Image Processing – Comparison of APNet with classic and modern CNN architectures

*Note: Sub-titles are not captured in Xplore and should not be used

Adithya Sundararajan Iyer (A)
Department of Computer Science
University of Texas at Dallas
Richardson, TX, United States
asi200000@utdallas.edu

Parvinder Singh (P)
Department of Computer Science
University of Texas at Dallas
Richardson, TX, United States
pxs200113@utdallas.edu

***Abstract*—This paper discusses our attempt at creating a new CNN architecture for the purpose of image processing, and comparing its performance with classic and modern architectures on different datasets such as MNIST, Fashion-MNIST and CIFAR100.**

***Keywords*—CNN, image processing, principal component analysis, convolution, pooling, APNet**

I. Introduction (A:50%, B:50%)

Computer technology used in image processing enables us to process, examine, and extract information from images. Analog and digital image processing are the two main techniques for processing images. Hard copies like scanned photos and printouts are subjected to the analog process, and the results are often images. In contrast, digital image processing involves running photos via an algorithm on a digital computer. [1] The outputs here are typically information related to that image, such as data on features, attributes, bounding boxes, or masks. It is used to manipulate digital images using computers. With the help of machine learning and deep learning, image processing techniques get a significant boost in outcome. A sort of artificial neural network (ANN) known as a convolutional neural network (CNN) is used in deep learning and is most frequently used to evaluate visual imagery and perform general image processing. [2]

Convolutional neural networks (CNNs) are currently being widely used in a variety of critical domains, including text processing and computer vision, in both academia and industry. Nearly all CNN architectures seem to adhere to the same fundamental design ideas, which include applying convolutional layers to the input in stages while varying the amount of feature maps and spatial dimensions down-sampled. Modern network designs investigate novel and creative ways to construct convolutional layers in a way that enables more effective learning, as opposed to classic network architectures, which were simply made up of stacked convolutional layers. The foundation of almost all of these architectures is a repeatable unit that is applied over the whole network.

In this work, we have carefully studied and implemented some classic and modern network architectures in order to understand their working as well as motivation. In our observation, the classic models tend to train fast and keep a relatively simple pipeline. The general idea was to increase layers, or “go deeper” in order to achieve better performance. Modern CNN algorithms take a different approach. They involve using one of two approaches. One is the inclusion of residual layers to solve the degradation problem that arises from the increase in the depth of the neural network. The other involves aggregating the outcomes of parallelly performing convolutions at different scales. We attempt to create an architecture that derives from the best characteristics of both schools of thought. Our original architecture implements the modern ideology but at a smaller scale hence retaining the simplicity and speed of the classic models.

II. Related Work (A:60%, B:40%)

In order to recognize handwritten digits for zip code recognition in the postal service, Yann Lecun created the LeNet-5 model in 1998. The convolutional neural network as we know it now was mostly developed using this innovative architecture. Convolutional layers minimize computation and force a breach in the network's symmetry by using a fraction of the channels from the previous layer for each filter. Average pooling is a technique used by the subsampling layers. [3]

Alex Krizhevsky et al created AlexNet in 2012 to participate in the ImageNet competition. Even though this model is much larger than LeNet-5, the overall architecture is relatively similar. Many members of the computer vision community were persuaded to seriously consider deep learning for computer vision applications as a result of the success of this model, which won the 2012 ImageNet competition. [4]

The convolutional structures outlined above have a deeper yet simpler variant, the VGG network, which was introduced in 2014.This model was thought to be quite deep when it was first introduced. [5] The Inception network, developed by Google researchers, won first place in the 2014 ImageNet competition for classification and detection challenges. [6][7]

III. Exploratory Data Analysis (A:40%, B:60%)

We have chosen three common datasets: MNIST [8][9], Fashion-MNIST [10][11] and CIFAR-100 [12][13]. We will do exploratory data analysis on these datasets to identify the type of data, the distribution of data across all labels, and use the information obtained for data pre-processing.

A. Data Distribution

The first step in Data Exploration is to load the data, and to examine train-test split in the dataset and dimensions of the data. Then we generate a histogram plot of the data distribution across all the labels in the dataset. Once we make sure that the data is approximately evenly distributed, we can assume that it is good for training since it is a balanced dataset. Fig. 1a shows the data distribution in the MNIST Dataset. Fig. 1b shows the data distribution in the Fashion-MNIST Dataset. Fig. 1c shows the data distribution in the CIFAR-100 Dataset.

