

2.1 Environment Setup and Utilized Libraries

The CNN algorithm's implementation relies on several Python libraries, each serving a specific function: PyTorch Core (torch, torch.nn, torch.optim, torch.nn.functional): Provides the foundational elements for neural networks. torch handles multi-dimensional tensors with GPU support. torch.nn contains modules for building network layers (e.g., nn.Conv2d, nn.Linear, nn.BatchNorm2d, nn.MaxPool2d). torch.optim offers optimization algorithms like optim.Adam. torch.nn.functional includes utility functions such as the relu activation.

TorchVision (torchvision.datasets, torchvision.models, torchvision.transforms): A companion library to PyTorch for computer vision tasks. Its transforms module is used for image manipulations like resizing, random cropping, horizontal flipping, and tensor conversion.

DataLoader (torch.utils.data.DataLoader, torch.utils.data.Dataset): Essential PyTorch utilities for creating custom datasets by subclassing Dataset and efficiently loading data in batches using DataLoader, which also supports data shuffling and parallel loading.

Pandas: A powerful data manipulation and analysis library, used here for handling the submission file in CSV format.

Matplotlib PIL (Pillow) (mat-

plotlib.pyplot, PIL.Image): Matplotlib is used for plotting and data visualization, such as displaying sample images. PIL (via the Pillow fork) is used for opening, manipulating, and saving image files. OS Zipfile: Standard Python libraries for operating system interaction and working with ZIP archives to extract the dataset. Glob: Used for finding files matching a specific pattern, particularly for creating lists of image file paths.

Seed and Computation Device Configuration: To ensure result reproducibility, a fixed seed (1234) was set for PyTorch's random number generators on both CPU and GPU (if available). The computation device (device) was dynamically set to 'cuda' if a compatible GPU was detected, otherwise defaulting to 'cpu'.

2.2 Hyperparameters and Their Algorithmic Role

The following hyperparameters were defined, playing a crucial role in governing the algorithm's learning process: Learning Rate (lr): 0.001: This parameter controls the step size for weight updates by the optimizer during backpropagation. A value of 0.001 is a common starting point for the Adam optimizer. Batch Size (batch-size): 100: Defines the number of training samples processed in one iteration (forward and backward pass). Using mini-batches balances computational efficiency and gradient estimate stability. Epochs (epochs): 10: An epoch represents one complete pass through the entire training dataset. The model was trained for 10 epochs.

2.3 Data Preparation: Code and Algorithm

Data Source and Extraction: The dataset was provided as ZIP files (train.zip, test.zip). The code utilizes the zipfile library to extract these files into the ../data/ directory. File Listing and Data Splitting: The glob library was used to generate lists of paths to all .jpg files. The train-test-split function from sklearn.model-selection was then employed to divide the initial training list into 80Image Augmentation and Transformation: Transformations were defined using torchvision.transforms.Compose. For Training (train-transforms): Images were resized to 224x224 pixels (transforms.Resize((224, 224))). This is a common input dimension for many CNN architectures. transforms.RandomResizedCrop(224) was applied, which crops a random section of the image and resizes it to 224x224. This algorithm helps the model become less sensitive to the exact object positioning. transforms.RandomHorizontalFlip() randomly flipped images horizontally with a 50% probability, an effective augmentation for datasets where horizontal orientation doesn't alter the class (e.g., cats/dogs). Finally, transforms.ToTensor() converted PIL Image objects (pixel values 0-255) to PyTorch FloatTensors (pixel values 0.0-1.0) and changed the dimension order from HxWxC to CxHxW, the expected input format for Py-

Torch CNNs.

