**HCMC UNIVERSITY OF TECHNOLOGY AND EDUCATION**

**FACULTY OF INFORMATION TECHNOLOGY**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

**Course: IT Project**

**Vietnam Currency Recognition using Deep Learning models**

**Lecturer Name :** Quách Đình Hoàng

**Course ID :** IT Project\_ Nhom 08FIE

**List of members:** Nguyễn Đức Trí - 22110082

Nguyễn Hải Triều - 22110081

Ho Chi Minh City, 12/2024

**Workplan**

|  |  |  |
| --- | --- | --- |
| **STUDENT NAME** | **STUDENT ID** | **TASK** |
| Nguyễn Đức Trí | 22110082 | **Define loss function and optimizer, Train network, make evaluation for network** |
| Nguyễn Hải Triều | 22110081 | **Prepare Data, Setup Dataset for Training Model, Define Model** |

**Acknowledgment**

We would like to express our deep gratitude to **Mr. Quach Dinh Hoang**, our project supervisor, for guiding us throughout the project. His valuable suggestions and insights were crucial in helping us complete the work and grasp the complex aspects of project development, as well as present it effectively. Without his support, many of these complexities would have been overlooked.

Projects are typically completed within ten weeks. However, due to much new knowledge as well as the time we do through each week is not optimal, the project will have many errors, which is inevitable. We are looking forward to receiving all the comments of our teachers to help our limited knowledge better.

Sincere thanks.

**Preface**

The purpose of this training, along with its content, is timely, and it has helped us gain confidence in introducing the project. We also believe that we've acquired some basic IT knowledge, and with more practice and experience, we will be able to thrive in today's competitive environment.

Writing this report is part of completing the course. I have done my best to organize all the topics we learned in the training program in a clear and structured way. Each topic is divided into individual chapters to make everything easier to understand. I used proper citations throughout the report. I hope the structure and content of this report will be helpful to all readers.

**Table of contents**

[**1.** **ABSTRACT** 1](#_Toc184934721)

[**2.** **INTRODUCTION** 2](#_Toc184934722)

[**3.** **METHODOLOGY** 3](#_Toc184934723)

[**3.1 REQUIREMENTS** 3](#_Toc184934724)

[**3.2. DATASET** 4](#_Toc184934725)

[**3.2.1 Data preperation** 4](#_Toc184934726)

[**3.2.2 Set up the dataset** 8](#_Toc184934727)

[**3.2.3 Data augmentation** 11](#_Toc184934728)

[**3.2.4 DataLoader** 12](#_Toc184934729)

[**3.3 MODEL DESIGN** 14](#_Toc184934730)

[**3.4 TRAINING AND VALIDATION PHASE** 20](#_Toc184934731)

[**3.4.1 Utility** 20](#_Toc184934732)

[**3.4.2 Training** 22](#_Toc184934733)

[**3.4.3 Training and Validation evaluation:** 28](#_Toc184934734)

[**4. FUTURE WORK** 34](#_Toc184934735)

[**5. REFERENCE** 35](#_Toc184934736)

# **ABSTRACT**

Given that money plays an essential role in everyday life and business transactions, currency recognition becomes crucial, particularly for those who are blind or have visual impairments. To address this issue, we propose a work to help visually challenged people identify different denominations of Vietnamese currency using deep learning techniques. By utilizing deep learning approaches, the system will enable these individuals to recognize banknotes more easily.

***Keywords:*** *-* Currency (Banknotes) Dectection, VGG, deep learning.

# **INTRODUCTION**

Object recognition is a crucial and highly demanded field within pattern recognition, with applications spanning a wide range of real-world scenarios. Objects can include anything from text in documents, vehicle license plates, human irises, sign language gestures, to facial features. Similarly, recognizing currency is just as vital as other forms of object recognition due to its practical significance in daily life.

Especially, the ability to accurately identify currency is vital for visually impaired individuals, facilitating independence in financial transactions and everyday activities. Traditional methods of currency recognition, such as tactile features or assistance from others, often fall short in providing the autonomy that visually impaired users require. Recent advancements in computer vision and deep learning present new opportunities to address these challenges effectively. This project seeks to harness these technologies to develop a system design for recognizing Vietnamese banknotes, enhancing the accessibility of currency recognition for visually impaired individuals.

