ECE479 - LAB3 FINAL REPORT

DNN Acceleration with Arm CV/ML Library on Raspberry Pi

(Mini-Lab at the end of report)

Berkay Sekeroglu (**berkays2**) / Hakan Ak (**hakanak2**)

05/02/2023

Selected track: Track II, DNN Accelerator and DNN Acceleration Library
Platform: CPU (Raspberry Pi) and Coral TPU (additionally, for comparison)

1 Motivation, problem and the challenges

Deep neural networks (DNNs) have become increasingly popular due to their exceptional performance in complex tasks and relationships. As challenging problems demand the understanding of complex relationships, DNNs play an increasingly important role in describing such connections. The ability to identify these complex relationships makes DNNs more computationally intensive. Consequently, some DNNs may take even several seconds to process a single image. Although powerful GPUs or TPUs can reduce latency, their high cost and impracticality in many situations remain a concern.

Moreover, as edge computing gains popularity and utility, there is a desire to run DNNs on these computationally-limited devices. In such cases, real-world applications require real-time inference speeds. Therefore, accelerating DNNs with minimal trade-offs is crucial since achieving high accuracy on powerful devices is no longer the primary concern.

To address such a challenge, we explored available libraries. One such library was the ComputeLibrary from Arm [1]. An existing solution, also explored in the labs in this course, was the Coral TPU USB Accelerator. We did compare the existing solution to another possible solution with the modifications of our own.

ComputeLibrary [1], provides "Arm NN SDK" [2] which utilizes the Compute Library to target some CPUs and GPUs, as efficiently as possible. ArmNN [2] can be utilized as an inference engine for CPUs and GPUs.

One main challenge we faced was, installing PyARMNN Library for Python. Down below we provide an installation guide for the library.

```
git clone https://github.com/ARM-software/armnn.git armnn #Get the
    library from GitHub
cd armnn/build-tool/scripts #Enter into the Arm NN build-tool directory
    where the Dockerfile and associated scripts are located.
sudo ./install-packages.sh

./setup-armnn.sh --target-arch=aarch64 --all
./build-armnn.sh --target-arch=aarch64 --all --neon-backend --cl-backend

./setup-armnn.sh --help #You can also check for different parameters.
./build-armnn.sh --help #You can also check for different parameters.
```

Code 1: Installing ARMNN[3]

```
cd armnn
export BASEDIR=‘pwd‘

# Python Environment
python -m venv env #Change the environment name env as you wish
source env/bin/activate #Use this line later to activate the environment.
pip install wheel

# Install SWIG
sudo apt-get install libpcre3 libpcre3-dev
cd $BASEDIR
mkdir swig
cd swig
wget http://prdownloads.sourceforge.net/swig/swig-4.0.2.tar.gz
chmod 777 swig-4.0.2.tar.gz
tar -xzvf swig-4.0.2.tar.gz
cd swig-4.0.2/
./configure --prefix=/path/to/armnn/swigtool/
sudo make
sudo make install
sudo nano /etc/profile

# Add the following lines to /etc/profile
#   export SWIG_PATH=/path/to/armnn/swigtool/bin
#   export PATH=$SWIG_PATH:$PATH
source /etc/profile
```

Code 2: Setting Python Environment and SWIG[4]

```
cd $BASEDIR/python/pyarmnn
export SWIG_EXECUTABLE=$BASEDIR/swigtool/bin/swig
```

```
export ARMNN_INCLUDE=$BASEDIR/include:$BASEDIR/profiling/common/include
export ARMNN_LIB=$BASEDIR/build-tool/scripts/build/armnn/aarch64_build

sudo apt-get install python3.6-dev build-essential checkinstall
    libreadline-gplv2-dev libncursesw5-dev libssl-dev libsqlite3-dev
    tk-dev libgdbm-dev libc6-dev libbz2-dev

python setup.py clean --all
python swig_generate.py -v
python setup.py build_ext --inplace

python setup.py sdist
# This will create tar.gz file under ./dist

python setup.py bdist_wheel
# This will create .whl file under ./dist, check for the name of the
    folder for the line below.

pip install dist/pyarmnn-32.0.0-cp39-cp39-linux_aarch64.whl

# Check Installation With
pip show pyarmnn

# Or with
python -c "import pyarmnn as ann;print(ann.GetVersion())"
```

Code 3: Installing PyARMNN[5]

2 Design in detail

First, we required a valid, trained model to test inference on the Raspberry Pi. Additionally, we aimed to test various models to gain a better understanding of the speedup across different models with varying frame rates per second (FPS) and accuracies.

