Report Summary

Abstract

This report details the approach and methodologies employed for the binary image classification of muffins and Chihuahuas. The project aims to evaluate various neural network architectures, leveraging GPU acceleration for efficient model training and evaluation. The primary focus is on the preprocessing techniques and the decisions made during the model development process.

note that this is a summary report, as other details are already included in the notebook.

Implementation Details

The models were implemented using PyTorch and were trained on Google Colab to leverage GPU support for faster training. Essential libraries included torchvision for data transformations and PyTorch for model building and training.

Project Discussion

Data Preparation and Transformation

Before selecting and training our initial model, it was crucial to preprocess the dataset to optimize it for the neural network. The preprocessing steps included transforming images from *JPG* to *RGB* or *grayscale* pixel values and scaling them down to a uniform size. This ensures that all images have consistent dimensions and color channels, which is essential for effective training and model performance.

The dataset was sourced from Kaggle and consisted of images categorized into *train* and *test* sets, each containing images of muffins and Chihuahuas. The dataset included:

- *Total Training Images*: 4733
- *Total Testing Images*: 1184

The testing data constituted *25.02%* of the training data. The training data was further split into training and validation sets using an *80-20* split, resulting in:

- *Training Images*: 3786
- *Validation Images*: 947

Data Transformation and Normalization

The images in the dataset underwent several transformations to prepare them for model training. Initially, each image was resized to *256x256* pixels to ensure uniformity in dimensions across the dataset. This resizing is crucial as neural networks require consistent input sizes for effective training.

Following resizing, the images were converted to *PyTorch* tensors. This conversion facilitates efficient matrix calculations, which are the foundation of neural network operations. Once in tensor form, the pixel values of the images were normalized. Normalization was performed to

have a mean of 0 and a standard deviation of 1, a common practice in deep learning to stabilize and speed up the training process. This normalization involved two main steps: scaling the pixel values to a range of *[0,1]* and then adjusting these scaled values to achieve the desired mean and standard deviation.

To enhance the generalization capabilities of the models and reduce the risk of overfitting, data augmentation techniques were applied exclusively to the training dataset. These augmentations included random horizontal flipping and random rotation. Horizontal flipping simulates different viewing angles, making the model robust to various orientations of the objects in the images. Similarly, random rotation introduces variability in image orientation, further aiding the model's ability to generalize from the training data.

For the validation and testing datasets, data augmentation was deliberately not applied. This decision ensures that the performance evaluation of the models is conducted on unaltered images, providing an accurate measure of their real-world applicability and generalization capability.

Model Selection

Given the binary nature of the classification task, we experimented with different deep learning architectures, focusing primarily on Convolutional Neural Networks (CNN's) due to their efficacy in computer vision tasks. We utilized Torch's powerful gradient tools to facilitate the training process. Our approach involved starting with a simple CNN to establish a baseline performance.

Training Details

- Loss Function: Cross Entropy Loss, which includes *nn.LogSoftmax* and *nn.NLLLoss.*
  - *nn.LogSoftmax*: This layer applies the log-softmax function to the input tensor. Log-Softmax is a variant of the softmax function that converts logits (raw, unnormalized scores) into log-probabilities, ensuring numerical stability and making it easier to compute the log-likelihood.
  - *Mathematically, for an input tensor* $\mathbf{z}$*, the log-softmax function is defined as*

$$LogSoftMax(\mathbf{z})_i = \log\left(\frac{e^{z_i}}{\sum_j e^{z_j}}\right) = z_i - \log\left(\sum_j e^{z_j}\right)$$

    where z_i *is the i-th element of the input tensor* $\mathbf{z}$*, and the sum* $\sum_j e^{z_j}$ *is computed over all elements j of the input tensor.*

  - *nn.NLLLoss*: Stands for Negative Log-Likelihood Loss. This loss function is used to train classifiers with log-probabilities. It computes the negative log-likelihood of the true class given the predicted log-probabilities, which encourages the model to predict higher probabilities for the correct class.
  - Given the predicted log-probabilities **p** and the true class labels *y*, the *NLLLoss* is computed as

$$NLLLoss(\boldsymbol{p}, y) = -p_y$$

where p_y is the log-probability corresponding to the true label y.