# **METHODOLOGY**

Currency recognition has been a focus of research in recent years. There are various ways to identify currency based on the features used for classification and the deep learning model applied. Every currency has specific features that make it easy to classify. In the case of Vietnamese currency, certain key features are essential for accurate recognition. For instance, every Vietnamese banknote has a special value that printed in the surface, which corresponds to the label I will define below.

The proposed explanation is reasonably straight forward classifying Vietnamese currency notes using deep learning approaches. There are various image classifier models, each following its methodology. In this project, we chose VGG (stands for Visual Geometry Group), which is a standard deep Convolutional Neural Network (CNN) architecture with multiple layers. VGG is now still one of the most well-known image recognition architectures compared to other models. Developed as a deep neural network, the VGGNet also surpasses baselines on many tasks and datasets beyond [ImageNet](https://viso.ai/deep-learning/imagenet/).

## **3.1 REQUIREMENTS**

Here are some basic requirements needed for the implementation.

++ Python 3.7 or above

++Pytorch 2.1.2

++ Tensorboard 2.12.1

++ Open-cv 4.10.0.84

## **3.2. DATASET**

### **3.2.1 Data preperation**

There will be two files to generate data for train and test set. Then we will save them to VNCurrency/currency\_train folder with the train set and VNCurrency/currency\_test with the test set. The file structure is given below:

CV

├── data

│ └── VNCurrency

│ └── currency\_train

│ └── currency\_test

Prepare banknotes in denominations from 1,000 VND to 500,000 VND.

The data preparation will be through personal webcam by continuous image capture (Frame-by-Frame). Images will be captured and saved to folder with given label at speed of 30 frames per second (Webcam recording speed)

Some captured images used for training:

A person holding up a paper money

Description automatically generated **A close up of a paper money

Description automatically generated** A hand holding a paper money

Description automatically generated

**A person holding a paper money

Description automatically generated** A hand holding a paper money

Description automatically generated

There are 10 labels: 9 for Vietnamese currency denominations (1000 VND, 2000 VND, 5000 VND, 10000 VND, 20000 VND, 50000 VND, 100000 VND, 200000 VND, 500000 VND) and 1 label (0) for images with no money present. Here is our pseudo code for appending images into **currency\_train** and **currency\_test**:

For currency\_train, we create a python file name *maketraindata.py* with 2000 images for each label.

A screenshot of a computer program

Description automatically generated

**Figure 1: Create 50000VND training images**

Likewise, with the testing data, we create *maketestdata.py* with 400 images for each label.

A screenshot of a computer program

Description automatically generated

**Figure 2: Create 50000VND testing images**

After having done with creating a data, the folder will have this structure:

CV

├── data

│ └── VNCurrency

│ └── currency\_train

│ └── 0000

│ └── .jpg (images with label 0)

│ └── 1000

│ └── .jpg (images with label 1000 VND)

│ └── 2000

│ └──.jpg (images with label 2000 VND)

│ ……

│ └── 500000

│ └──.jpg (images with label 500000 VND)

│ └── currency\_test

│ └── 0000

│ └── .jpg (images with label 0)

│ └── 1000

│ └── .jpg (images with label 1000 VND)

│ └── 2000

│ └──.jpg (images with label 2000 VND)

│ ……

│ └── 500000

│ └──.jpg (images with label 500000 VND)

### **3.2.2 Set up the dataset**

In reality, there will be no data that has already been formatted, so understanding and knowing how to customize dataset is very important and it is very commonly used. We will create Dataset into a class like the model above. This class inherits **torch.utils.data.Dataset** and has 3 required functions:

A computer screen shot of a program code

Description automatically generated\*\* **init**: Is the initialization function, receives parameters and initializes the corresponding parameters. In the Vietnamese currency problem, we have defined some parameters:

**Figure 3: \_\_init\_\_ method**

Here's an explanation of the function of each variable in the \_\_init\_\_ method:

**+ self.root = root**

* + Purpose: Stores the root directory of the dataset.
  + Usage: This variable is used to build the paths for either the training data (currency\_train) or test data (currency\_test), depending on whether the dataset is being used for training or testing.