Considering, models of different sizes and different inference speeds. We did choose three architectures: MobileNetV2 [6], Xception [7], and NasNetLarge [8] (highest FPS to lowerst FPS). MobileNetV2 is already working in real-time, and the other two are below 5 FPS, normally.

These models are already trained on ImageNet and pre-trained weights are available. However, ImageNet was too large for our practical purposes. Therefore we wanted to go with a smaller dataset which we can compare the accuracy on a smaller testing data. Therefore, we decided to utilize CIFAR-100 [9]. To achieve that, we used transfer learning to utilize the weights from pre-trained models (on ImageNet) to get good-enough accuracies on ImageNet.

First of all, we imported these libraries and models. We grabbed the weights for these models that are already pre-trained on ImageNet. For better generality, we added a data augmentation layer, which appliess random flips,

crops to the input and preprocesses the input. Then we feed this preprocessed input to our base model (which is one of the three architectures mentioned above). Since the base model has 1000 classes because of ImageNet, we added a dense layer with dropout (with 256 nodes) and an output layer of 100 classes.

As the next step, we trained the model while freezing the base model, for a couple of epochs. The purpose in this step was to teach the model the relations between those 1000 classes from ImageNet and 100 classes from CIFAR-100. During these few epochs, no training is done on our base model.

After transer learning, we did a fine-tuning step while unfreezing the base model. In this step, all of the model is available for training and all weights can be adjusted. This is an important step to get better accuracies and better generalizations. This step also took a few epochs (10-15), but it took more time because we are training the whole model in this step. Figure 1 describes the process of transfer learning and fine tuning.

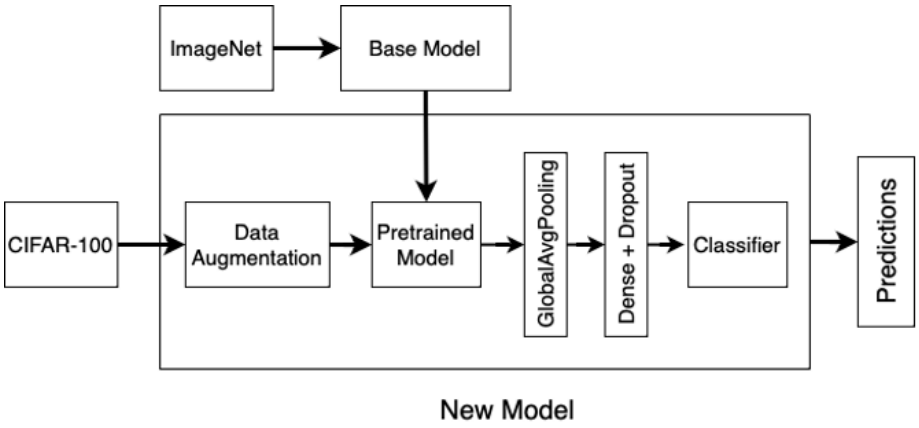

Figure 1: Transfer learning and fine-tuning

After having a good model in our hand, we applied post-training quantizaton on the Tensorflow model with UINT8 quantizaton. Then converting and saving our model as a TFLite model, now we have a valid model to test the inference time on Raspberry Pi CPU with and without Arm NN engine.

For the testing methodology, we carried out inference for 3 different parsers. TFLite, Coral and ARMNN.

ARM CV/ML Library can be used with different type of approaches. We decided to use the method where we can use the tflite file we created directly. In this approach, the layers in our models that ARMNN doesn't support does not allow us to do inference with PyARMNN. In that case, another methods, which sends the unsupported layers to the TFLite delegate can be used.

