Activity 18 – Convolutional Neural Networks

Kenneth V. Domingo
2015–03116
App Physics 186, 1st Semester, A.Y. 2019–20
Corresponding author: kvdomingo@up.edu.ph

1 Introduction

For this activity [1], I will be using the Keras API with the TensorFlow backend with GPU support to make training faster, to easily monitor loss values and metrics, and quickly tune the network's hyperparameters to our liking. All experiments were performed on a laptop running on Windows 10, Intel Core i5-7300HQ, 24GB RAM, and an NVIDIA GeForce GTX 1060 with 6GB VRAM.

2 Dataset

I will be using this dataset [2] for training and this dataset [3] for testing. Both datasets contain thousands of color images of cats and dogs that are roughly equally distributed. The images are not staged so they may contain other objects in the frame other than the intended subject, or even more than one cat/dog. The images are named by the class label followed by its sequence number (e.g., `cat.0001.jpg`). This makes it easier to pre-process later on.

2.1 Input pipeline

We store the image filenames in a dataframe and not the images themselves. This is because due to the number of images, it is unlikely that they will fit in the GPU memory all at once. After a few trials, the optimal batch size was determined to be 128. This means that only 128 images will be in memory at a time. The data flow is as follows:

1. Image is read from persistent storage (such as a hard drive).
2. Image is loaded to RAM and CPU performs pre-processing.
3. Image is copied to GPU memory for training.

In the above steps, transfer from RAM to GPU contributes most of the overhead, and usually, the GPU can be trained on the entire dataset much faster than the CPU can preprocess it. In order to optimize the use of resources, we parallelize the CPU and GPU operations such that the CPU prepares the next batch while the network is being trained on the GPU. A visualization of the parallelization as compared to standard serial programming is shown in Fig. 1.

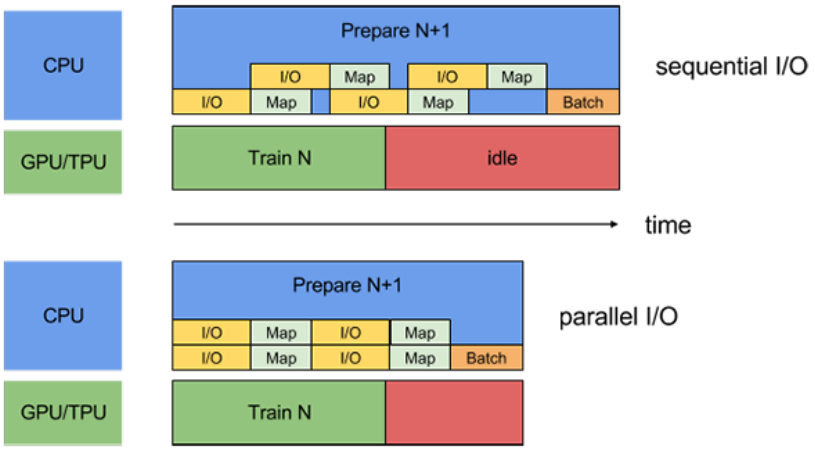

Figure 1: Comparison of standard serial programming with the parallelization scheme [4].

2.2 Pre-processing

Upon importing the images from disk, we rescale the image so that its values take up the range $[0, 1]$. Next, we augment the data by performing random affine transformations such as rotation, shear, flip, zoom, and translation. Finally, all images are reshaped to 224×224 px. 20% of the training set will be reserved for validation. The parallelization scheme will be handling the batching of the images.

3 Architecture

Our convolutional network architecture draws ideas from the designs of [5, 6]. The visualization of the network is shown in Fig. 2.