**+ self.transform = transform**

* + Purpose: Stores any transformations to be applied to the dataset.
  + Usage: If provided, transformations such as resizing, normalization, or data augmentation will be applied to each image when it is loaded (inside the \_\_getitem\_\_ method).

**+ self.currencyList = []**

* + Purpose: Initializes an empty list to store the categories of currency labels.
  + Usage: This list is populated with the names of subdirectories (e.g., different currencies) found in the dataset directory. Each subdirectory represents a category of currency (label).

**+** **self.labels = []**

* + Purpose: Initializes an empty list to store the label for each image.
  + Usage: Each label corresponds to the index of a currency in self.currencyList. Labels are assigned to images based on the directory they belong to.

**+ self.image\_paths = []**

* + Purpose: Initializes an empty list to store the file paths of all images in the dataset.
  + Usage: During dataset creation, all valid image file paths (with extensions like .jpg, .png, etc.) are stored in this list. This ensures quick access to each image during training or testing.

**+** **self.transform = transform**

* + This is a redundant assignment that duplicates self.transform = transform from earlier. It serves no additional purpose and could be safely removed from the code.

\*\* **len**: The function returns the length of the data.

A black background with white text

Description automatically generated

**Figure 4: \_\_len\_\_ method**

A screen shot of a computer code

Description automatically generated\*\* **getitem**: receives the index, this index is in the length of the data. This function aims to read data, process data, labels and return standard data to put into the model. Augmentation methods are implemented here. Depending on the data format, you can create the input data, in the example below, we have preprocessed and obtained data in the form of a dict, each example is a pair of image links and image labels. Image augmentation can use the **torchvision.transforms** library, so if you read the input data using Numpydarray, you can use the albumentations library.

**Figure 5: \_\_getitem\_\_ method**

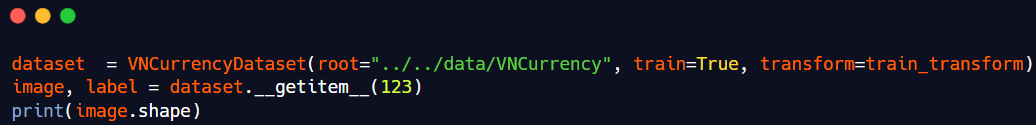
### **A screenshot of a computer program Description automatically generated3.2.3 Data augmentation**

**Figure 6: Data Augmentation**

The train\_transform pipeline is designed to apply various transformations to augment the training dataset. This helps improve the model's generalization by creating diverse versions of the training images.

* A.Resize(width=train\_size, height=train\_size)  
  Resizes the input image to the specified size (train\_size x train\_size, e.g., 224x224). Ensures consistency in image dimensions across the dataset.
* A.HorizontalFlip(p=0.5)  
  Randomly flips the image horizontally with a probability of 50%. This simulates variations in orientation (e.g., mirrored objects).
* A.RandomBrightnessContrast(p=0.2)  
  Adjusts the brightness and contrast of the image randomly with a probability of 20%. This simulates different lighting conditions.
* A.Blur()  
  Applies a mild Gaussian blur to the image. This can simulate slight camera out-of-focus effects.
* A.Sharpen()  
  Enhances the sharpness of the image, balancing smooth and sharp edges.
* A.RGBShift()  
  Shifts the RGB color channels randomly. This simulates slight variations in color caused by different lighting or camera settings.
* A.Cutout(num\_holes=5,max\_h\_size=25, max\_w\_size=25, fill\_value=0)  
  Randomly masks out square regions of the image (up to 5 holes). Each hole can have a maximum height and width of 25 pixels. This helps the model focus on global features instead of overfitting to specific local details.
* A.Normalize(mean=(0.485, 0.456, 0.406), std=(0.229, 0.224, 0.225), max\_pixel\_value=55.0)  
  Normalizes the image using the mean and standard deviation values of the ImageNet dataset. This is important for models pretrained on ImageNet, as it ensures consistent input distribution.
* ToTensorV2()  
  Converts the image to a PyTorch tensor (required for training).

### **3.2.4 DataLoader**

The whole process above is about creating the Dataset object, in order to generate each example, we can:

**Figure 7: Showing data in each example**

A screenshot of a video game

Description automatically generatedAnd in order to create data into a batch, we can use DataLoader():

**Figure 8: Showing data in a batch**

* dataset:

The dataset object you pass here should be an instance of a custom or predefined PyTorch dataset

* batch\_size=8:

The number of samples (images + labels) to be loaded in one batch.