Here are the usage of ARMNN parser to carry out inference. We used the function below for our real time inference script.

```
parser = ann.ITfLiteParser()
network = parser.CreateNetworkFromBinaryFile(tflite_filename)


graph_id = 0
input_names = parser.GetSubgraphInputTensorNames(graph_id)
input_binding_info = parser.GetNetworkInputBindingInfo(graph_id,
    input_names[0])
input_tensor_id = input_binding_info[0]
input_tensor_info = input_binding_info[1]
print('tensor id: ' + str(input_tensor_id))
print('tensor info: ' + str(input_tensor_info))
# Create a runtime object that will perform inference.
options = ann.CreationOptions()
runtime = ann.IRuntime(options)

# Backend choices earlier in the list have higher preference.
preferredBackends = [ann.BackendId('CpuAcc'), ann.BackendId('CpuRef')]
opt_network, messages = ann.Optimize(network, preferredBackends,
    runtime.GetDeviceSpec(), ann.OptimizerOptions())
# Load the optimized network into the runtime.
net_id, _ = runtime.LoadNetwork(opt_network)
print(f"Loaded network, id={net_id}")
# Create an inputTensor for inference.
output_names = parser.GetSubgraphOutputTensorNames(graph_id)
output_binding_info = parser.GetNetworkOutputBindingInfo(0,
    output_names[0])

def pyarmnn_inference(image):
    input_tensors = ann.make_input_tensors([input_binding_info], [image])

    # Get output binding information for an output layer by using the
        layer name.
    #output_names = parser.GetSubgraphOutputTensorNames(graph_id)
    #output_binding_info = parser.GetNetworkOutputBindingInfo(0,
        output_names[0])
    output_tensors = ann.make_output_tensors([output_binding_info])
    runtime.EnqueueWorkload(0, input_tensors, output_tensors)
    results = ann.workload_tensors_to_ndarray(output_tensors)
    return results[0]
```

Code 4: Inference with PyARMNN[4]

Coding for inference with TFLite and Coral will not be included here since it was covered in earlier labs.

We created a script for evaluating FPS and Accuracy for all 3 methods.

To calculate FPS of the model, we have inferenced 1000 test images from the CIFAR100 dataset for each of the methods, with a batch size of 1. The

reason we used 1 batch size is, in real time, we will not have all of the frames, thus we won't be able to move them to the memory all at once. We, then calculated the frame per second by averaging the whole time for 1000 images.

For the accuracy, the calculation was a standard accuracy calculation with ration of the number of true labels to the number of total labels.

We have repeated the same methodology for 3 different image recognition models, MobileNetV2, NasNetLarge and Xception.

We couldn't find many (good) resources on this topic, while comparing different models with a another optimization method (Coral TPU). We also explored transfer learning and inference optimization with libraries together in our project. Also we provide a real-time demonstration on how a library can make a "non-realtime" model into a "realtime" model.

3 Results and demonstration

Down below, we provide FPS and Accuracy results for 3 different inference methods and 3 different models, tested on CIFAR-100 testing split. It is important to note that there is an acceptable amount of decrease in accuracy with this much increase in FPS, after utilizing Arm NN.

The script we ran the benchmarks on can be accessible as 'benchmark.py'