- The overall *Cross-Entropy loss function* in PyTorch combines the above components and can be mathematically expressed as

$$CrossEntropyLoss(\boldsymbol{z}, y) = -\log\left(\frac{e^{z_y}}{\sum_j e^{z_j}}\right)$$

Where $\boldsymbol{z}$ is the input logits, and z_y is the logit corresponding to the true class label y.

Note that the logit is the raw, unnormalized score output by the last layer of the neural network for class i. These logits can be any real number and are converted to probabilities by applying the softmax function.

- Activation Function: only *ReLU* (Rectified Linear Unit) was used during this process.
  - The *ReLU* activation function is applied after each convolutional and fully connected layer. It introduces non-linearity into the model, allowing it to learn complex patterns.
  - Mathematically, ReLU is defined as:

$$ReLU(x) = \max(0, x)$$

where x is the input to the ReLU function.

First Model; Simple CNN

Design

The design of the CNN involved several key components. We employed a modest number of filters initially to capture low-level features, increasing the number of filters in deeper layers to identify more complex patterns. Max pooling layers were used to reduce spatial dimensions and control overfitting by down-sampling the feature maps. We also included fully connected layers; these layers mapped the learned features to the final classification output.

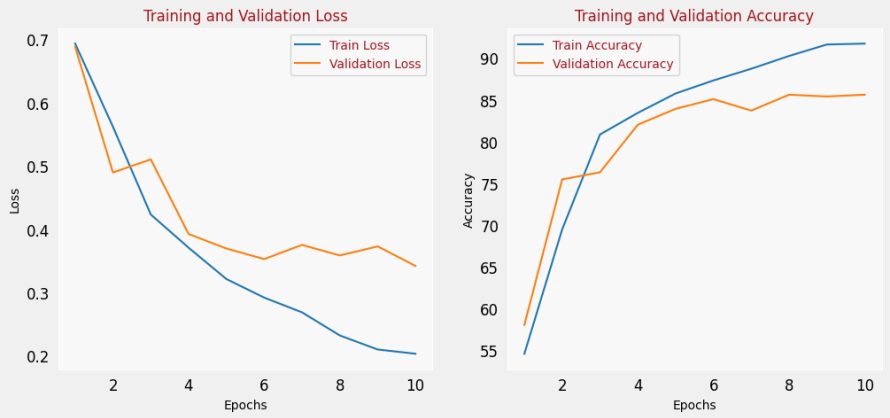
The design of the CNN involved several key components to capture low-level and complex patterns:

- *Convolutional Layers*: *16*, *32*, and *64* filters with *3x3* kernels. Each convolutional layer applies a set of learnable filters to the input image to detect features such as edges, textures, and more complex patterns in deeper layers.
- *Activation Function:* ReLU (Rectified Linear Unit) applied after each individual layer to introduce non-linearity.
- *Max Pooling*: *2x2* pooling applied after each convolutional layer to down-sample the feature maps, reducing their spatial dimensions and controlling overfitting.

- *Fully Connected Layers*: After flattening the feature maps from the final convolutional layer, a fully connected layer maps the learned features to the final classification output. This includes:
    - *FC1:* Input features from the flattened output of the last pooling layer, 512 output features.
    - *FC2:* 512 input features, 2 output features corresponding to the binary classification.

During training, we monitored both training and validation loss and accuracy to ensure the model was learning effectively and not overfitting.

Evaluation



For our simple CNN, we obtained some notable results. Over 10 epochs, the model exhibited consistent improvement in training performance, with a steady decrease in training loss and increase in training accuracy. However, the validation loss initially decreased but then fluctuated, indicating potential overfitting. Validation accuracy increased to a point and then plateaued, suggesting that while the model learned to classify the training data, its performance on unseen validation data could be improved.

Second Model; Deeper CNN

To address the limitations observed with the simple CNN, we designed a deeper network with additional convolutional layers. This model aimed to capture more complex features by increasing depth and incorporating regularization techniques.