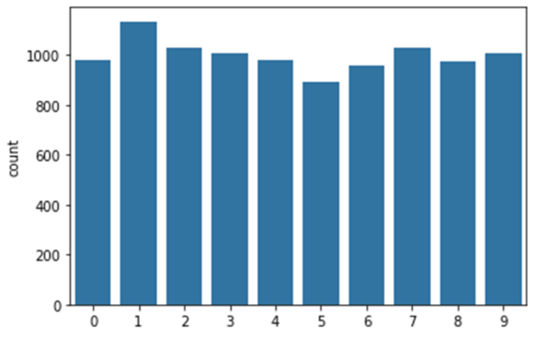

Figure 1a. MNIST Data Distribution

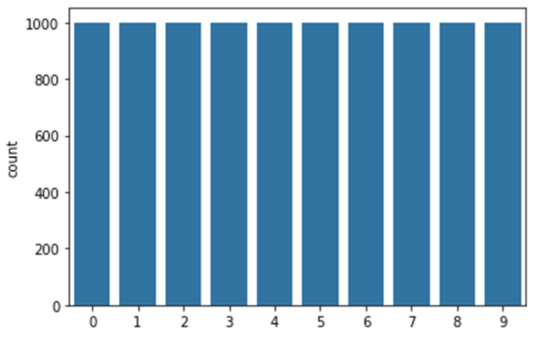

Figure 2b. Fashion-MNIST Data Distribution

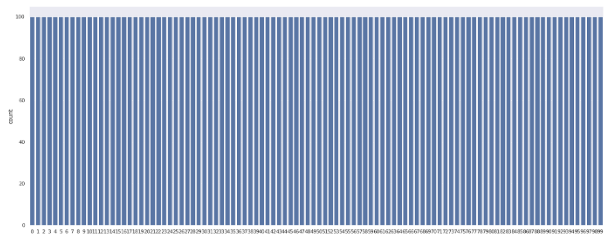

Figure 3c. CIFAR-100 Data Distribution

B. Data Visualization

Next we proceed to look at the first few training data points to get an idea of what the images in our dataset look like. This is very important considering the task is that of image processing. Visual representations always aid in better understanding of performance and also problems. Fig. 2a shows the first 16 training images in the MNIST Dataset. Fig. 2b shows the first 16 training images in the Fashion-MNIST Dataset. Fig. 2c shows the first 16 training images in the CIFAR-100 Dataset.

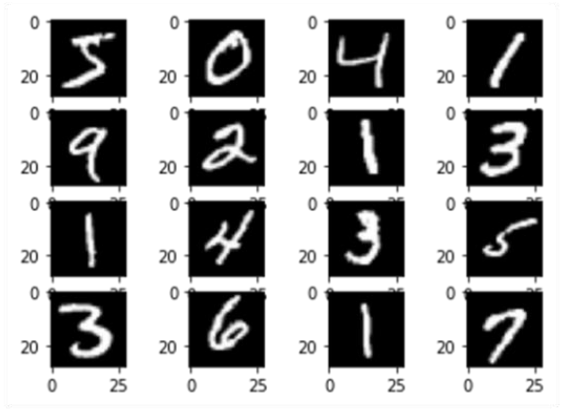

Figure 2a. MNIST Data Visualization

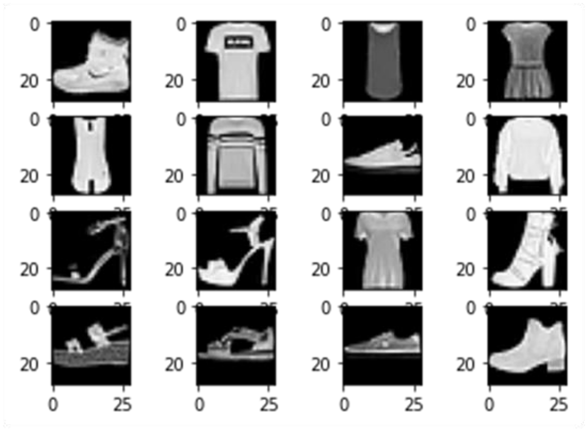

Figure 2b. Fashion-MNIST Data Visualization

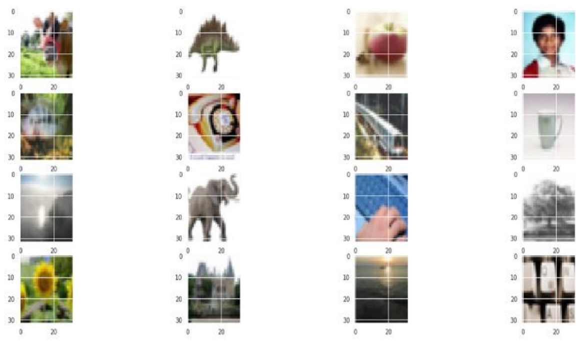

Figure 2c. CIFAR-100 Data Visualization

C. Principal Component Analysis

A common method for reducing the number of dimensions or features per observation in large datasets is principal component analysis (PCA). Principal component analysis (PCA) is a popular dimension reduction technique for analyzing large datasets containing a high number of dimensions/features per observation. The data that we pass to the built-in methods to perform PCA using the Scikit-learn module works best, in our experience, with regularized data [15]. Hence, we perform min-max normalization after checking the range of the training data. Then we create a data frame, which we can easily use to perform the principal component analysis on.

Note that we are performing PCA in 2 dimensions, and thus the results of this analysis will indicate the amount of variance that is explainable for each of the two principal components. Table 1 displays the values obtained from the 2D Principal Component Analysis of all three datasets used. Fig. 3a shows the scatter plot from the Principal Component Analysis in 2D of the MNIST Dataset. Fig. 3b shows the scatter plot from the Principal Component Analysis in 2D of the Fashion-MNIST Dataset. Fig. 3c shows the scatter plot from the Principal Component Analysis in 2D of the CIFAR-100 Dataset.

