Data Preprocessing and Model Building

Farzad Shahrivari

February 19, 2024

Question

You are tasked with building a simple ML model for identity document classification using the provided dataset of images. Use the provided images.zip dataset containing 10 classes, each with 100 examples.

Solution

Introduction

In this task, I employed a deep CNN model structure comprising three convolutional blocks followed by three linear layers. **The neural network's architecture, including parameters such as the number of convolution filters, filter size, and stride, as well as the use of advanced techniques like batch normalization and max pooling, was entirely devised by me**. Leveraging my extensive experience in computer vision and employing trial and error methods, I determined the optimal configuration for constructing these structures.
While the model's design is moderately simple, it yields outstanding accuracy for this task, achieving an **86**% accuracy rate, which aligns well with the project's objectives. However, it's worth noting that employing a more robust architecture like ResNet50 could potentially yield even higher accuracy, as it achieves 98% accuracy on this task. I will delve into potential improvements, including the adoption of ResNet50, in the subsequent sections.
Throughout this document, I aim to elucidate each aspect of my code to provide a comprehensive understanding of my process. It's essential to note that the Python file accompanying this report is computationally intensive to execute. This is primarily due to the high resolution of the images (2268×4032 pixels) and the inherently challenging nature of classifying passport images and ID cards.
On my local device, equipped with an RTX3070Ti GPU boasting 8GB of VRAM, along with CUDA (version 2.0.1) and cuDNN (version 11.7) libraries, executing the code typically takes approximately 5 minutes. Furthermore, this process consumes around 3GB of RAM and utilizes 94% of my GPU's VRAM to function optimally.

Libraries

The libraries utilized for this task include NumPy and Pandas for data loading, TorchVision for data preprocessing, PyTorch and PyTorch Lightning for constructing the CNN model, TorchMetrics for evaluating the model's accuracy, Scikit-learn metrics for streamlining calculations of essential metrics such as confusion matrix, precision, recall, and F1 score. Additionally, Matplotlib and Seaborn were employed to visualize the model's performance.

Data Preprocessing

For data preprocessing, the torchvision library was utilized, specifically the transform module, to perform necessary adjustments to the dataset. This process serves three primary purposes. Firstly, the dataset comprises images of extensive dimensions (2268×4032 pixels), leading to computational complexities and potential overfitting due to the capture of redundant features by the neural network. To mitigate this, the images were resized to a standard size of 224 by 224, which is commonly recommended for various image classification tasks, including the classification of the ImageNet dataset. Secondly, raw pixel values do not inherently convey feature importance to machine learning models. To address this issue, the data underwent normalization, wherein the pixel values were standardized to be more conducive for training tasks. This normalization process involves subtracting the mean values and dividing by the standard deviation for each channel in the image data. Lastly, the images were converted into tensors, facilitating compatibility with PyTorch for classification tasks. To visualize the processed images suitable for training of data, please see the following figure. Additionally, the dataset was partitioned into training, validation, and test

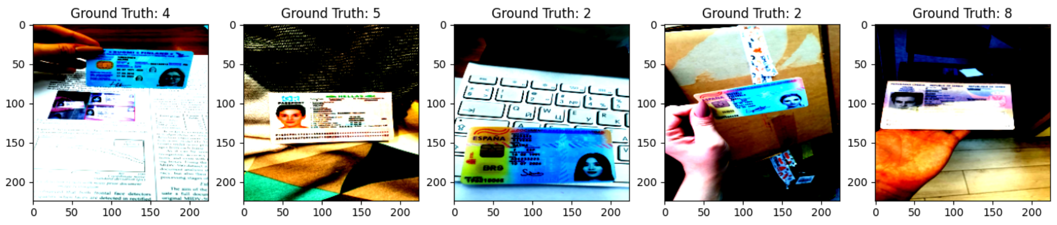
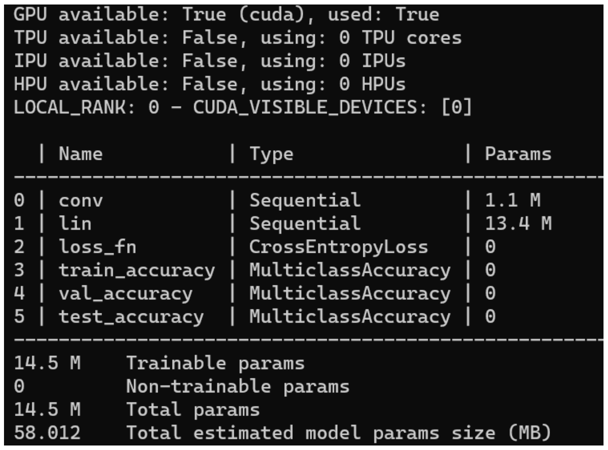