Design

The architecture of the Deeper CNN included the following components to capture more complex features and enhance generalization:

- *Convolutional Layers*: *32*, *64*, *128*, and *256* filters (4) with *3x3* kernels and padding of *1*. Each convolutional layer increases in filter size to capture increasingly complex patterns.
- *Batch normalization* was applied after the *64* and *128* filter layers to stabilize and speed up training by normalizing the activations.
- *Activation Function: ReLU* (Rectified Linear Unit) applied after each individual layer.

- *Max Pooling*: *2x2* pooling applied after each convolutional layer to reduce the spatial dimensions of the feature maps and control overfitting by down-sampling the data.
- *Dropout*: Applied after the first fully connected layer for regularization to prevent overfitting.
- *Fully Connected Layers*: The first fully connected layer connected *256 * 16 * 16* input features to *1024* output features, followed by dropout for regularization. The output layer reduced the *1024* features to *2* output classes, suitable for the binary classification task of distinguishing between muffins and Chihuahuas.

This architecture was specifically designed to overcome overfitting issues and improve the model's ability to generalize from the training data to unseen validation data.

Evaluation

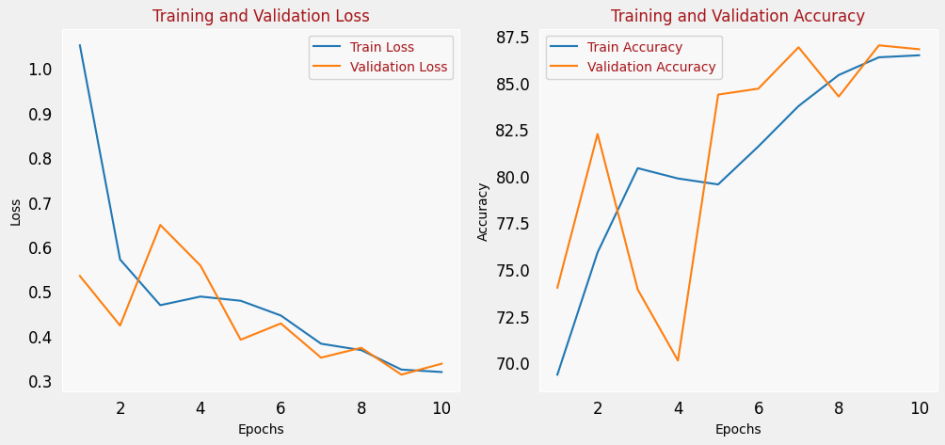
The training loss decreased more gradually compared to the simpler model, starting higher in the initial epoch. This might be due to the increased complexity and the model requiring more time to start generalizing from the data.

The training accuracy started higher and increased at a slower rate, aligning with the behavior observed in the training loss. It did not reach as high as the simpler model, suggesting the deeper model is not overfitting the training data.

The validation loss showed less fluctuation compared to the simpler model and decreased overall, indicating that the model is generalizing better and potentially benefiting from the added complexity and regularization techniques.

The validation accuracy was more stable and consistently increased, surpassing the simpler model by the final epoch. This suggests the model is generalizing well and could potentially perform better on unseen data.

The results from training the deeper CNN model showed that the increased depth and complexity led to a more gradual decrease in training loss and a corresponding increase in training accuracy. Despite a higher initial training loss, the model eventually generalized better to the validation data, indicated by a more stable validation loss and accuracy. These results highlighted the benefits of a more complex architecture and regularization techniques.



Third Model; ResNet-like CNN

To further improve the model's ability to learn complex patterns and reuse features, we designed a ResNet-like CNN. This architecture incorporates residual blocks to facilitate easier gradient flow, enhancing training efficiency and performance for deeper networks.

The architecture of the ResNet-like CNN included the following components:

- *Residual Blocks*: Implementing residual connections helps alleviate the vanishing gradient problem in deep networks by allowing direct paths for gradients during backpropagation. This ensures more efficient training and improved performance.
- *Variable Dropout Rates*: Using dropout with varying rates at different layers helps in understanding the optimal regularization needed at each layer, thereby reducing overfitting.
- *Global Average Pooling*: This technique reduces the spatial dimensions to a single value per feature map before the final fully connected layers. It minimizes the number of parameters, focusing on the most salient features.