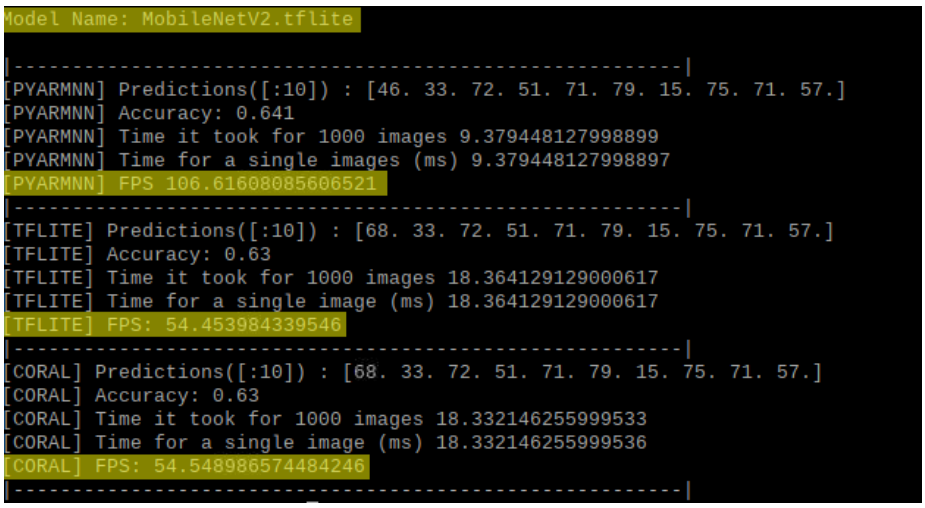

Figure 2: Benchmarking Results for MobileNetV2

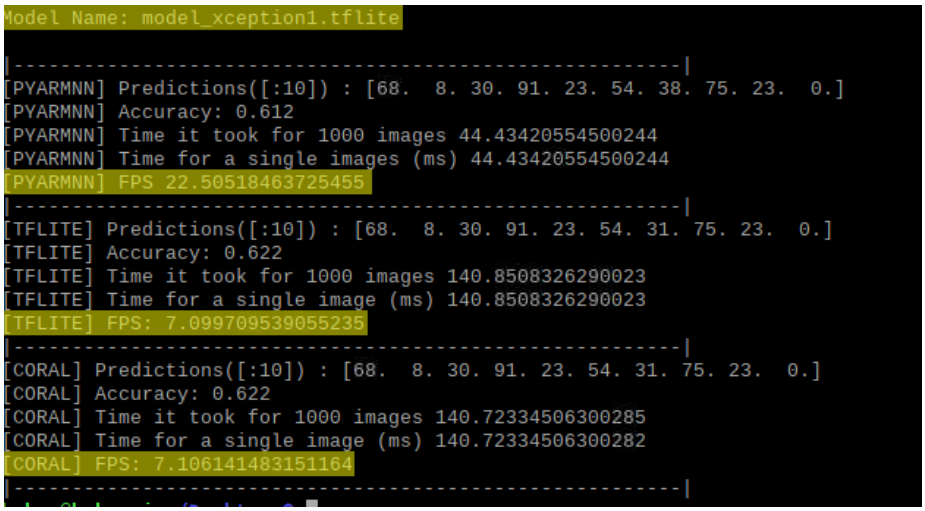

Figure 3: Benchmarking results for Xception

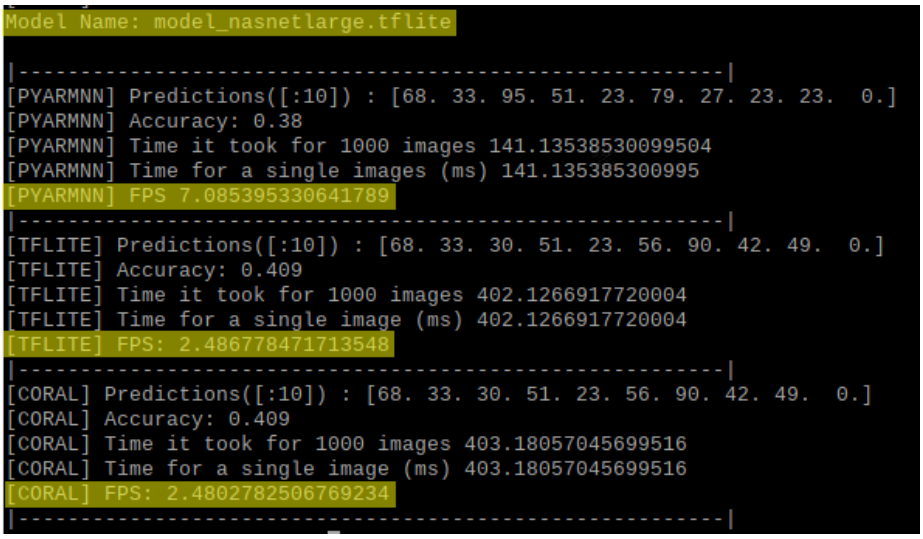

Figure 4: Benchmarking results for NasNetLarge

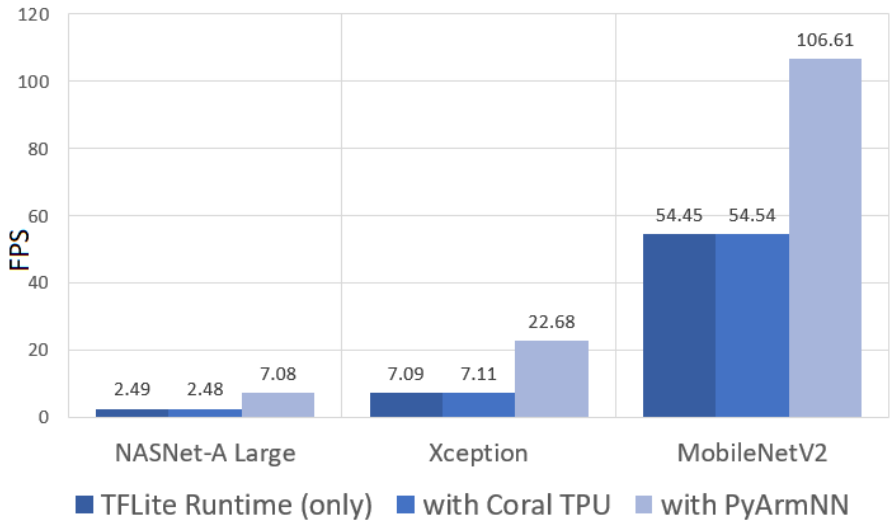

Figure 5: Comparison of FPS of all cases

As we can see in the above plot and the console outputs, we have obtained x1.95, x3.19, x2.84 speedup in MobileNetV2, Xception and NasNet-A Large models respectively.