Figure 1: Visualising a mini-batch that the training data-loader gives us

sets. Approximately 70% of the total data was allocated for training, 10% for validation, and 20% for testing. Careful shuffling was employed during this process to ensure the model's consistency in evaluation metrics across different subsets.

CNN Model

The model begins with a series of convolutional layers, followed by batch normalization and rectified linear unit (ReLU) activation functions to enhance feature representation. The initial convolutional layers, consisting of 3x3 kernels with padding, gradually increase the number of channels from 3 to 256, capturing hierarchical features at different levels of abstraction. Max-pooling operations are interspersed between convolutional layers to down-sample the feature maps, reducing spatial dimensions and enhancing computational efficiency. Furthermore, dropout layers with a dropout rate of 0.05 are incorporated after the second max-pooling operation to prevent over-fitting by randomly disabling a fraction of neurons during training. This regularization technique aids in improving the generalization capabilities of the model. Following the convolutional layers, the network transitions to fully connected layers to perform classification. These linear layers, connected via dropout regularization, progressively reduce the dimensionality of the feature representations extracted by the convolutional layers. The final linear layer outputs predictions for the classification task, mapping the learned features to the corresponding class labels. Overall, this CNN architecture effectively captures intricate patterns within the input images and demonstrates promising performance in classification tasks.



```
GPU available: True (cuda), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]

  | Name           | Type               | Params
-----------------------------------------------------
0 | conv           | Sequential         | 1.1 M
1 | lin            | Sequential         | 13.4 M
2 | loss_fn        | CrossEntropyLoss   | 0
3 | train_accuracy | MulticlassAccuracy | 0
4 | val_accuracy   | MulticlassAccuracy | 0
5 | test_accuracy  | MulticlassAccuracy | 0
-----------------------------------------------------
14.5 M    Trainable params
0         Non-trainable params
14.5 M    Total params
58.012    Total estimated model params size (MB)
```

Figure 2: Deep CNN structure

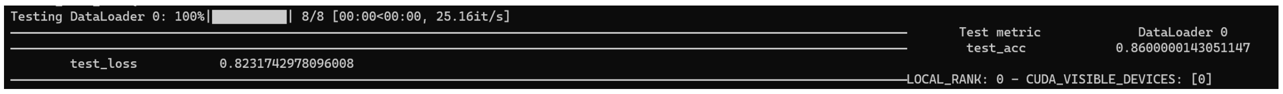
Hyperparameters

Let's delve into the selection process for the model's training parameters, commonly referred to as hyperparameters. Firstly, the choice of a batch size of 25 warrants discussion. This decision stems from a careful consideration of several factors. By opting for a batch size of 25, we strike a balance between sampling diversity and computational efficiency. This size ensures each training batch contains a representative mix of samples from our dataset, which consists of 10 classes with 100 images each. Maintaining this balance is crucial for preserving the integrity of the training process while optimizing computational resources.
Next, let's explore the rationale behind the selection of the optimizer. Stochastic Gradient Descent (SGD) emerged as the preferred optimizer for this task. This decision is underpinned by SGD's established efficacy in the domain of image classification. Through empirical experimentation, SGD consistently outperformed alternative optimizers such as Adam. To fine-tune the optimization process, a learning rate of 0.007 and a momentum of 0.9 were

meticulously chosen. Leveraging momentum in conjunction with a judiciously chosen learning rate enhances the optimization process's efficiency and convergence properties.
Lastly, let's consider the training regimen, encompassing the number of epochs and the incorporation of early stopping mechanisms. The model was trained over a span of 30 epochs, allowing it to iteratively refine its parameters through exposure to the training data. To safeguard against overfitting, a critical concern in machine learning, early stopping was integrated with a patience threshold of 10 epochs. This safeguard ensures that training halts prematurely if the model demonstrates signs of overfitting, thereby preserving its ability to generalize effectively.