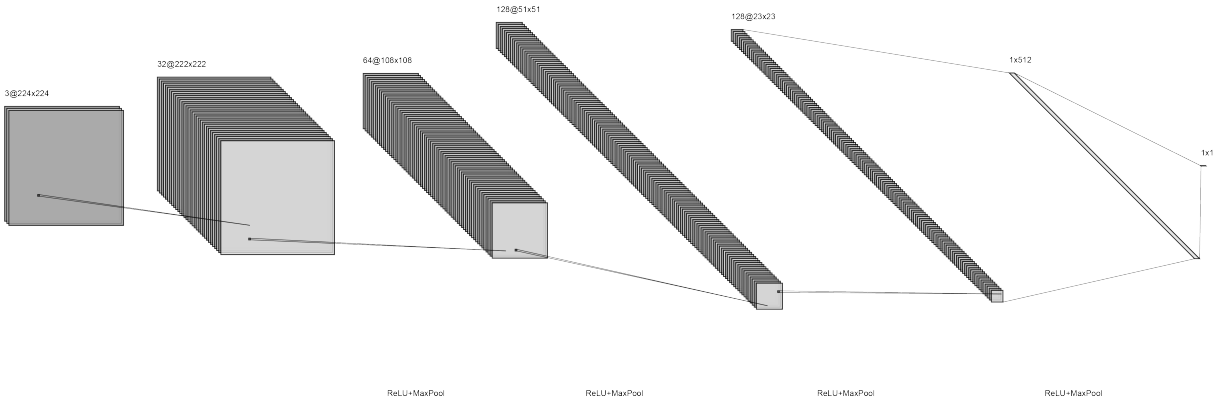

Figure 2: The architecture of the convolutional network used. Visualization generated using [7].

Our input image has a dimension of $224 \times 224 \times 3$. The first layer is a 2D convolutional layer with 32 filters and a kernel size of 3×3, followed by a ReLU activation, and then by a max pooling layer with a pool size of 3×3 and a stride of 2. This is succeeded by 3 more layers which repeat the same sequence, but with 64, 128, and 128 convolution filters, respectively. After the last convolutional layer, a dropout layer with a factor of 0.5 is employed. This randomly sets a fraction of the previous layer's weights to zero—in this case, half of the weights—to prevent overfitting. This is followed by a 512-unit fully-connected layer, activated by ReLU. The output layer is a lone fully-connected unit which

is activated by a sigmoid. All layers are initialized with a random Gaussian distribution of zero mean and unit standard deviation, and all biases are initialized to zero. Because our output has only one node but we expect it return a binary value, we select our loss function to be the binary cross entropy, defined as

$$H = -\frac{1}{N}\sum_{i=1}^{N} y_i \log[p(y_i)] + (1 - y_i)\log[1 - p(y_i)] \tag{1}$$

where y_i is the label, and $p(y_i)$ is the predicted probability of an image of having label y_i for all N images [8]. For the optimizer, we use stochastic gradient descent (SGD) with learning rate $\eta = 0.01$, momentum $\mu = 0.9$, and a decay of 5×10^{-4}. We also use a learning rate scheduler, which reduces the learning rate by a factor of 10 if the validation loss does not improve over 5 epochs. Lastly, we also utilize a checkpointer, which saves a snapshot of the model weights if the validation loss decreases after an epoch. While monitoring the loss and accuracy, we now train the network until the validation loss drops below 0.1 or when the validation loss starts to diverge from the training loss.

4 Results and Discussion

Figure 3 shows the loss and accuracy curves after training the network. At epoch 13, the validation loss is minimum, and starts to diverge from the training loss afterwards. Training was manually stopped at epoch 29 after observing no further improvement in validation loss. We restored the best weights which were obtained at epoch 13, which had a validation loss and accuracy of 0.33 and 87.00%, respectively. Overall training took around 80 minutes.