* shuffle=True:

Randomly shuffles the data before each epoch. This helps improve generalization by ensuring that the model does not memorize the data sequence.

* drop\_last=True:

If True, drops the last incomplete batch if the dataset size is not divisible by the batch\_size.

* num\_workers=4:

Specifies the number of subprocesses to use for data loading. This helps increase the number of workers that can speed up data loading by performing data transformations and reading from disk in parallel.

The result will look like this:

A screen shot of a computer

Description automatically generated

**Figure 9: Showing data in a batch**

## **3.3 MODEL DESIGN**

We can create a model as a class by inheriting nn.Module, which allows you to create a deep learning model as a class. This class has 2 required functions, init and forward:

\*\*init: an initialization function that receives variables and parameters to help you initialize variables and functions of the initialized object. Because this Class inherits nn.Module, when initializing a new object of the Class, the inherited class must be initialized, so in the init function there must always be super().\_\_init\_\_().

**A screen shot of a computer program

Description automatically generatedFigure 10: \_\_init\_\_ method for model design**

\*\*forward: a function that receives initial input data. The data will go through each layer of the model in turn and return the model output.

A computer screen shot of a code

Description automatically generated

**Figure 11: forward method for model design**

A black screen with white text

Description automatically generatedThe result when using torch.summarylibrary to print out the configuration:

**Figure 12: VGG Model**

Our model implementation is based on the *VGG paper*. We choose configuration C as the model for implementation on Vietnamese currency recognition problem. The total parameter of our model reached approximately 130 million parameters. The depth of configuration C is 16 weight layers (16 conv layers + 3 fully connected layers). The input image is processed through a series of convolutional (conv.) layers, utilizing filters with a very small receptive field of **3 × 3**. This filter size is the smallest capable of capturing the spatial concepts of left/right, up/down, and center. In configuration C, **1 × 1 convolution filters** are also employed. These filters act as linear transformations of the input channels, followed by non-linearity.

Key characteristics of the convolutional layers:

* Stride: Fixed at 1 pixel for all convolution operations.
* Padding: Spatial padding of 1 pixel is applied to the input of the **3 × 3 convolution layers**, ensuring the spatial resolution is preserved after each convolution.

Spatial pooling is implemented using **five max-pooling layers**, applied after selected convolutional layers. Not all convolutional layers are followed by max-pooling. Max-pooling is performed using a **2 × 2 pixel window** with a stride of 2, reducing the spatial dimensions while retaining important features.

After the stack of convolutional layers, the network includes three **fully connected (FC)** layers:

1. The first two fully connected layers contain **4096 channels** each.
2. The third fully connected layer outputs **10 channels**, corresponding to the number of classes in the recognition task.

The final layer is a **softmax layer**, which outputs class probabilities. The configuration of the fully connected layers remains consistent across all network architectures. In addition, all hidden layers are equipped with the rectification (ReLU) non-linearity (Simonyan & Zisserman, 2014, pp. 2–3).

A diagram of a block diagram

Description automatically generatedHere is the visualization of VGG model:

**Figure 13: VGG Model Visualization**

## **3.4 TRAINING AND VALIDATION PHASE**

During the training process, the input to our model is a fixed size 224 x 224 RGB image. The only preprocessing we do is augmenting the data that affects the value of each pixel. To obtain the fixed-size 224 x 224 input image, we resize the training and testing images. To futher augment the training set, we use like what we did in the custom dataset step. However, in the validation set, we only do rescalling and normalization process.

### **3.4.1 Utility**

A computer screen shot of a program code

Description automatically generatedBefore we go to the training phase, we have written down a utils.py file. This file is a place to save things which do not relate to the mainstream of the training file. It is just a utility module for parsing command-line arguments. Its primary role is to provide a function, get\_args(), which uses the Python *argparse* library to define, parse, and return a set of command-line arguments. Here are some details about utils.py:

**Figure 14: utils.py**

Use case: This script likely serves as part of a deep learning training pipeline. It provides configurable options for:

* Specifying dataset paths.
* Setting training hyperparameters (e.g., batch size, epochs, learning rate).
* Choosing between optimizers.
* Resuming from checkpoints.
* Managing outputs (logs, trained models).