Results

The performance of the model was assessed through various metrics and visualizations. The overall accuracy of the model on the test set was found to be 86%, accompanied by a test loss of 0.82. Cross-entropy loss function was utilized for this classification task, deemed suitable for such applications. Notably, the highest accuracy of the model was achieved at epoch number 17.



```
Testing DataLoader 0: 100%|██████████| 8/8 [00:00<00:00, 25.16it/s]
──────────────────────────────────────────────────────────────────────────       Test metric          DataLoader 0
──────────────────────────────────────────────────────────────────────────        test_acc         0.8600000143051147
        test_loss           0.8231742978096008
──────────────────────────────────────────────────────────────────────────LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES: [0]
```

Figure 3: Observing model accuracy and model loss function on a test data

Additionally, visual representations of the model's training and validation progress throughout epochs were generated, depicting the trends of validation and training losses, as well as validation and training accuracies.

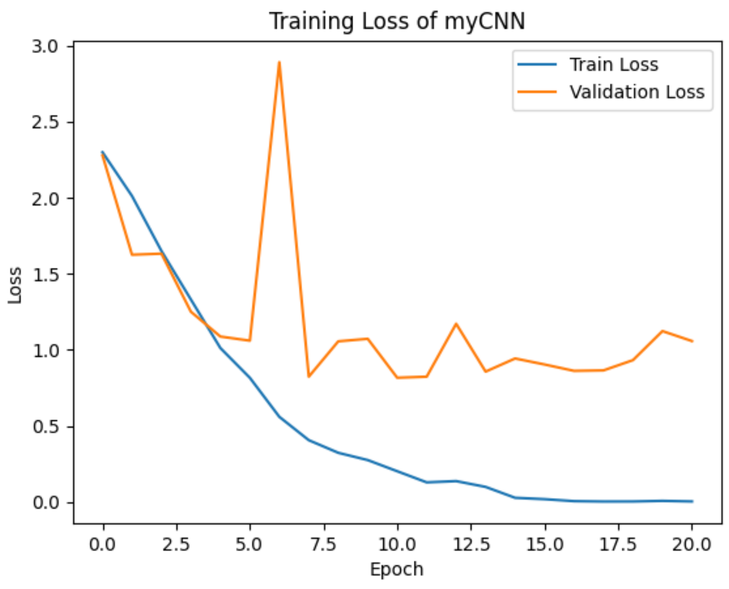


Figure 4: Observing model loss function per epoch

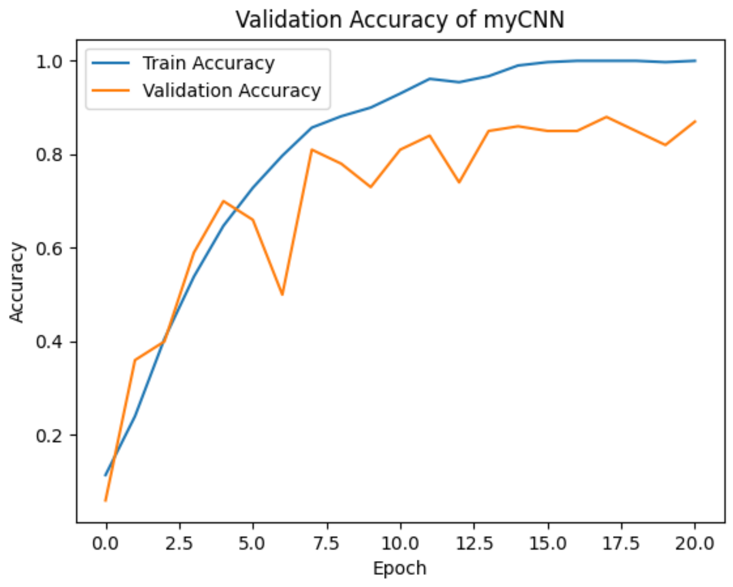
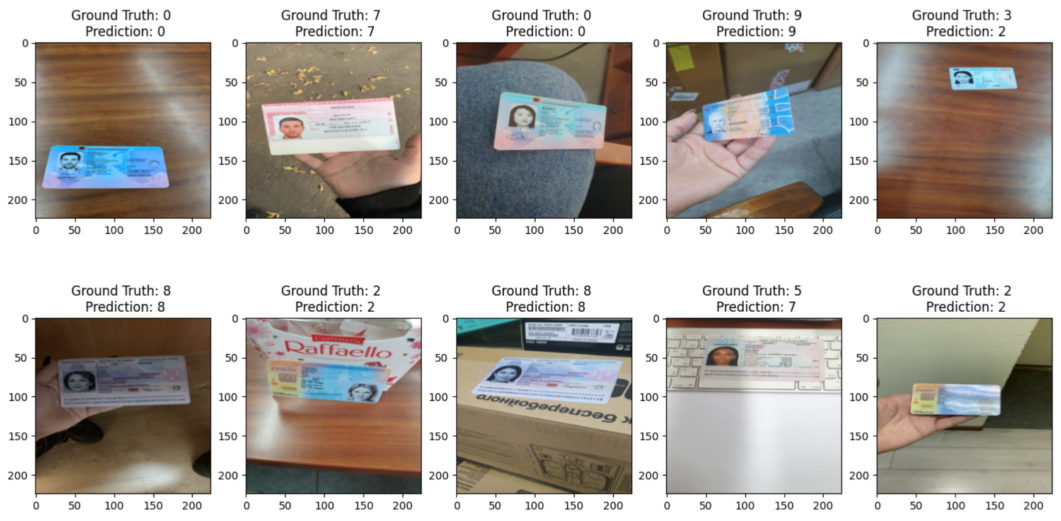


Figure 5: Observing model accuracy function per epoch

Furthermore, to provide a qualitative assessment of the model's performance, a selection of 10 images from the test dataset was randomly chosen. These images were accompanied by their ground truth labels (actual labels for the images) and model predictions, allowing for an intuitive understanding of the model's classification capabilities.



