Pooling Layer Analysis on Convolutional Neural Networks

ADL Final Project

Nick Fogg  
 Computer Science  
NC State University  
 Raleigh NC USA  
 nsfogg@ncsu.edu

David Sweasey  
 Computer Science  
 NC State University  
 Raleigh NC USA  
 dbsswease@ncsu.edu

ABSTRACT

In deep learning, convolutional neural networks (CNNs) are one of the main types of neural networks responsible for image recognition and classification. CNNs are widely used across various applications, some being detecting objects, recognizing patterns, or classifying facial expressions. CNNs pass through an alternating series of convolution and pooling layers where they process the input image and determine its classification. In this report we explore different algorithms that can be utilized in the pooling layer to compress the image and ensure easier detectability. We test four different pooling layers: max, average, adaptive max, and adaptive average. To provide a more comprehensive analysis we tested these pooling layers on three diverse types of datasets: MNIST handwritten digits, CIFAR10 colored objects, and FER2013 facial expressions. Through our testing we aim to find the best pooling method that should be used for each scenario, in hopes of maximizing accuracy and efficiency.

1. INTRODUCTION

CNNs are primarily used for computer vision: the perception of the world as humans by computers. [1] In real-world applications, both efficiency and accuracy of computer vision is paramount. Pooling layers are a primary factor in computational speedup within a CNN. They perform a “dimensional shrink” to significantly decrease the number of computations that must be completed. However, much of the previous layer’s information was lost, and in some cases, this information contained pertinent details. [1]

During a preliminary examination of the current CNN structure, it was identified that there are potential slowdowns in the algorithm, specifically regarding the pooling layers. Max pooling tends to outperform average pooling, but due to the nature of the algorithm it does so with a loss of information. Too much loss in detail can result in a much lower model accuracy [1]. To address this challenge, we are considering altering the current approach to pooling in the CNN algorithm. An in-depth examination of when max pooling prevails over average pooling and vice versa is required to gain insight into the mechanisms behind CNNs.

We want to explore ways of optimizing pooling layers to minimax computational overhead and output accuracy. In many models, the hyperparameters serve as the basis for how pooling layers behave. The parameters include filter size and stride. [1] Although these parameters are important, they are not the focus of the paper, and we do not want to introduce too many variables.

Once the parameters are set, another problem arises: data preservation. The different forms of pooling yield differing results. Average pooling preserves a bit of all data by computing the average value of a region whereas max pooling returns the largest value. We hope to find a healthy balance between the two methods, potentially with an additional parameter corresponding to a weighted average that combines the benefits of average and max pooling. Another aspect of this pooling procedure is the adaptive process. In newer CNNs, work in adaptive CNN models is being used. This adaptive model builds on the previous fusion operations instead of creating its own. Since CNNs operate by repeatedly alternating convolution and pooling layer, that data from other layers is utilized in every decision henceforth. This history can allow for an improved perception of what features need to be necessitated.

By the end of the project, we hope to find optimizations in pooling layer methods such that our CNN is accurate, efficient, and training requires fewer iterations of backpropagation.

2. METHODS

To evaluate the performance of different pooling techniques, we constructed a simple convolutional neural network to use for three datasets. The network includes three convolutional layers, each followed by the rectified linear unit (ReLU) activation function, finally followed by three pooling layers. Then, to model a typical CNN [2], the tensor is flattened and ran through two fully connected layers then yet again into ReLU, and then a dropout is applied with a probability of 0.3. Dropout is a proven technique to prevent overfitting [3], so is included to reflect an ideal real-world model. A single fully connected layer acts as the final and output layer of the model. We employed cross entropy loss as our loss function, and Adam as the optimization function, and used a learning rate of 0.0001.

The entirety of the model is built within Python, and we integrated PyTorch as the neural network dependency. PyTorch provides functionality to seamlessly implement a neural network, import and transform data, and set parameters and hyperparameters. Using DataLoaders [4], data is seamlessly broken up into batches, suffled, (and tensorized, if necessary), and parallelization can be optimized by specifying a *num\_workers* parameter. We chose two workers, a reasonable number for most platforms; however, this parameter is not terribly important for this paper since the focus is on pooling layers. For the same reason, a small *batch\_size* of 8 was chosen. Although a larger batch size may speed up training, it can have detrimental effects on a model’s accuracy [5]. To train and validate the data, DataLoaders can be iterated on, allowing for seamless creation of training and testing loops. In Figure 2.1, we present the code for using a PyTorch DataLoader. The *train* dataset is used to train the model, and the *validate* dataset is used to test the accuracy of the model after each epoch.