The utility of this script is to make the training process flexible and user-friendly, avoiding hard-coded configurations in the main training script. Users can modify training parameters directly from the command line without editing the code.

A black and white text

Description automatically generatedFor example:

**Figure 15: Example of parse use case**

* Loads dataset from ./data/VNCurrency.
* Sets batch size to 16.
* Trains for 10 epochs.
* Uses a learning rate of 0.0001 with the SGD optimizer.

### **3.4.2 Training**

**A computer screen shot of text

Description automatically generated**At first, we need to loader the data from the custom dataset process we have done above.

**Figure 16: Loading dataset**

The variable *device* means that we will use our computer’s GPU for speeding up our training process. There are two parameters of DataLoader() function: *training\_params* and *valid\_params* in which *batch\_size, num\_workers, drop\_last* and *shuffle* parameters are implemented. Specifically, I have set the default workers are all the number of workers available in my computer, in this case are 4. So, when the *num\_workers* is set to 4, at most 4 workers simultaneously putting data into RAM. The batch size I fit to the model is 8, meaning that 8 images will be loaded into the model. However, the reason why the variable *drop\_last* set to True in training process is because if the dataset is not divisble by the batch\_size, it leads to an incomplete last batch. If the last batch is smaller than the other batch, the gradient estimate become noisier and less stable, this can be proven by the formula:

where:

* : the parameter update.
* : the learning rate.
* N: the batch size.
* ∇L(xi​,θ): the gradient of the loss for the i-th sample.

Moreover, the *shuffle* variablein training is set to True which ensures the model sees a randomized order of data romoting better generalization, faster convergence, and robustness to overfitting to specific patterns in the data order. On the one hand, the *drop\_last* variable of validation is set to False, and *shuffle* is set to False indicating there is no need to shuffle data.

A computer screen shot of a program

Description automatically generated **Figure 17: Training setup code**

Next, the VGG16\_ConfigC model is being initialized with the number of classes equal to the length of train\_set.currencyList (in this case 10 labels). Then the model will use GPU for training. CrossEntropyLoss() is chosen as the loss function, commonly used for recognition tasks. Both *optimizer* and *scheduler* are initialized as None. If the optimizer is set to "adam", it will be selected with learning rate 0.001. Else if the optimizer is set to "sgd" (which means Stochastic Gradient Descent), it will be selected with learning rate 0.01 and that learning rate scheduler (MultiStepLR) is set up to reduce the learning rate by a factor of gamma=0.1 at specified milestones.

We also determine the checkpoint loading. If a previous checkpoint exists, we will:

* Load the model's state dictionary (state\_dict) from the checkpoint.
* Load the optimizer's state dictionary.
* Set the starting epoch to one greater than the checkpointed epoch.
* Retrieve the best accuracy so far from the checkpoint.

If no checkpoint is provided, training starts from epoch 0, and the best accuracy is initialized to 0.

In addition, TensorBoard is a powerful visualization toolkit provided by TensorFlow that helps you monitor and debug machine learning experiments. Thus, we utilize TensorBoard in order to visualize the loss, the accuracy of training and validation process, and the confusion matrix. If a directory for TensorBoard logs exists, it is deleted and recreated to ensure a fresh start. Similarly, a directory for saving trained models is created at the currently standing file/folder if it doesn’t already exist. A TensorBoard writer is initialized to log training metrics during training.

A computer screen shot of a program code

Description automatically generated

**Figure 18: Training code**

We define the number of epochs as 6, (this number can be modified right before the program has started running) which represents the total number of times the entire training dataset is passed through the model. Inside a for loop, the model will first switch to model.train(), enabling features like dropout and batch normalization. When entering the second for loop of training process, we have cleared the gradients from the previous step. The reason why we do that is in Pytorch, gradients are accumulated by default during backpropagation. This means that every time you call loss.backward(), the gradients calculated for each parameter are added to the gradients from the previous step. If we don’t clear the gradients using optimizer.zero\_grad() at the beginning of each training step, the accumulated gradients will affect the parameter updates, leading to incorrect results. After solving the problem of clearing gradient, images will be fit into the model for making predictions. The loss is then calculated by comparing the predictions to the ground truth labels, its gradient will be calculated afterward. The statement optimizer.step() updates the model's parameters based on the gradient. The loss for the current batch (loss.item()) is logged to the progress bar and added to the losses list. The data taken into TensorBoard after training a model will look like this:

A graph showing the value of a stock market

Description automatically generated

**Figure 19: TensorBoard illustration**

The y-axis represents the loss value of training process, whereas the x-axis represents the number of iterations of all epochs (each loss result will be recorded on each iteration). That is all about the training step, we will proceed to the next part is validation.