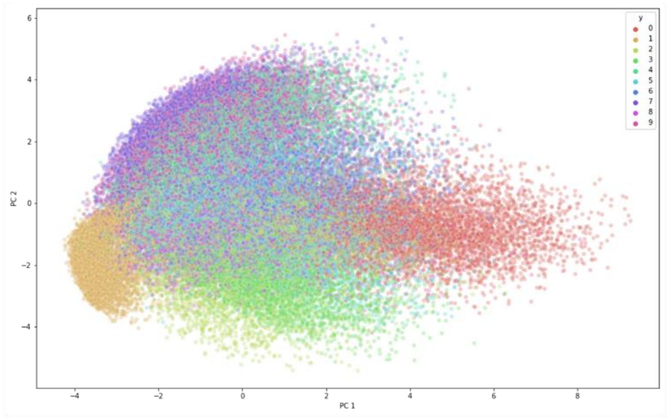

Figure 3a. MNIST PCA

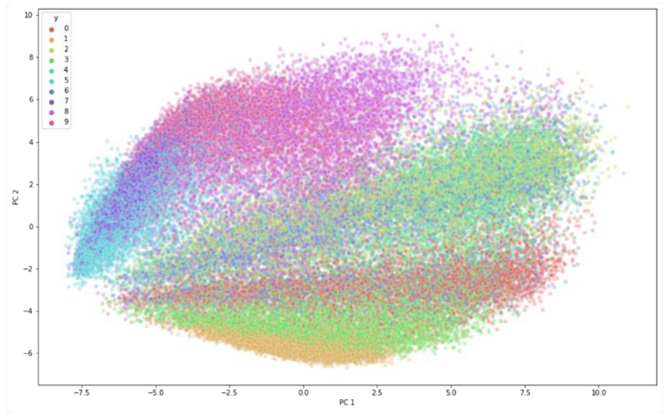

Figure 3b. Fashion-MNIST PCA

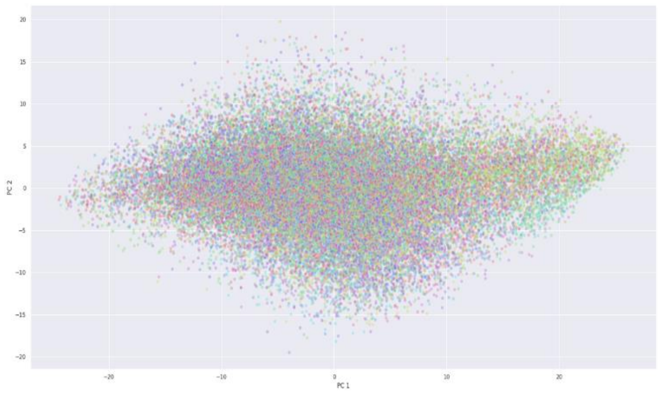

Figure 3c. CIFAR-100 PCA

D. Data Pre-processing

- After performing an exploratory data analysis on each dataset and gaining an understanding of what type of image data we're dealing with, we now need to pre-process the loaded data. First we reshape our training and/or testing data in order to fit the input formats for our models.

- Now each pixel in our data item counts as a feature to be observed. Like any machine learning experiment, we know that not every feature has the same magnitude of values, and the range of the values of this feature does not determine its level of contribution to the end result. In order to tackle this potential issue, we perform normalization using min-max scaling.

- We have seen that the training and testing labels are integer-encoded. But this is categorical, and there is no direct relation with the magnitude of the labels assigned to the objects represented by the dataset and the actual value of the data items. Thus, we do one-hot encoding for all the numeric labels in our dataset.

- Now that all our data is in the right format for training our CNN models, we divide the training data into the training and the validation datasets in a 90-10 ratio.

IV. Architecture (A:60%, B:40%)

We now proceed to understand the architectures that we are working with so that we can implement them using Keras, the high-level API that runs on top of TensorFlow, which is an open-source end-to-end machine learning platform. [14] In the following subsections, we have explained our understanding of the popular CNN architectures, on the basis of which we have implemented the deep learning models.

A. LeNet-5

In the 1990s, Yann LeCun, Leon Bottou, Yosuha Bengio, and Patrick Haffner proposed the LeNet-5 neural network design for character recognition in both handwriting and machine printing. The architecture is uncomplicated and easy to comprehend.

Two sets of convolutional and average pooling layers make up the LeNet-5 architecture, which is then followed by a flattening convolutional layer, two fully-connected layers, and a

SoftMax classifier. First convolutional layer with six feature maps or filters of size 5x5 and a stride of one is applied to the input image. Next, the LeNet-5 adds a subsampling or average pooling layer with a stride of two and a filter size of 2x2. A second convolutional layer with 16 feature maps of size 5x5 and a stride of 1 is then present. Only 10 of the 16 feature maps in this layer are linked to the six feature maps in the layer below, as can be seen in the illustration in Fig. 4.