Specifically, the architecture of the ResNet-like CNN included:

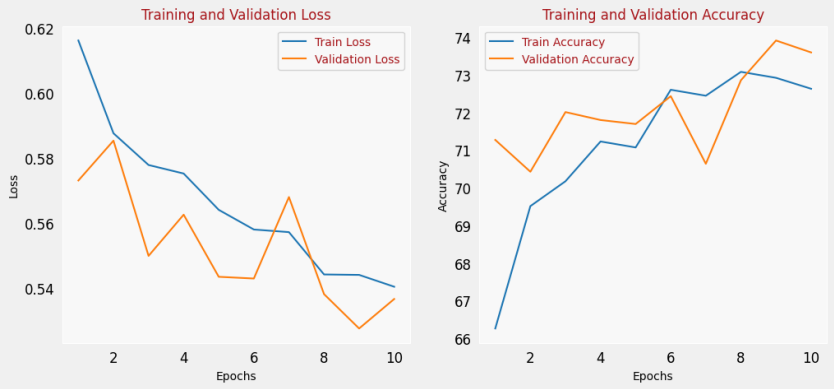
- *Initial Convolutional Layer*: *64* filters with a *7x7* kernel, stride of *2*, and padding of *3*, followed by batch normalization and *ReLU* activation.
- *Activation Function: ReLU* (Rectified Linear Unit) applied after each individual layer.
- *Max Pooling*: *3x3* kernel, stride of *2*, and padding of *1*, applied after the initial convolutional layer to reduce spatial dimensions.
- *Residual Block 1*: Two convolutional layers with *64* filters and *3x3* kernels, batch normalization, and *ReLU* activation. A shortcut connection was added if the input and output dimensions differed.
- *Residual Block 2*: Two convolutional layers with *128* filters and *3x3* kernels, batch normalization, and *ReLU* activation. A shortcut connection was included for dimensionality matching.
- *Global Average Pooling*: Applied before the final fully connected layers to reduce each feature map to a single number.
- *Dropout Layer*: Variable dropout rate applied before the final fully connected layer to enhance regularization.
- *Fully Connected Layer*: *128* input features connected to *2* output classes, suitable for binary classification.

Evaluation

The training loss of the *ResNet-like* CNN displayed a gentle reduction across epochs, punctuated by noticeable volatility. This pattern reflected the model's learning process but hinted at potential challenges in training stability.

The training accuracy climbed modestly but did not reach the heights of the simpler or deeper models, implying a slower and potentially less optimized learning process.

The validation loss experienced swings and a less predictable trajectory compared to the second model, suggesting that while the model improved, its generalization to the validation set was less stable.



The validation accuracy ascended incrementally but failed to achieve the robustness or peak performance of the second model. This could indicate that the ResNet-like model was less adept at deciphering and generalizing the underlying patterns in the validation data.

Comparison

The *Simple CNN*, with *16*, *32*, and *64* filters across its convolutional layers, showed initial promise but struggled with overfitting. Training accuracy improved steadily, but validation accuracy plateaued, and validation loss fluctuated, indicating limitations in generalization.

The *Deeper CNN*, featuring *32*, *64*, *128*, and *256* filters with batch normalization and dropout, demonstrated superior performance. Training and validation accuracies were higher and more stable, with consistent loss reduction. This model's architecture effectively captured complex features and generalized well.

The *ResNet-like* CNN incorporated residual blocks and global average pooling, aiming for advanced feature learning. However, it exhibited fluctuations in loss and accuracy, suggesting instability. While it showed learning capability, further tuning (e.g., more epochs, hyperparameter adjustments) is needed for improved performance.

The *Deeper CNN* outperformed the *ResNet-like CNN* in both training and validation accuracy, proving more effective for this task. The *Deeper CNN's* consistent loss reduction and higher accuracy make it the stronger candidate. The *ResNet-like CNN* requires additional tuning to realize its potential.