Ground Truth: 0 Prediction: 0
Ground Truth: 7 Prediction: 7
Ground Truth: 0 Prediction: 0
Ground Truth: 9 Prediction: 9
Ground Truth: 3 Prediction: 2
Ground Truth: 8 Prediction: 8
Ground Truth: 2 Prediction: 2
Ground Truth: 8 Prediction: 8
Ground Truth: 5 Prediction: 7
Ground Truth: 2 Prediction: 2

Figure 6: Visualizing predictions

A crucial aspect of evaluating the model's performance is the confusion matrix, which provides insights into the model's precision, recall, and F1 scores for each individual class. In the confusion matrix, each class is represented by a label ranging from 0 to 9, corresponding to specific document types. For instance, class 0 represents the ID Card of Albania, while class 6 represents the Passport of Latvia. The confusion matrix reveals the model's ability to correctly classify instances within each class, as well as any misclassifications or errors observed. For instance, for class 0 (ID Card of Albania), out of a total of 19 images, 15 were correctly classified, while one was misclassified as class 2 (ID Card of Spain), one as class 3 (ID Card of Estonia), and two as class 4 (ID Card of Finland). The corresponding precision, recall, and F1 score metrics provide a comprehensive understanding of the model's performance for each class. Notably, the overall precision, recall, and F1 score of the model were all observed to be 86%, reflecting a balanced performance across the different classes.