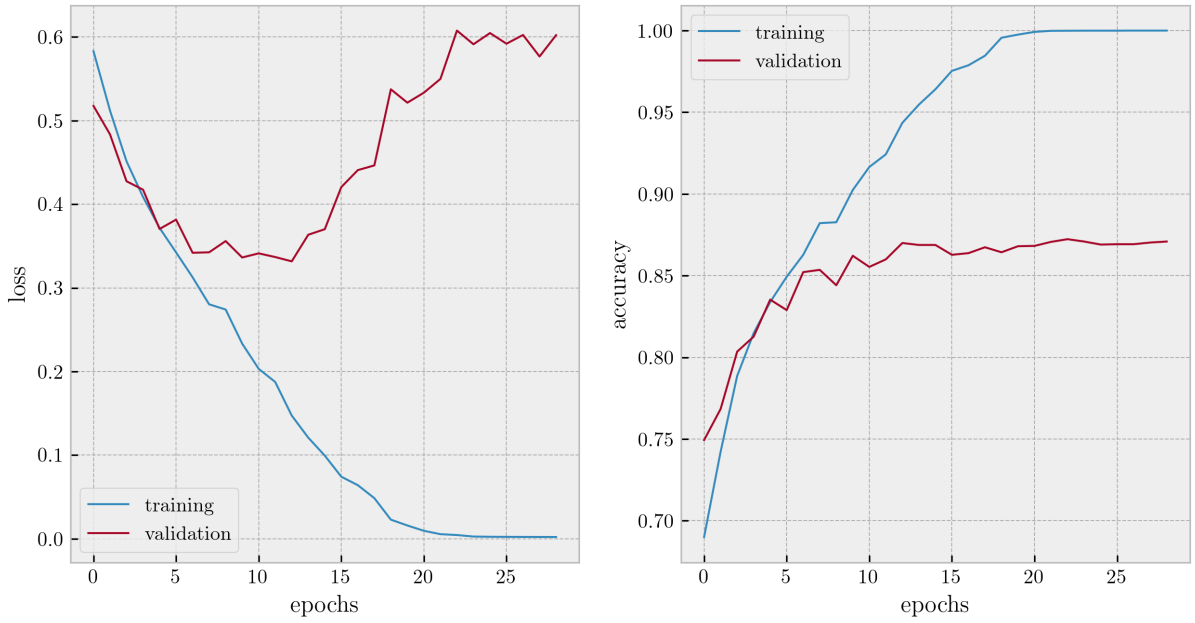

Figure 3: Loss and accuracy curves for the training and validation sets.

Feeding the test images in the network yields a loss of 0.15 and an excellent 94.95% accuracy. Sample predictions are shown in Fig. 4. Only one out of the 18 predictions is wrong. Upon observation we can see why the network may have been confused: due to the dark color of the cat.

For increasing the accuracy of the network, one may consider adding another dropout layer after the deep fully-connected layer, playing around with the convolutional layer parameters, and using a deeper network.

Table 1: Self-evaluation.

Technical correctness	5
Quality of presentation	5
Initiative	2
TOTAL	**12**

References

[1] M. N. Soriano, *A18 – Convolutional Neural Networks* (2019).

[2] Kaggle, Dogs vs cats (2015), https://www.kaggle.com/c/dogs-vs-cats/data.

[3] Kaggle, Cats and dogs dataset to train a DL model (2017), https://www.kaggle.com/tongpython/cat-and-dog.

[4] A. Agarwal, Building efficient data pipelines using TensorFlow (2019), https://towardsdatascience.com/building-efficient-data-pipelines-using-tensorflow-8f647f03b4ce.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ImageNet classification with deep convolutional neural networks, *Advances in Neural Information Processing Systems* **25**, 1097 (2012).

[6] L. Strika, Convolutional neural networks: A Python tutorial using Tensorflow and Keras (2019), https://www.kdnuggets.com/2019/07/convolutional-neural-networks-python-tutorial-tensorflow-keras.html.

[7] A. LeNail, NN-SVG: Publication-ready neural network architecture schematics, *Journal of Open Source Software* **4**, 747 (2019).

[8] D. Godoy, Understanding binary cross-entropy/log loss: a visual explanation (2018), https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a.