We believe that the reason we have got better results in Xception and NasNet was becasue MobileNetV2 was a already lighter model that is made to work in IoT devices and it was already on real-time.

We have also created a script to make the models work in real time. It can be found with the name 'classify.py'

Here is a working demonstration of the model. It provides the predicted classes within the video box, Also prints out the FPS at the same time in the console.

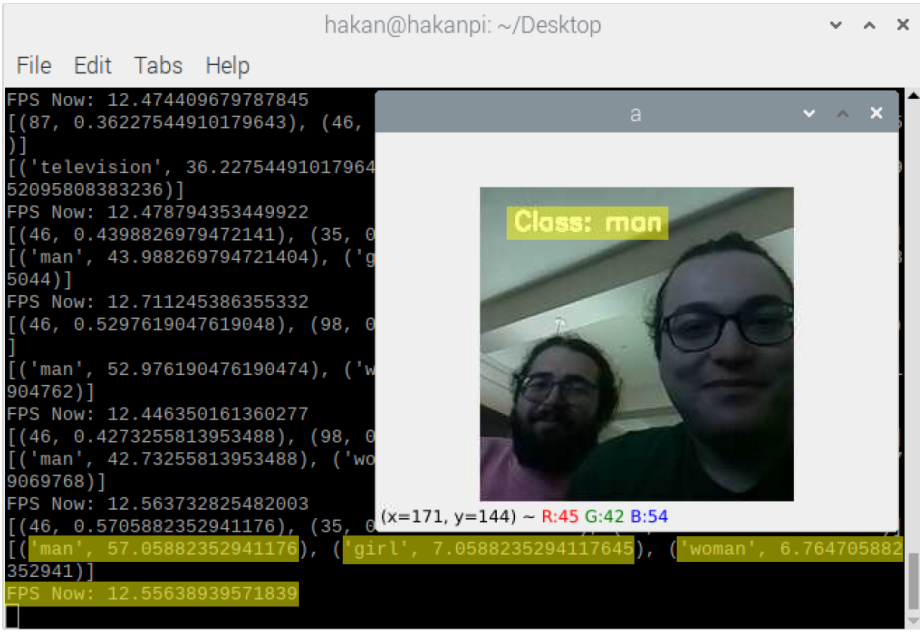

Figure 6: Working example of the real time inference with the Xception model.