The primary motivation is to disrupt the network's symmetry, which limits the number of connections to a manageable level. With a filter size of 2x2 and a stride of 2, the fourth layer is once more an average pooling layer. A fully linked convolutional layer with 120 1x1 feature mappings makes up the fifth layer. A fully connected layer with 84 units makes up the sixth layer. The SoftMax output layer, which is fully linked, has 10 potential values that correspond to the digits 0 to 9.

LeNet-5 uses the 'tanh' activation in each of its layers except the output. Fig. 4 shows the LeNet-5 architecture as given in the original publication [3].

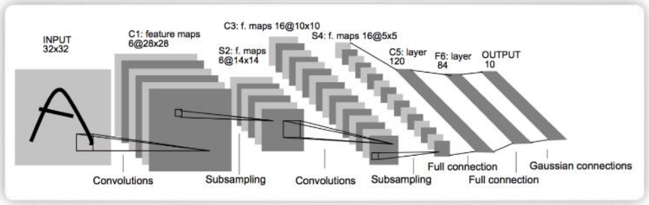

Figure 4. LeNet-5 Architecture [3]

B. AlexNet

There are eight learnable layers in the AlexNet. ReLU activation is used in each of the model's five layers, with the exception of the output layer, which obviously employs the SoftMax function. The model has five layers, each of which combines max pooling with three fully connected layers.

According to research by Alex Krizhevsky and colleagues, utilizing the ReLU as an activation function increased training speed by approximately six times. Deep convolutional networks based on ReLU can be trained much more quickly than those based on tanh and sigmoid. Additionally, they made use of the dropout layers to keep their model from overfitting. Another idea that is frequently mentioned is dropout, which can successfully stop neural networks from being overfit. In contrast to the general linear model, the model is kept from overfitting using a regular technique.

Dropout is implemented in the neural network by changing the neural network's internal structure. Update the parameters in accordance with the neural network's learning process after randomly deleting certain neurons from a given layer of neurons with a specified probability while maintaining the identities of the input layer and output layer neurons. Continue randomly removing neurons from the dropout layer in the following iteration until the training is complete.

Fig. 5 shows the AlexNet architecture followed in this experiment [4][16].

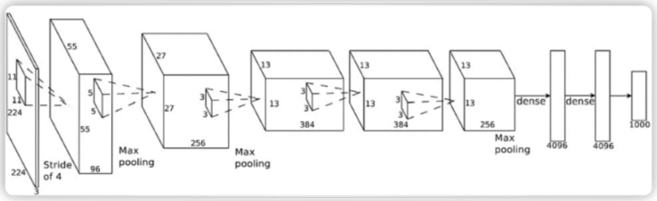

Figure 5. AlexNet Architecture [4][16]

C. VGG-16

The full name of VGG is the Visual Geometry Group, which belongs to the Department of Science and Engineering of Oxford University. It has released a series of convolutional network models beginning with VGG, which can be applied to face recognition and image classification.

The VGG architecture can range from VGG-11 to VGG-19, depending on the number of convolution layers stacked up. Custom implementations could even have more layers depending on the use-case. However, it is uncommon to find lesser layers, as this simply defeats the purpose of VGG being a deep network. A VGG model always has 3 dense (fully connected) layers, and the remaining layers are split into five sets of convolution layer stacks. An important point to note here is that in this network, every convolution layer does not necessarily have a pooling layer after it. Five pooling layers are distributed across the convolution layers.

Fig. 6 shows the VGG-16 architecture diagram used as reference for implementation [5][17].

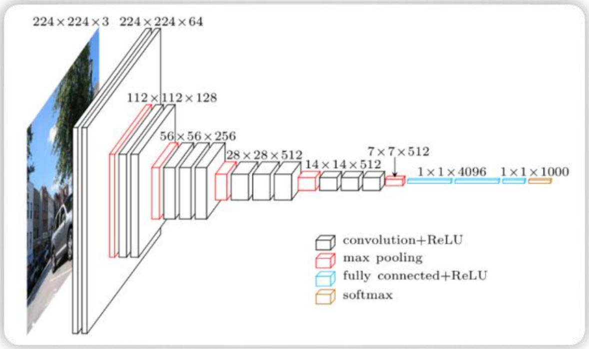

Figure 6. VGG-16 Architecture [5][17]

D. GoogLeNet (Inception)

GoogLeNet (Inception v1) was introduced by researchers at Google collaborating with various universities in 2014 [f]. It outperformed both AlexNet as well as VGG at the ILSVRC 2014 image classification challenge. This design uses methods like global average pooling and 1-1 convolutions in the middle of the architecture to generate deeper architecture.

The fully connected layers are employed at the network's conclusion in the earlier architecture, like AlexNet. The majority of parameters in many architectures that raise computing cost are found in these completely connected levels. The GoogLeNet architecture uses a technique called global average pooling, which reduces the number of trainable parameters to 0 at the network's end.

In the Inception module, three convolution layers (of kernel sizes 1, 3, 5) and a max-pooling layer are applied on the input parallelly. These outputs are then concatenated or stacked together to give the inception module's final output. This ensures better handling of objects at multiple scales. We have implemented the naive version of the inception module in our experiments instead of using dimension reduction due to computational limitations.