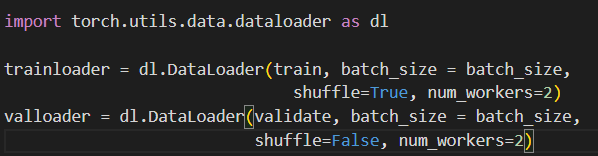


Figure 2.1: DataLoader implementation

2.1 Data Input

Within PyTorch, many datasets are provided through *torchvision.datasets*. We chose MNIST, a set of 28x28 black and white images of handwritten numbers; and CIFAR10, 32x32 color images of various objects, such as an airplane or a dog. [6] Figure 2.2 shows the code used to

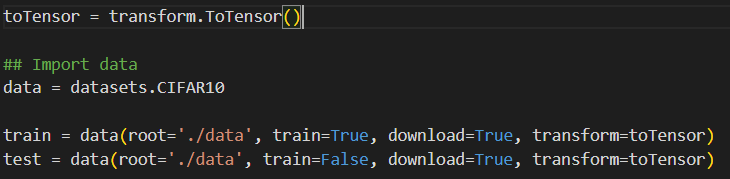


Figure 2.2: Importing torchvision dataset

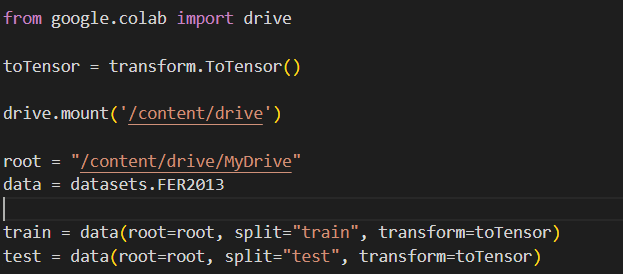


Figure 2.3: Importing FER2013 dataset

import these datasets. Figure 2.3 shows the code for importing an external dataset, FER2013 (facial recognition) for our experiments. This method is similar, but it requires accessing a Google Drive for the CSV files. Note the *transform=toTensor*. PyTorch allows the user to easily specify input data transformations; we only used *toTensor* to not introduce confounding variables.

With the data successfully imported, we then create DataLoaders to be used in the loops, as mentioned before.

2.2 Train and Validate Loops

PyTorch does not provide a built-in looping mechanism; instead, users must rely on DataLoaders as an iterator and build it based on them. However, the code is not too complex, and was based on PyTorch documentation [7]. Essentially, each iteration takes a batch of inputs (we set at 8), runs it through the model, calculates accuracy and loss, backpropagates this loss, and then runs an optimizer. Our loop reports running accuracy and loss every 500 batches as a sanity check to ensure the model is improving and not just guessing.

The validation loop is similar, it simply lacks the report every 500 batches. Instead, it reports the total accuracy and loss on the entire validation set. These values represent the accuracy of the model after an epoch and are used as the data for our experiments.

