DSA4212 Assignment 1
Group 3

Nicholas Russell Saerang, A0219718W, e0550376@u.nus.edu
Clifton Felix, A0219735X, e0550393@u.nus.edu
Jason Ciu Putra Sung, A0219734Y, e0550392@u.nus.edu

1 Introduction

The goal for this project is to experiment with models and find the model with the highest test accuracy to predict the provided test data with a maximum of 120s training time (excluding the pre-processing and evaluation stages). All models presented in this report are trained within the 120s time limit starting from randomized weights, and the test dataset is not involved during the model training.

The dataset used in this project consists of 9,296 training and 3,856 test images with labels. There are 10 different labels in this dataset: fish, dog, device, chainsaw, church, horn, truck, petrol, golf, and parachute. Each image is of size 128x128x3.

2 Experiments

2.1 Overview

2.1.1 VGG

The VGG (Visual Geometry Group) CNN model according to [SZ14] is a convolutional neural network architecture designed by the Visual Geometry Group at the University of Oxford, introduced in the 2014 ImageNet Large Scale Visual Recognition Challenge (ILSVRC) and had the highest accuracy at that time. The model's architecture is based on the straightforward notion of employing smaller convolutional filters (3x3) rather than bigger ones to enhance the depth of the network. It is made up of a number of convolutional layer blocks, followed by max-pooling layers and fully linked layers.

The deep architecture of the VGG model is one of its primary characteristics. There were 16 layers in the initial VGG model, which was more than any other CNN model at the time. Better performance on image classification tasks may result from the model's depth, which enables it to learn more intricate features from the input photos. However, the model is also more computationally expensive and challenging to train than shallower models due to the deep design.

2.1.2 ResNet

Residual networks, or ResNet in short, according to [Sah18], is another type of CNN model where each layer feeds into the next layer like other usual CNN models, but in addition, it also feeds into a few layers after it, usually 2-3 layers.

The motivation behind ResNet is the existence of the vanishing gradient problem and

accuracy degradation caused by the excessive depth of the neural network. The ability of ResNet to skip some layers allows us to have shallower sub-networks and avoid those accuracy bottlenecks. The skipping can be done using skip connections or residual connections.

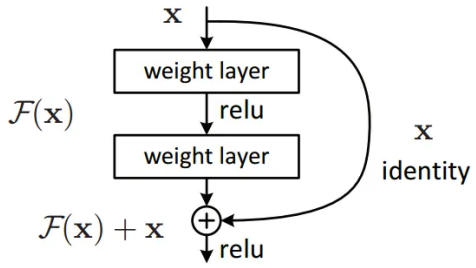

Figure 1: ResNet skip connection [HZRS16]

The image above shows how a skip connection works. Given the input x, the whole network block is trying to learn the true distribution $\mathcal{H}(x) = \mathcal{F}(x) + x$. Due to the presence of the identity connection, the network (consisting of several weight layers and activation functions) is actually trying to learn the residual distribution $\mathcal{F}(x)$, which is the same objective of the neural network block if we remove the skip connections.

However, the presence of skip connections helps us to compute simpler functions as well as propagate larger gradients from final layers to initial layers so that these layers can still learn fast enough without having to experience the vanishing gradient problem.

2.2 Data Preprocessing

Note: The baseline JAX model uses **only** image shuffling, while the 3-block VGGNet uses **only** image shuffling, image resizing, and data augmentation.

2.2.1 Image Resizing

There are several reasons why this is done:

- Memory limit. Without resizing, we only have a limited amount of images for training but data augmentation will exceed the given Google Colab memory limit and thus cannot be done.
- Efficiency. With a smaller image, we can still obtain the important features of an image, so the model training can run faster while maintaining its final accuracy.
- Compatibility. Several models used as references for this experiment are initially used for the CIFAR-10 dataset [Kri], whose training sets are images of size 32x32x3, thus matching the intended image size.

In this experiment, we used OpenCV's `resize` method with inter-area interpolation to resize all images into a 32x32x3 image.

2.2.2 Data Augmentation

Data augmentation is also performed to avoid overfitting. To augment the training images, we used Tensorflow's `ImageDataGenerator` object that allows us to flow a stream of images where different augmentation methods are applied in random intensities, such as rotation, width shift, height, shift, horizontal flip, shearing, and zooming.