4 Conclusion and Discussion

It should be noted that, our first concern was not to get a super-high accuracy. This is already done with better training and better data. The architecture of the model is mostly what changes the inference time. Therefore this project focuses on the FPS and the inference time. The only accuracy that is worth discussing is the decrease in accuracy when we utilize PyArmNN on top of TFLite, which in all cases, was an increase less than 1% or 2%, which we think is super cool.

Two of the models we have tested, namely NasNetLarge and Xception, were originally not "real time" models. When we use them with ARM CV/ML Library, we bring this feature to those models. On the other side, with MobileNetv2, we gained less increase compared to the much heavy models. This is because MobileNetV2 is already a lightweight model that can run on weak devices. Therefore, we believe it is better to use this library on models that are not real-time, so that heavy models with high accuracy and low FPS values can be turned into models that have comparable accuracy to the real time models and a real time FPS.

In this project we compared Arm NN inference engine to a different and existing solution (Coral TPU) for Image Classification DNN acceleration. We can see that there is an increase around 2 to 3 times in FPS, just by utilizing

Arm NN engine. By doing so, we did lose on accuracy no more than 1% or 2%, which makes this library a really good solution to run a "non-realtime" model as a "realtime" model.

5 Codes and Files

We provided our codes all inside a .zip file. "transferlearning.ipynb" is our notebook for training images on CIFAR100 dataset. "benchmark.py" shows a simulation over a number of test images (we used 1000). "x_test.npy" and "y_test.npy" are test data from CIFAR-100. "labels.py" has some functions for us to use in other files and does not need to be run. "classify.py" shows a demonstration with Picamera and .tflite files.

6 Mini-Lab: Google Colab with GPU

We included the notebook for the training and testing part of minilab in the files. Here is the testing and training curves for our network using GPU.