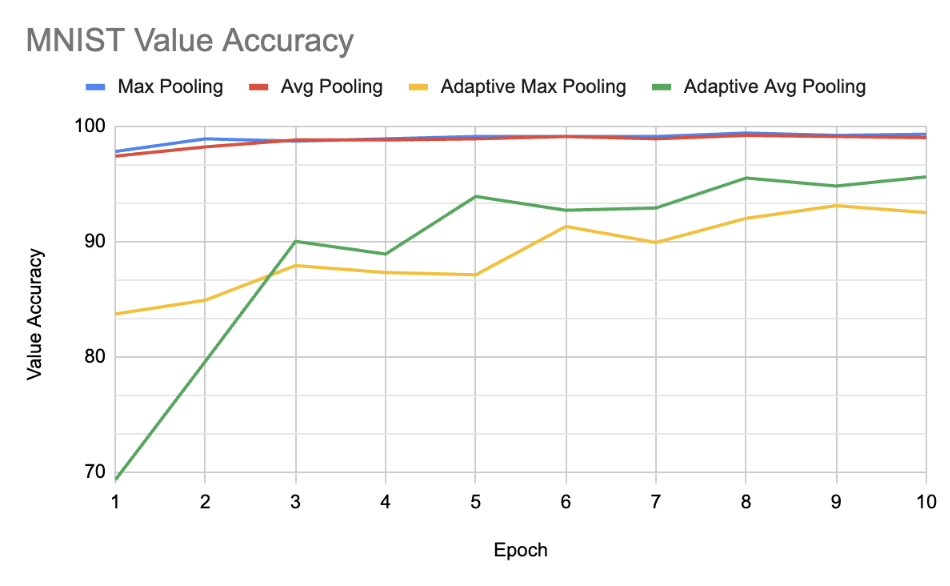
3.3 Experimentation

As mentioned previously, three separate datasets were chosen for classification, and four pooling techniques were tested on each. The four techniques are max pooling, average pooling, adaptive max pooling, and adaptive average pooling. The latter two are built-in PyTorch layers, but PyTorch documentation does not provide information on what specifically they do. The CNN constructed contains three pooling layers; all three are set before each trial. For each trial, the validation accuracy after each epoch was recorded as well as the time to complete training. We utilized Google Colab’s T4 GPU to run these trials. A high-end GPU is important to not introduce any unwanted bottlenecks, and it is closer to industry standards. Runtime and accuracy were plotted separately for each pooling method. These experiments were repeated three times, once for each dataset.

3. RESULTS

3.1 MNIST Dataset

The MNIST dataset is a set of 70,000 28x28 pixel images. The dataset has images of handwritten digits from zero to nine in various handwritings. The goal for the CNN in this dataset is to classify the image to the digit it is trying to describe.

Figure 3.1: MNIST Value Accuracy

The above figure depicts the accuracy results of max pooling, average pooling, adaptive max pooling, and adaptive average pooling. Each CNN is tested over ten epochs, where the trends start to flatten out. Max and average pooling perform similarly, with max pooling starting with slightly better performance. Adaptive average pooling has very poor initial performance but over time has better accuracy than adaptive max pooling.

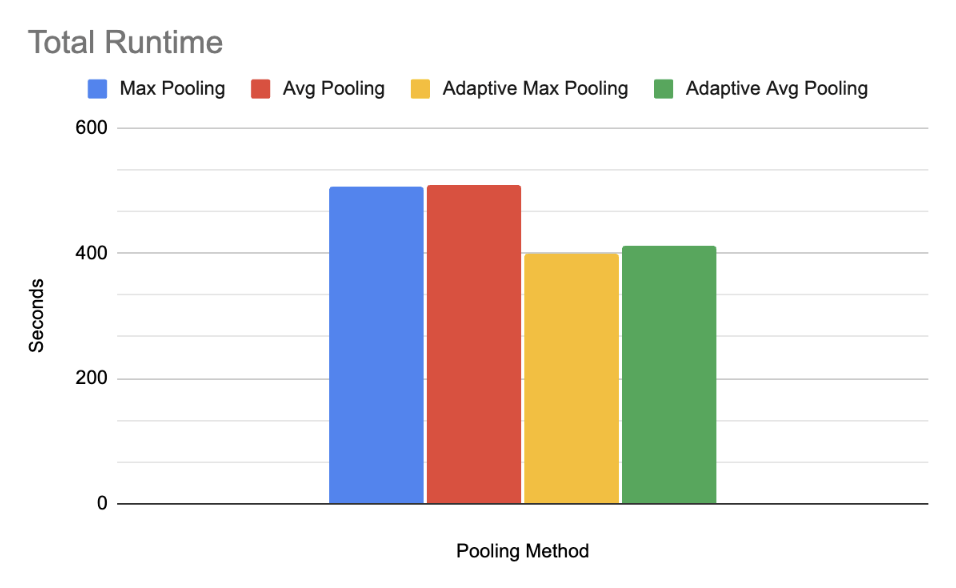
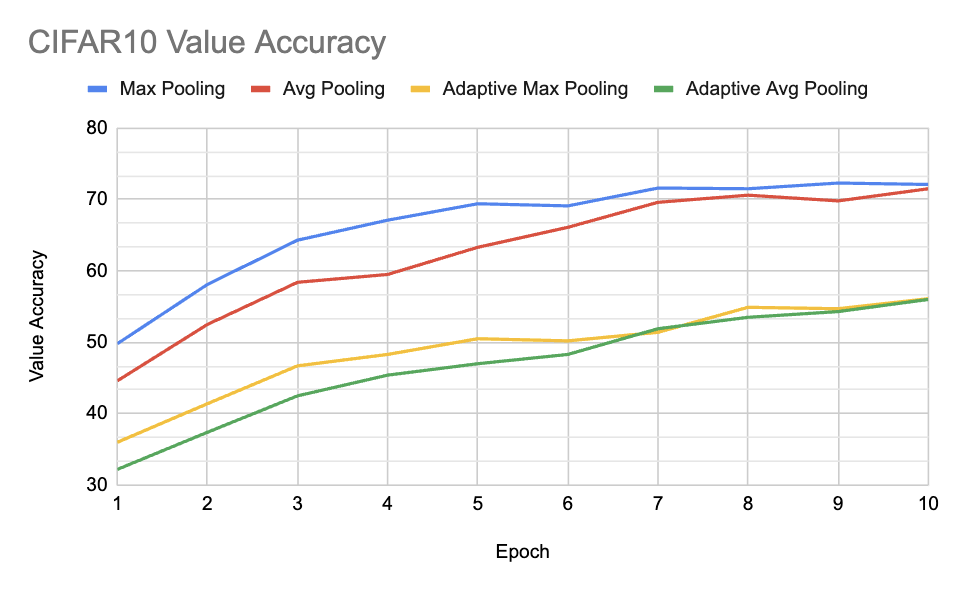
Figure 3.2: MNIST Runtime