**A computer screen shot of a program code

Description automatically generated Figure 20: Validation code**

The statement model.eval() switches the model to evaluation mode. There is no need to have gradient computation, which is unnecessary during validation and helps save memory and computation. Same as training step, for each batch in the validation dataset, the model makes predictions based on the input images. The loss is also counted each iteration. Next, the variable *maxVal\_ofIdx* finds the index of the highest probability (the predicted class) and it will be saved in *all\_prediction* variable.

However, the validation loss is written to TensorBoard once per epoch, which is different with the training step, because we want to to it reflects the overall performance of the model. Logging per iteration would introduce unnecessary noise, clutter the visualization, and increase resource usage without adding meaningful insight. We also print the accuracy on each epoch and then show the confusion matrix for 10 labels (see in Figure [21] & [22]).

### **3.4.3 Training and Validation evaluation:**

WeA screenshot of a computer program

Description automatically generated use optimizer Adam Gradient in the training. Figure [21] shows the average loss value after training 1 epoch. At first, we set the value of epochs as 6, but in the progress of training 3 epochs, we have witnessed that the Cross Entropy loss values cannot go down to even 2. That indicates a very poor performance in our training process.

**Figure 21: Average results of training 3 epochs**

A graph with red lines

Description automatically generated Figure [22] displays the detail of every loss value shown in training. The loss values, after having a good drop at some first 900 iterations, increase back and remain unchanged at approximately 2.36 until the end. Figure [21] is a bit different (nearly 2.5 loss value), in which it shows the average loss value of each epoch instead of each iteration.

**Figure 22: Result of train/loss 3 epochs in TensorBoard**

A graph with red lines

Description automatically generated

**Figure 23: Result of validation/loss 3 epochs in TensorBoard**

Figure [24] shows the validation accuracy, which is 0.1, indicating a very low performance, the model only predict 10% accuracy over 100%. Therefore, some solutions must be carried out.

A white paper with black lines

Description automatically generated

**Figure 24: Result of validation/accuracy 3 epochs in TensorBoard**

Similarly, the loss values are very high when we choose the setting optimizer Stochastic Gradient Descent (SGD) (see in Figure [25], [26]). We have mentioned earlier that the learning rate will reduce a factor of gamma=0.1 in the specific milestones (in this case is 3rd, 6th, 9th epoch per 10 epochs overall). Training begins with the initial learning rate 0.01, which is reduced by a factor of γ=0.1 at the 3rd epoch (to 0.001), again at the 6th epoch (to 0.0001), and finally at the 9th epoch (to 0.00001) allowing for finer adjustment as training progresses. However, the results didn’t improve when switching the setting.

A graph with red lines

Description automatically generated

**Figure 25: Result of train/loss 10 epochs in TensorBoard**

A graph of a graph with a line

Description automatically generated with medium confidence

**Figure 26: Result of validation/loss & accuracy 10 epochs in TensorBoard**

At first, we thought that the model did not perform well because we add too much data augmentation which might distort features to the point where the model struggles to learn meaningful patterns. Thus, we tried to remove some parameters like HorizontalFlip(), RandomBrightnessConstrast, Blur, Sharpen, RGBShift and Cutout, which is the same transformer as the validation step. Noticing that all these improvements will be done using AdamGrad optimizer. However, the result still turned out to be not good, still around 2.3 (see in Figure [27]&[28]).