Additionally, inception architecture makes use of auxiliary classifier branches in the intermediate stages of training. They do not operate during testing, but simply help improve performance during training. The two auxiliary units compute loss midway while training, and contribute to the final loss calculation with a 30% weightage. The auxiliary layers not only provide regularization but also help in combating the problem of vanishing gradient which can happen due to the deep nature of the network.

Fig. 7 shows the pictorial representation of the GoogLeNet architecture used for reference for this experiment [6]. Since the model executed in this work has been modified to not employ dimension reduction in the inception modules, it would show a difference from the figure.

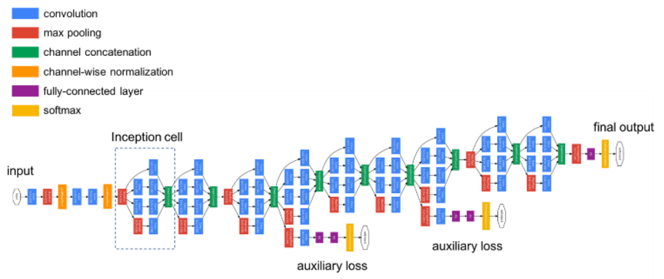

Figure 7. GoogLeNet Architecture [6]

E. *APNet*

Inspired by the simplicity and speed of training architectures such as LeNet-5, and by the filter concatenation strategy and channel-wise normalization of the Inception architecture, we have experimented with a custom architecture of our own.

First the input is passed to a convolution layer with 32 filters and a kernel size of 7. Then the output of this layer is max-pooled and batch normalization is performed on it. The next layer is again a convolution layer, but with 64 filters and a kernel size of 11. The output of this first goes through batch normalization before the max pooling layer. Now the output of the second max pooling layer is passed to two convolution layers parallelly which are then concatenated before passing through a max pooling layer again. The parallel convolution layers have kernel sizes of 3 and 5. The output of the final pooling layer is flattened onto a fully-connected layer of size 1024. This FC layer implements a dropout of 60% before passing through to the dense output layer with the SoftMax activation and a size equal to the number of classes. Every other layer uses the ReLU activation function.

Fig. 8 shows the outline of the architecture of the current APNet model.

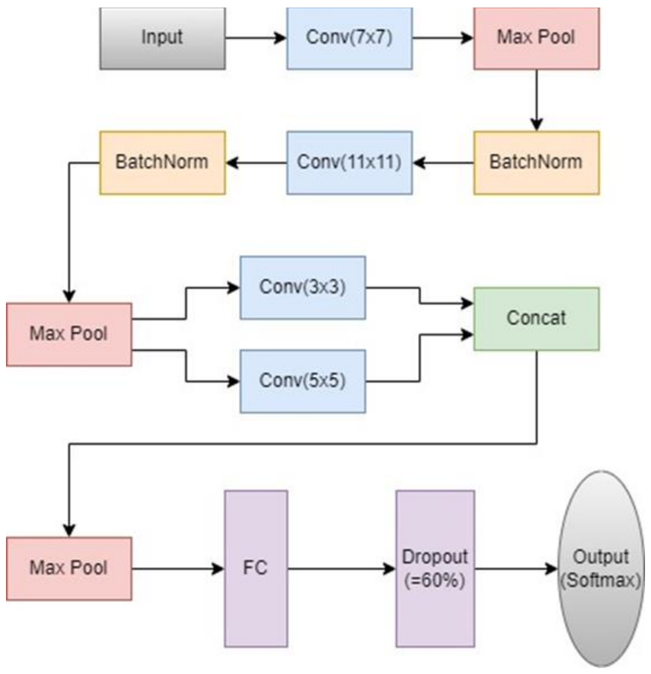

Figure 8. APNet Architecture

V. Code Implementation (A:45%, B:55%)

Given below is only a count of the parameters for each of the architectures. For a brief overview on the code implementation, please refer to the slide deck. For a more detailed explanation of the code implementation, please refer to the code files attached.

A. *LeNet-5*

Total params: 545,546
Trainable params: 545,546
Non-trainable params: 0

B. *AlexNet*

Total params: 25,666,370
Trainable params: 25,665,666
Non-trainable params: 704

C. *VGG-16*

Total params: 33,637,066
Trainable params: 33,637,066
Non-trainable params: 0

D. *GoogLeNet (Inception)*

Total params: 113,777,486
Trainable params: 113,776,974
Non-trainable params: 512

E. *APNet*

Total params: 331,882
Trainable params: 331,690
Non-trainable params: 192

VI. Training Cycle (A:50%, B:50%)

We tune the various hyperparameters for our models and finally run each one on all the datasets. This helps us to compare the performance and the working behind each architecture. We analyze our findings to then create an architecture of our own which we termed APNet. The aforementioned procedure is described in further detail in the subsections that follow.

The process of hyperparameter tuning for neural network models required a lot of time and effort. The entire model needs to run as though it was training as usual, and this process repeats for every value of the hyperparameter that we are trying to tune. Considering there are multiple hyperparameters to tune for our current experiment, this would demand each model to run on the entire dataset around a dozen times each. We simply do not have the resources to execute over 50 runs of a deep learning model on a dataset with at least 50,000 data points. Thus we choose 25% of our training data for the sake of hyperparameter tuning. This reduces our computational load overall by roughly 50% (note that this is not 75% because we still would be running the finalized model, i.e. after hyperparameter tuning, on the entire dataset).