2.4 CNN Model Architecture: Implementation with nn.Module

The CNN model was defined as a class Cnn inheriting from nn.Module. __init__(self) (Constructor): This method defines the network's layers. Convolutional Block 1: An nn.Conv2d layer with 3 input channels (RGB), 16 output channels, a 3x3 kernel, stride 2, and no padding. The stride of 2 performs downsampling. This is followed by nn.BatchNorm2d(16) for batch normalization across the 16 output channels, which helps stabilize and accelerate training. An nn.ReLU() activation function introduces non-linearity. An nn.MaxPool2d layer with a 2x2 kernel and stride 2 further downsamples the feature maps. Convolutional Block 2: Similar structure, taking 16 input channels and producing 32 output channels, with nn.Conv2d, nn.BatchNorm2d, nn.ReLU, and nn.MaxPool2d. Convolutional Block 3: Similar structure, taking 32 input channels and producing 64 output channels. After this block, the feature maps are 3x3 in spatial dimensions. Fully Connected Layers (Classifier Head): The first nn.Linear layer, self.fc1, takes the flattened output from the last pooling layer (64 channels * 3x3 = 576 features) and maps it to 10 output features. A self.dropout = nn.Dropout(0.5) layer was defined but not explicitly used in the model's forward method as presented in the notebook. The second nn.Linear layer, self.fc2, acts as the output layer, mapping the 10 features from fc1 to 2 output logits, representing scores for 'cat' and 'dog'. An nn.ReLU() activation, self.relu, is used after self.fc1. forward(self, x) (Forward Pass): This method defines the data flow through the network. The input x passes sequentially through the three convolutional blocks. The output tensor from the third block is then flattened using out.view(out.size(0), -1) before being passed to the fully connected layers. The dropout layer, though defined, was not called in this path. The model instance was created and moved to the selected computation device using .to(device).

2.5 Loss Function and Optimizer: Selection and Implementation

Optimizer: optim.Adam was chosen. Adam is an adaptive learning rate optimization algorithm well-suited for many problems. It was initialized with the model's parameters and the predefined learning rate. Loss Function: nn.CrossEntropyLoss() was used. This criterion combines LogSoftmax and NLLLoss, making it suitable for multi-class (or binary, with 2 output neurons) classification tasks. It expects raw logits as input from the model and class indices as targets.

3 Algorithmic Results and Evaluation

Training and Validation Performance Analysis: The table below summarizes the algorithm's performance over 10 epochs, based on the executed code:

Epoch	Train Acc.	Train Loss	Val. Acc.	Val. Loss
1	0.7186	0.5486	0.7590	0.4888
2	0.7359	0.5261	0.7684	0.4757
3	0.7498	0.5052	0.7988	0.4418
4	0.7585	0.4889	0.8044	0.4261
5	0.7547	0.4946	0.8028	0.4260
6	0.7631	0.4829	0.8094	0.4113
7	0.7713	0.4712	0.8186	0.4072
8	0.7740	0.4647	0.8176	0.3997
9	0.7763	0.4630	0.8220	0.3912
10	0.7841	0.4517	0.8254	0.3971

Table 1: Training and validation metrics (Epochs 1–5)

The algorithm showed consistent improvement in accuracy on both training and validation sets. The final validation accuracy at Epoch 10 was approximately 82.54Test Set Prediction and Submission File Generation: After training, model.eval() was called to set the model to evaluation mode. The code generated predictions on the test_loader. Raw logits from model(data) were passed through F.softmax(preds, dim=1) to convert them into probabilities. The probability of an image being a dog (class 1) was extracted. These probabilities, along with their corresponding file IDs, were stored. The results were sorted by file ID and saved to result.csv using Pandas. Visualization of Predictions: The code used Matplotlib to display a sample of 10 random images from the test set along with their predicted labels ('cat' or 'dog'), determined by a probability threshold of 0.5.

4 Conclusion

This project successfully demonstrated the construction, training, and evaluation of a Convolutional Neural Network for binary image classification of cats and dogs using PyTorch, with a focus on analyzing the algorithm and its code implementation. The CNN algorithm, despite a relatively simple architecture and 10 epochs of training, achieved a commendable validation accuracy of approximately 82.54The code analysis reveals adherence to good PyTorch practices, from defining a custom Dataset class and an nn.Module model to implementing the training loop. Algorithmic choices such as the Adam optimizer and CrossEntropyLoss were appropriate. While the achieved accuracy is a good starting point, the discussion on limitations and future work outlines several strategies for further enhancing the algorithm's predictive capabilities and optimizing the code implementation. Overall, this project establishes a solid foundation for understanding and applying CNNs to image classification problems, emphasizing the connection between algorithmic theory and programming practice.

References