Figure 3.2 shows respective runtimes for each pooling layer used. It is noted that while max and average pooling have higher accuracy, they tend to take longer to run as well. The difference in runtimes between max and average and their respective adaptive counterparts is around 100 seconds.

3.2 CIFAR10 Dataset

The CIFAR10 dataset is a set of 60,000 32x32 color images. The dataset has 10 classes, each a different labeled object. The objective for the CNN is to identify the object the small image is depicting.

Figure 3.3: CIFAR10 Value Accuracy

The graph in figure 3.3 describes the model accuracy after ten epochs. The first note is overall worse performance than the MNIST dataset as this is a much more complex classification problem. The same trends are seen with max and average being in a higher class than their adaptive versions. Max pooling starts performing better than average right away but over time average catches up. The adaptive pooling layers experience a similar trend with adaptive max starting ahead and around the seventh epoch adaptive average has very similar results.

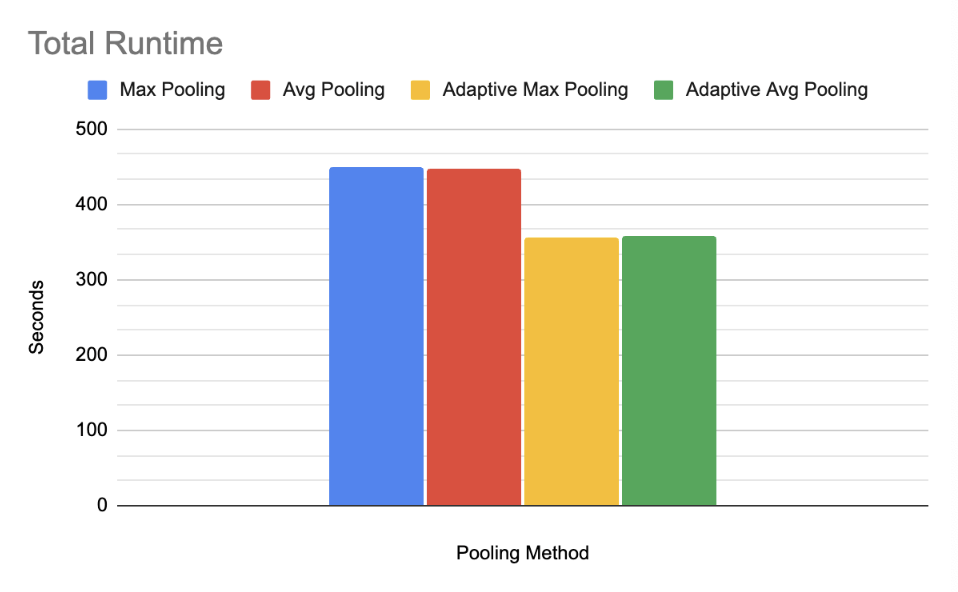
Figure 3.4: CIFAR10 Runtime

Figure 3.4 shows the runtimes of the pooling layers across CIFAR10. Max and average pooling takes around 100 seconds longer than the adaptive layers. Max and average pooling times differ by less than four seconds and the adaptive pooling layers differ by less than 2 seconds.

3.3 FER2013 Dataset

The FER2013 dataset is comprised of 32,298 48x48 pixel images. The dataset has 6 classes, each for a different emotion represented as distinct integers. The goal for the CNN model is to process the images of human faces and determine their emotion.