The image below shows different augmentations applied to a set of 5 images.

Figure 2: Data augmentation

2.2.3 Image Shuffling Before Training

The original dataset is initially sorted by the label, so all images of the same class are clustered. This might affect the training process in a negative way (such as bias), and therefore shuffling is done to preserve the equal expected distribution across all classes.

2.2.4 Image Normalization

Any images used for training or evaluation will be normalized by the mean of the training dataset values as well as its standard deviation. This is done to ensure the integrity of the pixel intensity values across all datasets.

2.2.5 Image Padding

The training set is padded using NumPy's `pad` method with a border of 4 so the first and last 4 rows and columns are reflected outwards to create a 40x40x3 image. This is used in tandem with the next preprocessing technique, cropping-cutout.

2.2.6 Image Cropping and Cutout

Out of the padded image, we took a large portion of the picture (32x32x3) and used that for training instead. The cropping that starts from a randomly selected point on the image will allow more variety on the dataset on top of the one data augmentation provided. Lastly, a small portion (12x12x3) of the cropped image is cut out (pixels set to 0) to test if the model still recognizes the image despite the presence of a small "hole" on it.

2.2.7 Further Data Splitting

The training dataset is split into 80% pure training dataset and 20% validation dataset to help with the hyperparameters fine-tuning process.

2.3 Models

2.3.1 Baseline: JAX, FLAX, OPTAX

For this assignment, we used the given model as our baseline and fine-tuned some of its parameters. The initial model implemented the VGG architecture with three blocks consisting of normalization, convolutional, and ELU activation layers, followed by pooling, dense, and softmax layers. The three convolutional layers each had 32, 64, and 128 kernel filters of size 3x3 and stride 2.

We implemented this model using optimization libraries such as JAX, FLAX, and OPTAX. JAX optimized the entire training process, including the computation of the loss function, gradient descent, and so on, while FLAX served as the neural network framework. Lastly, we used the ADAM optimizer from the OPTAX library with a learning rate of 0.01.

We trained this model on the training data for approximately 65.5 epochs in 107.9 seconds with a batch size of 512. At the last iteration, the model achieved a training accuracy of 94.8%, a validation accuracy of 70.2%, and a test accuracy of 69.7%. This suggests that the model overfits on the training data.

To prevent overfitting, we reduced the number of layers in the model. Specifically, we reduced the model to two blocks by removing the last block consisting of normalization, convolutional, and ELU activation layers. After some fine-tuning, we found that the best performance was achieved when each convolutional layer had 32 and 64 filters, and the learning rate for the ADAM optimizer was set to 0.003. We trained the model for 113.7 epochs and found the parameters with the highest validation accuracy. With this model, we achieved a training accuracy of 85%, a validation accuracy of 71.8%, and a test accuracy of 72.8%.

We also improved the model by shuffling

the training data after every epoch. According to [Zin], the training data between epochs helps prevent overfitting and reduce variance by ensuring that each batch is more representative of the whole training data. Instead of fixing the batch size, we specified the number of steps per epoch to 15 and shuffled the data after every 15 batches.

After training this final baseline model for 110 epochs, we selected the best parameters based on validation accuracy and achieved a training accuracy of 82.8%, a validation accuracy of 73%, and a test accuracy of 73.8%.

2.3.2 3-Block VGGNet (VGG8)

Improving upon the baseline model, we created a 3-Block VGGNet with a total of 8 layers, which is similar to the baseline model. However, instead of FLAX, we utilized TensorFlow's Keras library. The overall architecture is based on [Bro20].

Each of the three VGG blocks in our model consists of a pair of convolutional layers with 32, 64, and 128 kernel filters of size 3x3, respectively, followed by a batch normalization layer after each convolutional layer. After each block, we pass the result to a max pool layer of size 2x2 and then set a dropout rate that increases after each block (0.2, 0.3, and 0.5). Finally, the output of the last max pooling layer is passed into 2 dense layers, one of size 128 and the other of size 10. All layers use the RELU activation function, except for the last one, which utilizes softmax.

After resizing images and data augmentation and performing fine-tuning, we discovered the optimal configuration, setting the batch size to 256 and having the learning rate employ an exponential decay of rate 0.8, starting at 2e-3. The model was then trained for 15 epochs and achieved a training accuracy of 88.8% and a test accuracy of 78.6%.