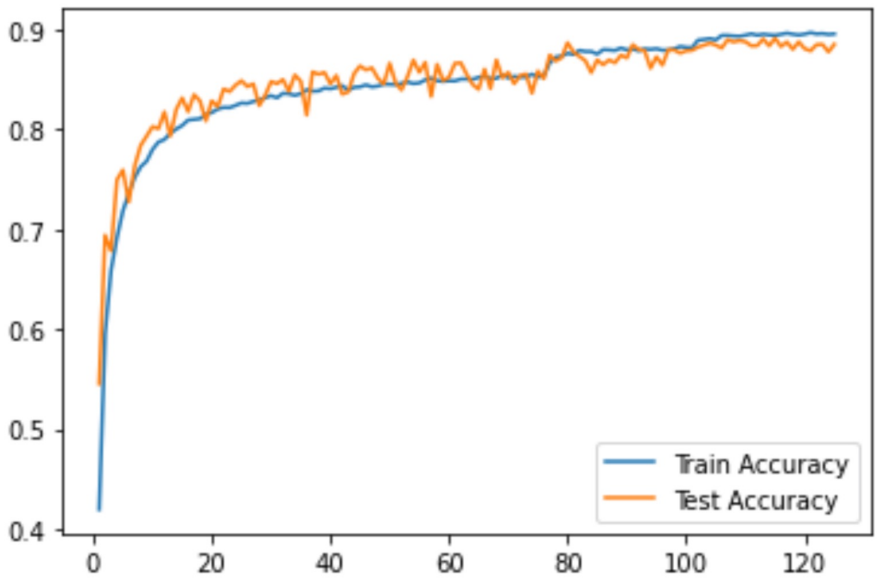

Figure 7: Accuracy vs. epochs

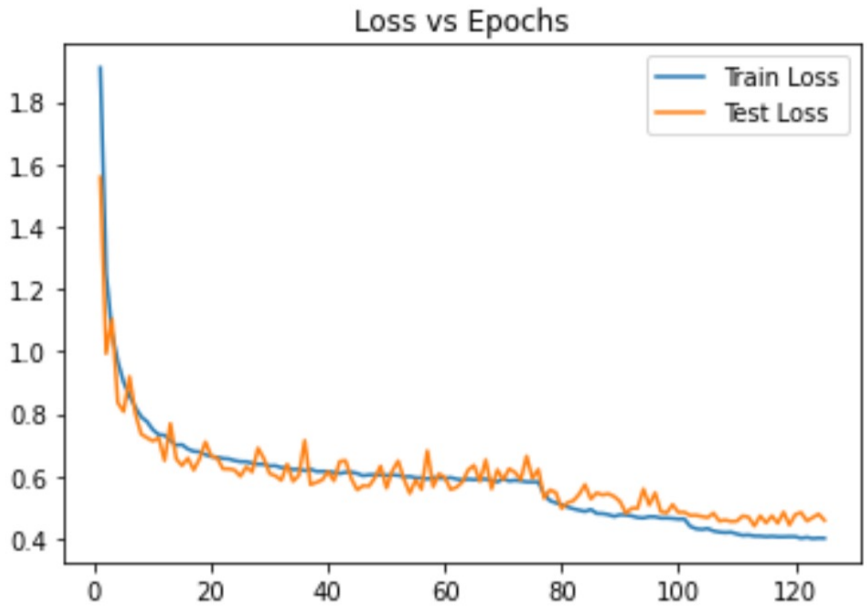

Figure 8: Loss vs. epochs

The next image shows that we got +85% accuracy on our testing data, after training.

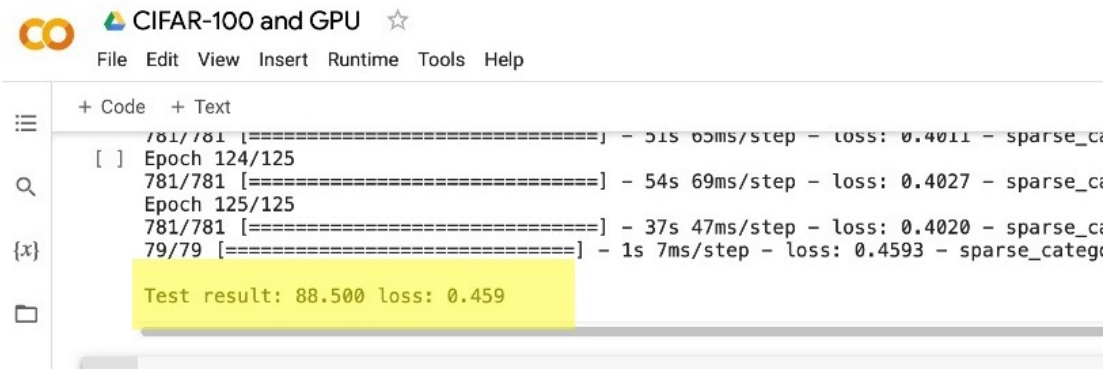

Figure 9: Caption

References

[1] A. Ltd., *Compute Library – Arm® — arm.com*, https://www.arm.com/technologies/compute-library, [Accessed 02-May-2023].

[2] *ArmNN — developer.arm.com*, https://developer.arm.com/Tools%20and%20Software/ArmNN, [Accessed 02-May-2023].

[3] *Armnn/build-tool at f233be4f05a5b50ae59ad48efa341d7312d2ad08 · ARM-software/armnn — github.com*, https://github.com/ARM-software/armnn/tree/HEAD/build-tool, [Accessed 02-May-2023].

[4] *Documentation x2013; Arm Developer — developer.arm.com*, https://developer.arm.com/documentation/102107/0000/Before-you-begin, [Accessed 02-May-2023].

[5] *Armnn/python/pyarmnn at v22.05.01 · ARM-software/armnn — github.com*, https://github.com/ARM-software/armnn/tree/v22.05.01/python/pyarmnn, [Accessed 02-May-2023].

[6] *MobileNetV2: Inverted Residuals and Linear Bottlenecks — arxiv.org*, https://arxiv.org/abs/1801.04381, [Accessed 02-May-2023].

[7] *Xception: Deep Learning with Depthwise Separable Convolutions — arxiv.org*, https://arxiv.org/abs/1610.02357, [Accessed 02-May-2023].

[8] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, *Learning transferable architectures for scalable image recognition*, 2017. eprint: arXiv:1707.07012.

[9] A. Krizhevsky, *Learning multiple layers of features from tiny images*, https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf, [Accessed 02-May-2023].