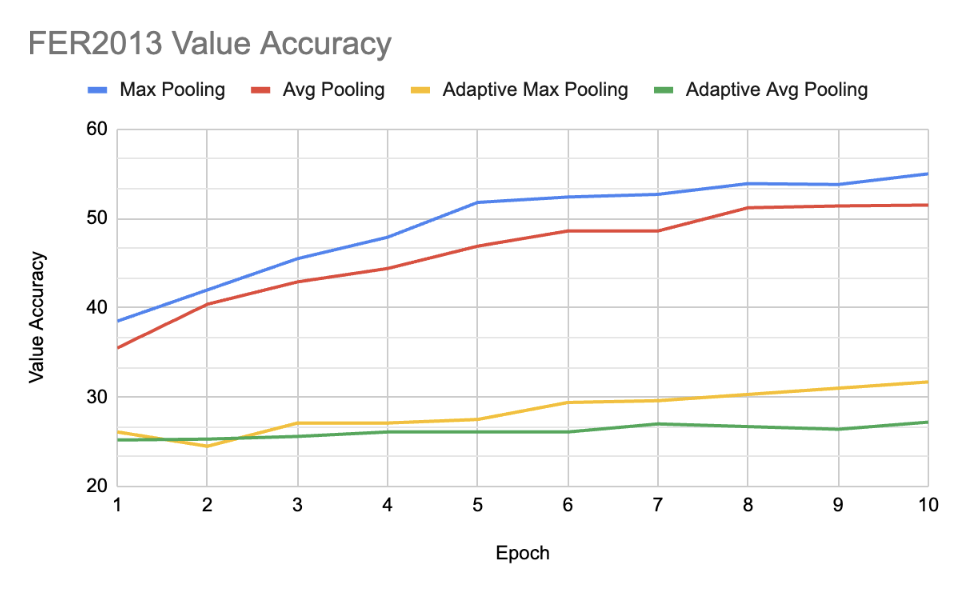
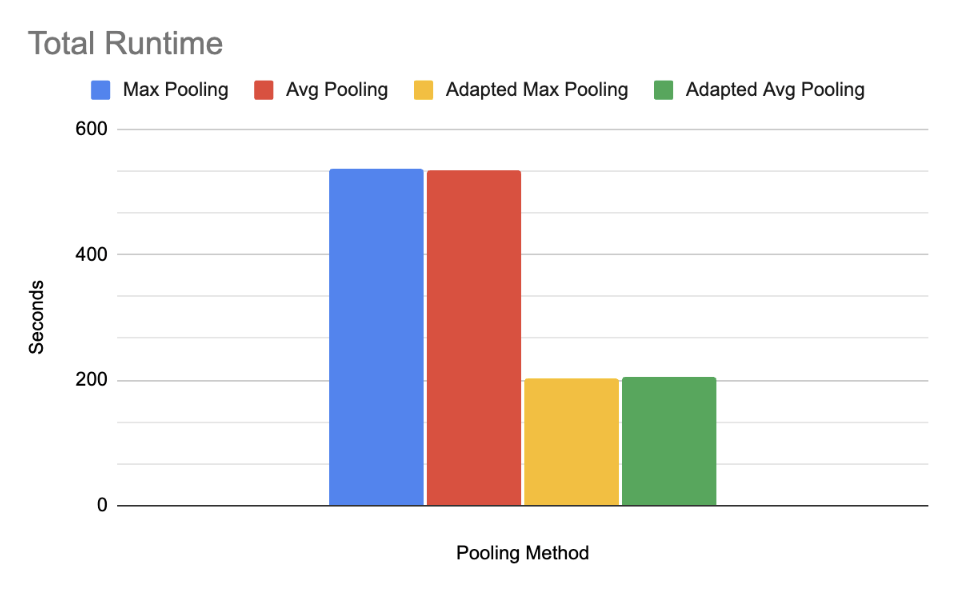
Figure 3.5: FER2013 Value Accuracy

Figure 3.5 describes a different trend from the other datasets. Max pooling starts ahead and performs better than average pooling for all the epochs. Adaptive max pooling also outperforms adaptive average pooling. There is the largest difference in any of the datasets at the end of the ten epochs between max and average pooling versus their adaptive counterparts.