5-fold cross-validation

To ensure robust generalization of the *Deeper CNN* model, we employed 5-fold cross-validation, a statistical method used to evaluate the performance of a model. The process involves dividing the dataset into five equal-sized subsets or folds. Each fold is used once as a validation set while the remaining folds serve as the training set. This rotation ensures that every data point gets to be in a validation set exactly once and in the training set four times.

Mathematically, the procedure can be described as follows:

1. **Dataset Splitting**:
   Let the dataset D consist of N samples. The dataset is divided into 5 folds $D1, D2, D3, D4, D5$, each containing $\frac{N}{5}$ samples.

2. **Training and Validation**:
   For each fold i (where i ranges from 1 to 5):
   - Use D_i as the validation set.
   - Use the remaining $D \setminus D_i$ as the training set.

3. **Model Training and Evaluation**:
   For each fold i:
   - Train the model on D/D_i .
   - Evaluate the model on D_i, and record the validation error rate E_i.

4. **Averaging the Results**:
   Compute the average validation error rate across all folds:

$$E_{cv} = \frac{1}{5}\sum E_i$$

   Here, E_{cv} represents the cross-validated error rate, providing a more reliable estimate of the model's performance.

The advantage of this method is that it reduces the variance associated with the random split of the training and validation data. By averaging the error rates across multiple folds, we obtain a robust estimate of the model's generalization error.

Implementation

In this study, the Deeper CNN model was subjected to 5-fold cross-validation for various combinations of hyperparameters (learning rate and batch size).

The training process involves iterating over a specified number of epochs, during which the model undergoes both training and validation phases. In the training phase, the model learns from the training data by performing forward and backward passes to update the weights using gradient descent. The validation phase involves evaluating the model's performance on the validation set, with the key metric being the *zero-one loss* (error rate), which indicates the proportion of incorrect classifications.
Hyperparameter tuning was conducted using a grid search approach, where combinations of learning rates and batch sizes were tested to identify the optimal settings for the Deeper CNN model. The learning rates and batch sizes tested were:

- Learning Rates: 0.001, 0.0001
- Batch Sizes: 64, 128

For each combination, the model's performance was evaluated using 5-fold cross-validation. The goal was to find the combination that resulted in the lowest average validation error rate across all folds.

Evaluation

The DeeperCNN architecture, consisting of convolutional layers, batch normalization, pooling layers, and dropout for regularization, was evaluated using 5-fold cross-validation. The optimal configuration achieved a cross-validated *zero-one loss* of *11.38%* with a learning

rate of *0.001* and a batch size of *64*. The model includes four convolutional layers with filter sizes increasing from *32* to *256*, allowing for the progressive learning of complex features.

Batch normalization layers after the second and third convolutional layers stabilize and accelerate training by reducing internal covariate shift, thereby enhancing generalization. Max pooling layers, following each convolutional set, reduce spatial dimensions and retain essential features, improving efficiency and effectiveness. The dropout layer after the first fully connected layer prevents overfitting by randomly zeroing out neurons, enhancing generalization by preventing reliance on specific features.

Influence of Hyperparameters on Cross-Validated Risk Estimate

The cross-validated zero-one loss was minimized at a learning rate of *0.001*. This learning rate effectively balances convergence speed and model stability, preventing issues associated with lower learning rates (*0.0001*), such as slow convergence and suboptimal solutions, and higher learning rates, which can cause overshooting of minima and instability.

A batch size of *64* yielded the lowest error rate. Smaller batch sizes, by introducing more noise into gradient estimation, act as a form of regularization, improving generalization. In contrast, larger batch sizes (*128*) resulted in higher error rates, likely due to convergence to sharper minima that generalize less effectively.

Final Cross-Validated Risk Estimate

The final cross-validated risk estimate, reflected by a zero-one loss of *11.38%* for the optimal configuration, represents the proportion of misclassifications. This metric directly quantifies the model's classification error rate, offering a precise evaluation of its generalization capability.

Sources:

Dataset used:
https://www.kaggle.com/datasets/samuelcortinhas/muffin-vs-chihuahua-image-classification