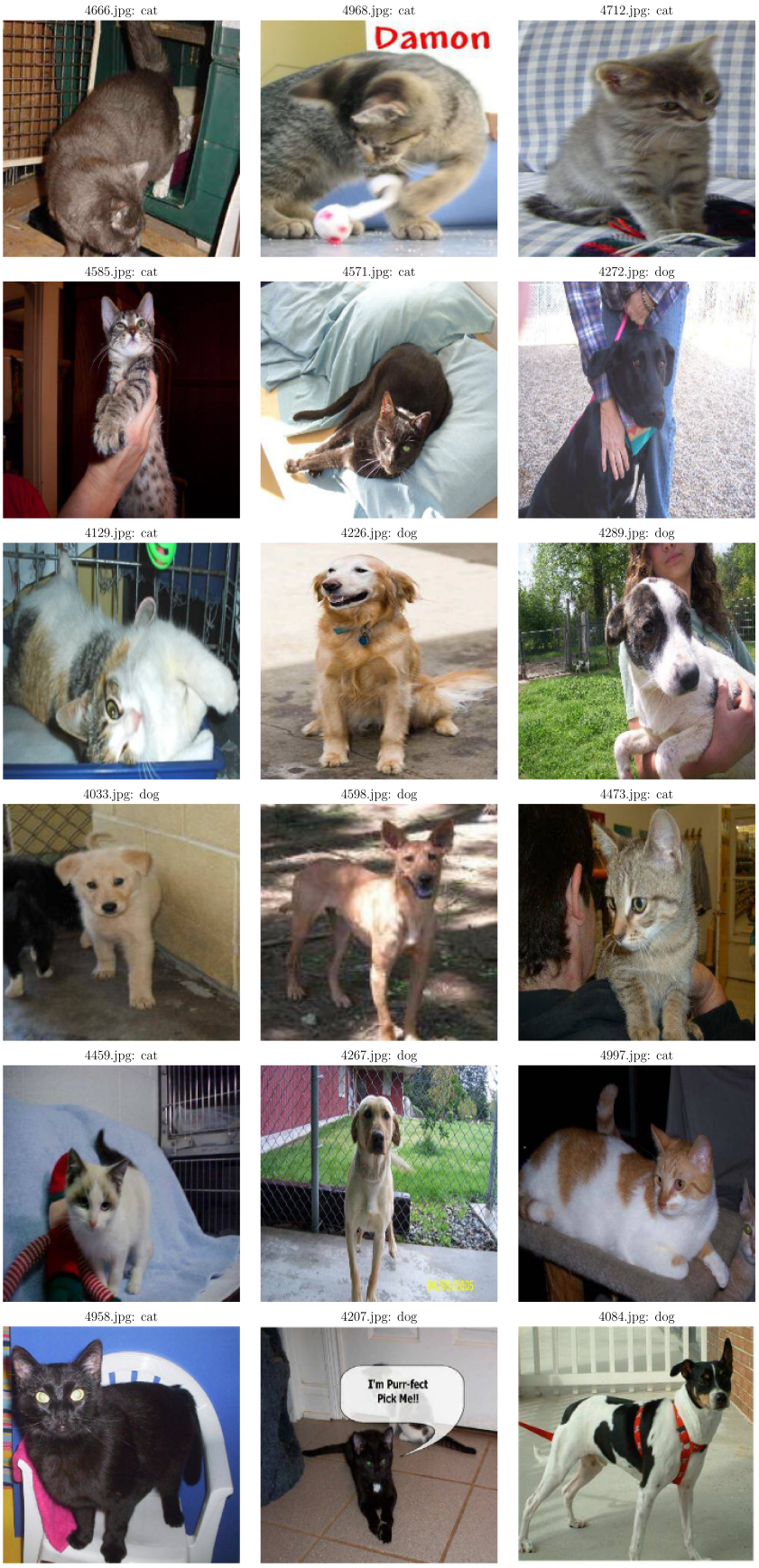

Figure 4: Sample filenames and predictions of cat and dog images.