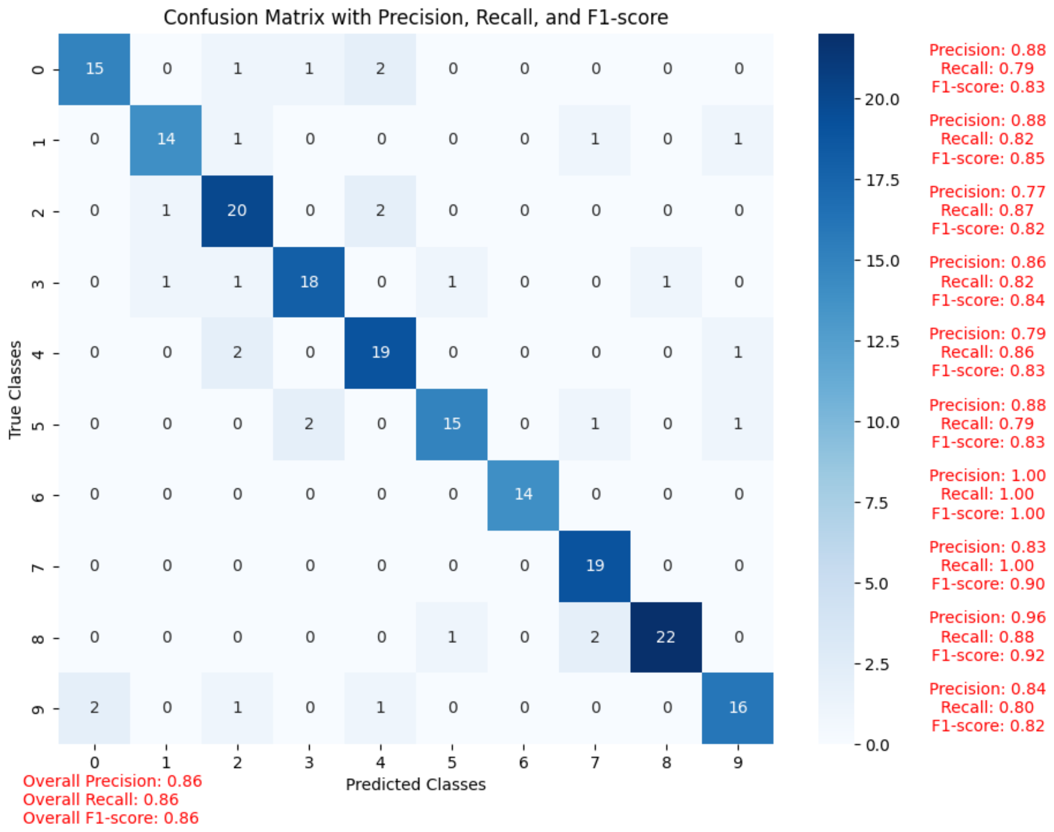


Figure 7: Confusion matrix with precision, recall, and f1-score

Final Notes

- The code is implemented in Python and is compatible with environments that support CUDA. To simplify the setup process, I've included a Dockerfile and a requirements.txt file containing all necessary libraries. Running the machine learning model is now straightforward. Simply execute the following command in your terminal:

docker run –name my_all_gpu_container –gpus all -t nvidia/cuda
Once executed, you'll be able to observe the results. For additional information regarding recommended hardware specifications, please refer to the "Introduction" section and review the concluding paragraph. In crafting this code, emphasis has been placed on clarity and readability. Descriptive variable names have been chosen to enhance comprehension, while comments preceding each section provide valuable insights into the functionality of the code.

- To further enhance the model's performance, additional data augmentation techniques can be employed, such as random cropping, color jittering, and random flipping or rotation. Optimization of hyperparameters and the addition of more layers to the neural network structure, including residual connections for deeper models, are also potential avenues for improvement. Additionally, considering the use of higher resolution images, given that the current images were cropped to 224x224, could yield better results. It's important to note, however, that there exists a trade-off between model complexity and performance metrics. While implementing these methods may potentially improve accuracy by up to 98%, it will likely lead to increased complexity of the neural network, longer training times, and larger model file sizes, particularly for deployment purposes. Therefore, a decision was made to develop a moderately simple model with exceptional performance, achieving a total accuracy of 86%, which aligns well with the project's objectives.

- To run the ResNet50 model, please follow these steps:

  - Uncomment the lines specified within triple quotation marks on lines 95 and 96.
  - Remove the custom model implementation starting from line 99 to line 130.
  - Delete the provided code block from lines 142 to 145.
  - Uncomment the line within triple quotation marks on line 148.

  Please note that the ResNet50 model is considerably deeper and more complex compared to the custom-built model. While it can achieve an accuracy of up to 98% with a significantly lower loss function (0.02), it also occupies significantly more space.