2.3.3 Final: DAWNBench ResNet9

The customized ResNet9 model from David Page [Pag] for the Stanford DAWN Deep Learning Benchmark (DAWNBench), along with all its utility functions, gave us the best result, which was approximately 99.8% training accuracy and around 85.6% test accuracy. Therefore, it became the final deep learning pipeline on top of the given data preprocessing.

The network's architecture consists of a single preparation layer, three main layers, and a wrap-up part. The preparation layer comprises a single convolution-normalize-activate block with three input channels for the convolution layer. Each of the main layers includes a single convolution-normalize-pool-activate block with 64, 128, and 256 input channels, respectively, and 128, 256, and 512 output channels, respectively, for each convolution layer. Both the first and last main layers have one residual block (after the activation function) that consists of two convolution-normalize-activate blocks with the same input and output channels as the respective main convolution layers. Lastly, the wrap-up part is a combination of a global pooling layer, a flatten layer, a dense layer, and the final output layer. Overall, there are nine convolution layers and linear layers combined, hence the name ResNet9.

Each of the convolution layer weights was initialized with random values using the Xavier uniform initializer instead of the default Kaiming uniform initializer. There is no rigorous reason for such a choice other than it being a matter of preference. Moreover, these convolution layers have the same kernel size of 3x3, the same stride of 1, and the same padding of 1. Right after each convolution layer comes the batch normalization layer with default values of ϵ, 1e-5, and momentum, 0.1.

The bag of tricks provided at [Pag19] suggested that we perform two things: freeze the

normalization layer weights to improve training time and prevent the model from being overcomplicated, and put the pooling layer right before the activation layer so that the activation function is computed on a smaller tensor, thereby improving the training time and accuracy (previously around 85.3%). The model uses maximum pooling instead of average pooling, customized with a kernel size of 2x2. The only difference is the final pooling on the wrap-up part, which uses a kernel size of 4x4 instead of 2x2. Another trick that we applied to the final model is the use of the Continuously Differentiable Exponential Linear Unit (CELU) activation function instead of the ReLU activation function. Further experiments validated our claim that it slightly outperforms ReLU (which has approximately only 84% test accuracy), and hyperparameter tuning led us to discover the best value of α for the CELU activation function, which is 0.05.

The learning rate is a piecewise linear function that also determines the number of epochs for training. It increases linearly up to 0.4 for four epochs, then decreases linearly to 0.03 for the next 31 epochs, then decreases linearly with a lower rate all the way down to 0 for the next 25 epochs. The optimizer used for the final model training is a customized Stochastic Gradient Descent (SGD) provided by [Pag] with a weight decay of 0.064 and a momentum of 0.9. The weight decay is linearly proportional to the batch size, which we selected to be 128.

The training pipeline concluded after 51-53 epochs with a timeout callback. As previously mentioned, the training accuracy is approximately 99.8%, and the test accuracy is about 85.6%. Additionally, the validation accuracy is about 87.4%. These results may vary due to the time it takes for the first epoch to load all the data into the GPU and the unpredictable randomization performed by the GPU. However, the differences should not be significant.

3 Bibliography

[Bro20] Jason Brownlee. How to develop a cnn from scratch for cifar-10 photo classification, Aug 2020. URL: https://machinelearningmastery.com/how-to-develop-a-cnn-from-scratch/.

[HZRS16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[Kri] Alex Krizhevsky. Cifar-10 and cifar-100 datasets. URL: https://www.cs.toronto.edu/~kriz/cifar.html.

[Pag] David Page. Davidcpage/cifar10-fast. URL: https://github.com/davidcpage/cifar10-fast.

[Pag19] David Page. How to train your resnet, Nov 2019. URL: https://myrtle.ai/how-to-train-your-resnet/.

[Sah18] Sabyasachi Sahoo. Residual blocks — building blocks of resnet, Nov 2018. URL: https://medium.com/@ssahoo.ai/fd90ca15d6ec.

[SZ14] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[Zin] Clark Zinzow. Per-epoch shuffling data loader: Mix it up as you train! URL: https://www.anyscale.com/events/2021/06/22/per-epoch-shuffling-data-loader.