A graph showing a line graph

Description automatically generated with medium confidence

Figure 27: Result of train/loss 3 epochs after removing some data augmentation

A graph of loss and loss

Description automatically generated

**Figure 28: Result of validation/loss & accuracy 3 epochs after removing some data augmentation**

Therefore, adding data augmentation does not affect the loss of the model much.  We considered the VGG model to be a problem in our training. We have noticed that we use the criterion CrossEntropyLoss which expects raw logits from the model outputs, then internally applies the softmax function and computes the negative log-likelihood in a numerically stable manner. We additionally apply the Softmax layer on the final Fully Connected layer of the VGG model, it takes the raw logits of the model and converts these logits to probabilities. Those probabilities will then take to the CrossEntropyLoss causing the calculation to be redundant.

To simplify it, we have a mathematical proof: consider we have the logits z = [z1, z2, z3, …, z10] is a vector of logits. When applying it to softmax layer, pi = , which produces 10 values corresponding to 10 labels. Those pi values will then pass to the CrossEntropy Loss = - log\_softmax(pi), producing a redundancy in calculation.

A graph with a red line

Description automatically generated

**Figure 29: Result of train/loss 3 epochs after removing the softmax layer**

Figure [29] represents the results after removing softmax layer of the VGG model. The loss curve becomes smoother because the softmax operation is part of the loss calculation in frameworks like PyTorch meaning the cross-entropy loss directly handles logits without requiring explicit softmax application. On the other hand, the results of training are still not enhanced.

A blue and yellow square with black dots

Description automatically generatedFigure 30: Confusion matrix

We also try to print the confusion matrix to have a closer look. The y-axis represents the true labels and the x-axis represents the predicted labels (range from 0 to 9). From the Figure [30] we can see that only label 4 (which is 10000 VND) got 100 % predicted accurately, and 9 labels left got 0%, indicating severe overfitting or class imbalance in the model's training process.

Thus, the above improvements drive our attention to two reasons that make our model retrieving the bad result. The first thing is the VGG model is not suitable for this currency dataset and the second thing is our dataset have a problem.

# **4. FUTURE WORK**

In this project, we explored the task of Vietnamese currency recognition using deep learning models. Our primary goal was to develop a system capable of accurately identifying and distinguishing between different denominations of Vietnamese currency. By designing and training a robust deep learning model, we laid the foundation for a solution to assist in this recognition task. While we achieved some results, these works are still ongoing, and there are several directions for improvement and further development.

Moving forward, one of our primary goals is to enhance the accuracy of the model. This involves experimenting with more advanced architectures, refining hyperparameters, and incorporating larger, more diverse datasets to reduce overfitting and improve generalization. Additionally, techniques like transfer learning and fine-tuning with state-of-the-art pre-trained models could further boost performance.

Another critical area for future development is the integration of counterfeit banknote detection. By expanding the dataset to include counterfeit samples and training the model to distinguish between genuine and fake notes, we aim to address a real-world concern in currency handling. This could significantly enhance the practicality of our system for broader societal use.

Finally, we plan to develop a fully functional application based on our model to assist visually challenged individuals. By integrating the model into an accessible mobile or web-based application, we aim to provide real-time assistance for tasks such as currency recognition, denomination differentiation, and counterfeit detection. This would ensure that our work not only remains theoretical but also delivers meaningful social impact.

**5. REFERENCE**

[1] Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for Large-Scale image recognition. *Computer Vision and Pattern Recognition*. http://export.arxiv.org/pdf/1409.1556.

[2] Thắng, N. C. (2020, May 17). *Nhận dạng tiền Việt Nam với Transfer Learning (VGG16 CNN Classify)*. Mì AI. <https://miai.vn/2020/04/21/nhan-dang-tien-viet-nam-voi-transfer-learning-vgg16-cnn-classify/>.

[3] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, *60*(6), 84–90. <https://doi.org/10.1145/3065386>.

[4] Al-Edreesi, E., & Al-Gaphari, G. (2024, June 18). *Real-time Yemeni Currency Detection*. arXiv.org. <https://arxiv.org/abs/2406.13034>.

[5] *Get started with Tensorboard.* (n.d). TensorFlow.

<https://www.tensorflow.org/tensorboard/get_started>.

[6] Pachón, C. G., Ballesteros, D. M., & Renza, D. (2021). Fake Banknote Recognition Using Deep Learning. *Applied Sciences*, *11*(3), 1281.

<https://doi.org/10.3390/app11031281>.