A. *Hyperparameter Tuning – number of epochs*

An epoch is defined as the model going through the entire dataset one time. A neural network model needs to go through the entire training data multiple times in order to learn better. Usually, the deeper the model is, the lesser the number of epochs needed for the model to converge.

We run each model on the given dataset for up to 50 epochs, and in some cases 100 epochs. After a certain range of epochs, the validation accuracy for every model has peaked and starts to stagnate, or in some rare cases, drop. However, the training accuracy continues to show a growth even after that point. This suggests that beyond those many number of epochs, the model begins to overfit the training data to an extent where it is unable to generalize the entire dataset. This is how we pick the cut-off number for the particular architecture.

B. *Hyperparameter Tuning – learning rate*

The learning rate is what determines the step size at each iteration while moving towards the loss function minima. If the learning rate is too low, the model would need too many iterations to reach the minima, and even if it does, it might get stuck in a local minima rather than the global value. If the learning rate is too high, the model might never converge and the loss function never optimizes, giving very unstable values no matter how many epochs the model goes through.

Every model has a different value of learning rate that is ideal for it, and so we go through the range of values = [0.001, 0.01, 0.1] for each architecture. We could observe that the model gives a stable increase in performance with a particular learning rate, or maybe the model could converge faster with some other learning rate value. By noticing trends in loss and accuracy, we determine the best learning rate for our use-case.

C. *Hyperparameter Tuning – momentum*

Momentum is a hyperparameter that assists in the stochastic gradient descent algorithm. It functions similar to an adaptive learning rate algorithm, but instead of changing the learning rate with time/epochs, it adds onto the parameter update apart from the update that comes from the learning rate step. It can accelerate the model into getting out of a local minima, and it can dampen the model to keep it inside the global minima.

We use momentum values of = [0.6, 0.75, 0.9] for tuning. Some models show considerable changes with different momentum factors, while others do not. Finding the value which yields a good combination of high validation accuracy and low validation loss is a good way to determine the best momentum factor for the model.

D. *Finalized Models*

Every architecture is implemented using the Keras Layers module [17], and uses the SGD optimizer [18]. The hyperparameters fixed for each architecture after tuning are given in Table 1.

Model/HP	**Epochs**	**Learning rate**	**Momentum**
LeNet-5	20	0.1	0.75
AlexNet	15	0.01	0.9
VGG-16	25	0.01	0.9
GoogLeNet	20	0.001	0.9
APNet	40/60	0.001	0.75

Table 1. Finalized Hyperparameters

VII. Results and Analysis (A:55%, B:45%)

The results of running every architecture on each of the datasets have been tabulated. The training accuracy, validation accuracy, and testing accuracy have been reported for comparison.

The graphs shown in the figures represent the model performance during training time. We can see the variation of training/validation losses through the epochs to determine stability and how closely the model is fitting to the optimum function. We can observe the trends in training/validation accuracies to know if our model is on track or if it is overfitting the train data.

Figure 9a, 9b, 9c show the training loss and accuracy graphs for LeNet-5 Architecture on MNIST, Fashion-MNIST, and CIFAR-100 datasets respectively.