Figure 3.6: FER2013 Runtime

Above figure 3.6 is a graph of the runtimes for the FER2013 dataset. Again, max and average pooling have much longer runtimes than the adaptive layers. The difference between non-adaptive and adaptive layers is the most pronounced in all the datasets examined, with over a 300 second performance difference.

4. DISCUSSION

Through the experimentation of different datasets, there were some common patterns that occurred. To note, max pooling and average pooling had much higher value accuracy than the adaptive pooling layers across all datasets. Inversely, the adaptive max and adaptive average pooling layers had a much quicker runtime to reach that performance than the standard max and average pooling. This shows that between the two types of layers, depending on requirements of the project, such as favoring runtime versus accuracy, different pooling layers can accomplish different goals.

When comparing max and average pooling layers, initially max pooling performs at a higher accuracy than average. After ten epochs of training, typically average pooling will catch up in accuracy to become negligible to max, save for the FER2013 dataset.

Regarding adaptive max and adaptive average, the same trend with the max counterpart initially performing better remains. The gap between the two is also more dignified. However, after ten epochs average pooling’s performance cannot be as easily described. In MNIST adaptive average pooling overtook adaptive maximum pooling and did so early on after only three epochs. In CIFAR10, accuracy between adaptive average and adaptive max became indistinguishable after seven epochs. In FER2013, adaptive average started at the same accuracy as adaptive max, however only increased slightly while adaptive max took higher accuracies.

Interestingly, for FER2013, the adaptive pooling techniques perform no better than a simple guess. This and the quicker runtimes yield a potential conclusion. Adaptive pooling does a poor job on some datasets of accurately capturing important features. This could be because of excessive down sampling, or just poor outputs from the input kernels. These techniques, however, do perform significantly better on different datasets. Adaptive pooling might require additional parameter optimization, so for more general tasks, the basic max pool and adaptive pool perform better.

To conclude, this data demonstrates that depending on the requirements of the CNN, there are different pooling layers for the job. If accuracy is of highest concern, using max pooling layers is the best way to go about it, with average being an acceptable substitute. If runtime is of highest concern, the adaptive pooling layers are the best tool for the job. According to our results, with more simple data adaptive average gets better results and with more complex data adaptive max performs better.

**5. RELATED WORKS**

Several papers explore similar things, but an interesting one within the medical realm explores the efficacy of different pooling techniques on various datasets containing images of eye ailments. [8] The paper, *Pooling in Convolutional Neural Networks for Medical Image Analysis* reveals a similar conclusion; different datasets tend to require different pooling methods, and finding the ideal technique is simple trial and error. The authors assert that a linear combination of average and max pooling tends to perform well, but not the best. However, further works could explore the coefficients of this linear combination, say x and y, where and are average and max pooling results, respectively. The medical paper did not mention specifically how their combination was determined, but an artificial intelligence technique could be used to determine these.

REFERENCES

[1] Saha, S. (2022) *A comprehensive guide to Convolutional Neural Networks - the eli5 way*, *Medium*. Available at: https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53 (Accessed: 21 February 2024).

[2] Mishra, M. (2020) *Convolutional Neural Networks, explained*, *Medium*. Available at: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939 (Accessed: 30 April 2024).

[3] *Dropout* (no date) *Dropout - PyTorch 2.3 documentation*. Available at: https://pytorch.org/docs/stable/generated/torch.nn.Dropout.html (Accessed: 30 April 2024).

[4] *Datasets & dataloaders* (no date) *Datasets & DataLoaders - PyTorch Tutorials 2.3.0+cu121 documentation*. Available at: https://pytorch.org/tutorials/beginner/basics/data\_tutorial.html (Accessed: 30 April 2024).

[5] Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., & Tang, P. T. P. (2017, February 9). *On large-batch training for Deep Learning: Generalization gap and sharp minima*. arXiv.org. https://arxiv.org/abs/1609.04836

[6] Krizhevsky, A. (n.d.-b). CIFAR-10 and CIFAR-100 datasets. https://www.cs.toronto.edu/~kriz/cifar.html

[7] *Training with pytorch*. Training with PyTorch - PyTorch Tutorials 2.3.0+cu121 documentation. (n.d.). https://pytorch.org/tutorials/beginner/introyt/trainingyt.html

[8] Nirthika, R., Manivannan, S., Ramanan, A., & Wang, R. (2022, February 1). Pooling in Convolutional Neural Networks for Medical Image Analysis: A Survey and an empirical study - neural computing and applications. SpringerLink. https://link.springer.com/article/10.1007/s00521-022-06953-8#Tab6