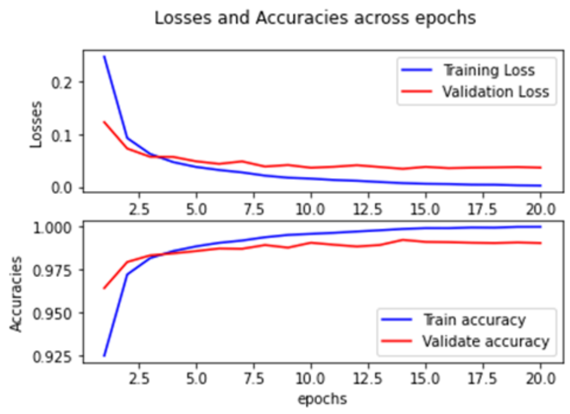

Figure 9a. LeNet5-MNIST

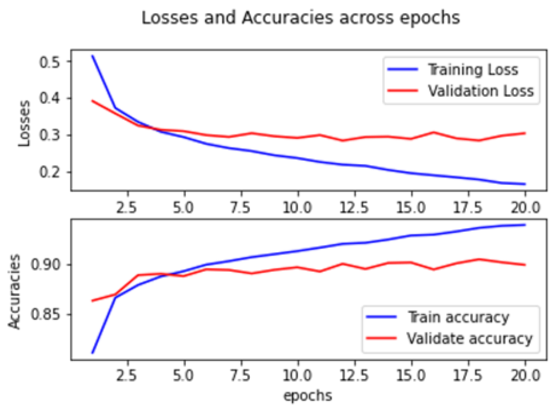

Figure 9b. LeNet5-Fashion-MNIST

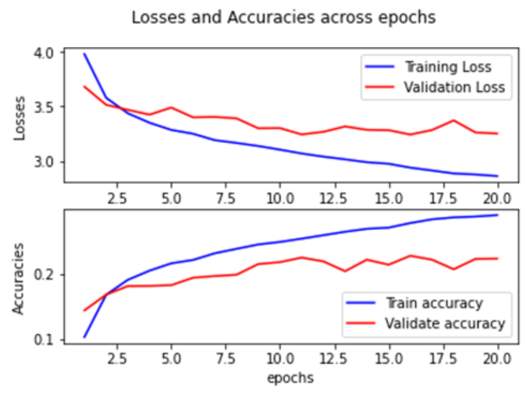

Figure 9c. LeNet5-CIFAR-100

Figure 10a, 10b, 10c show the training loss and accuracy graphs for AlexNet Architecture on MNIST, Fashion-MNIST, and CIFAR-100 datasets respectively.

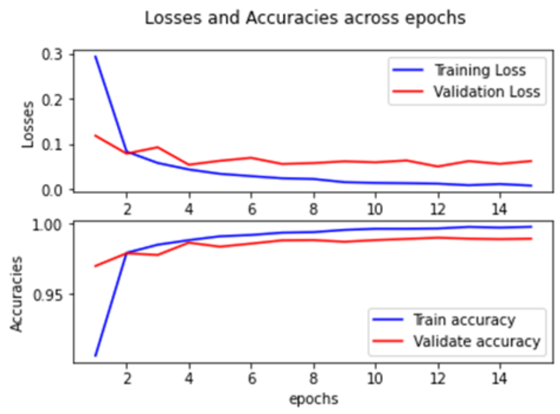

Figure 10a. AlexNet -MNIST

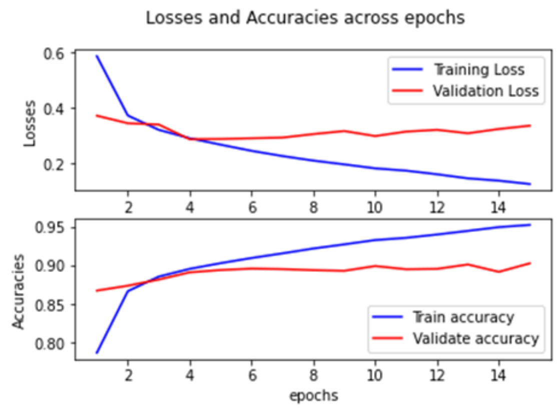

Figure 10b. AlexNet -Fashion-MNIST

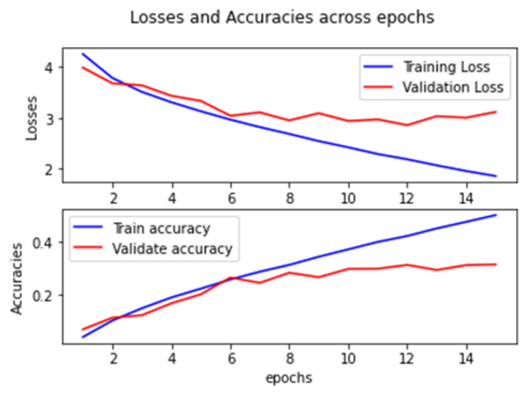

Figure 10c. AlexNet -CIFAR-100

Figure 11a, 11b, 11c show the training loss and accuracy graphs for VGG-16 Architecture on MNIST, Fashion-MNIST, and CIFAR-100 datasets respectively.

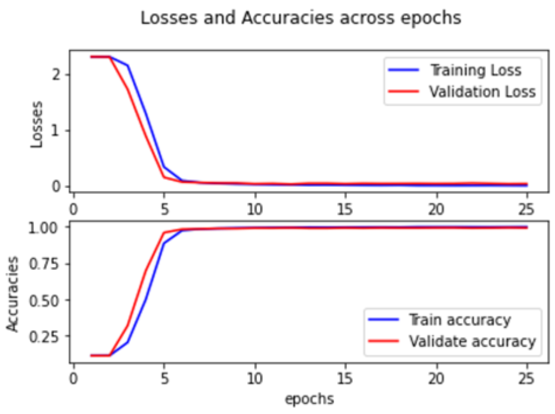

Figure 11a. VGG-16-MNIST

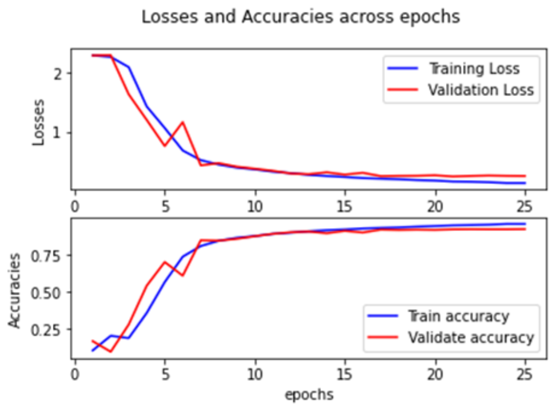

Figure 11b. VGG-16-Fashion-MNIST

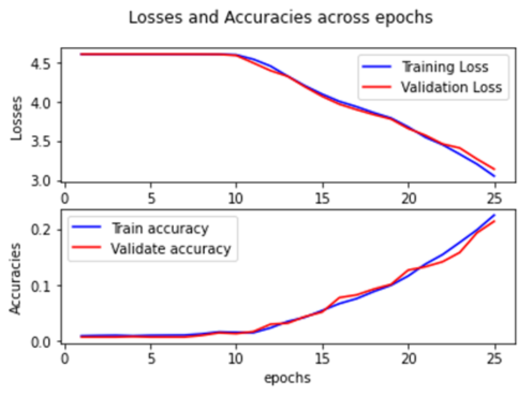

Figure 11c. VGG-16-CIFAR-100

Figure 12a, 12b, 12c show the training loss and accuracy graphs for GoogLeNet Architecture on MNIST, Fashion-MNIST, and CIFAR-100 datasets respectively.

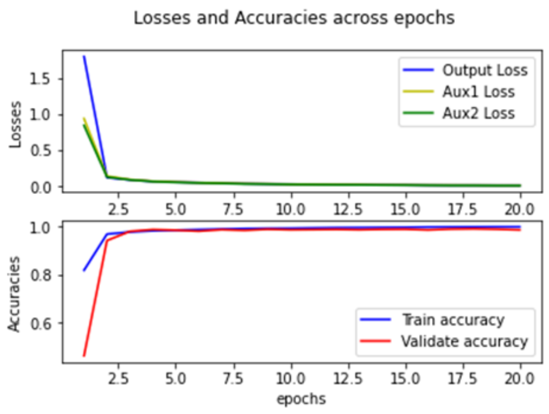

Figure 12a. GoogLeNet -MNIST

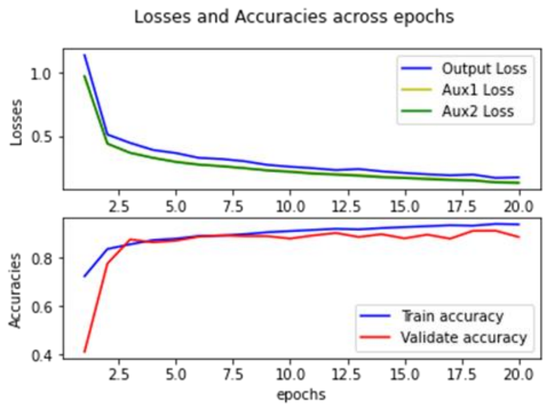

Figure 12b. GoogLeNet -Fashion-MNIST

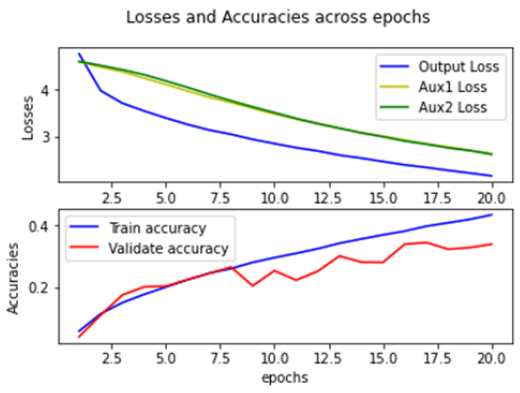

Figure 12c. GoogLeNet -CIFAR-100

Figure 13a, 13b, 13c show the training loss and accuracy graphs for APNet Architecture on MNIST, Fashion-MNIST, and CIFAR-100 datasets respectively.

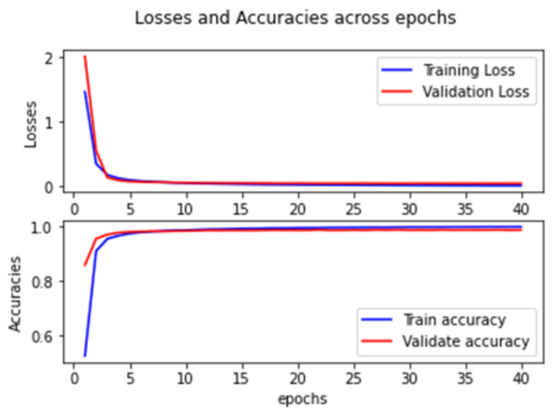

Figure 13a. APNet -MNIST

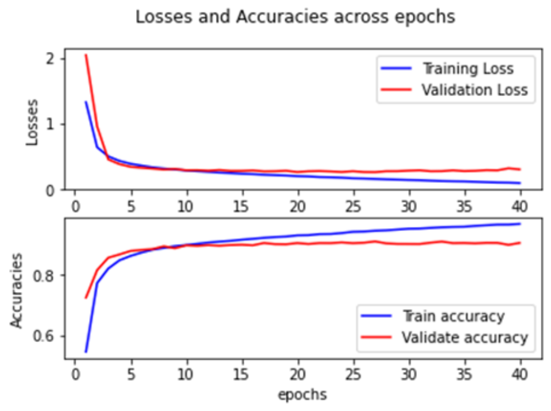

Figure 13b. APNet -Fashion-MNIST

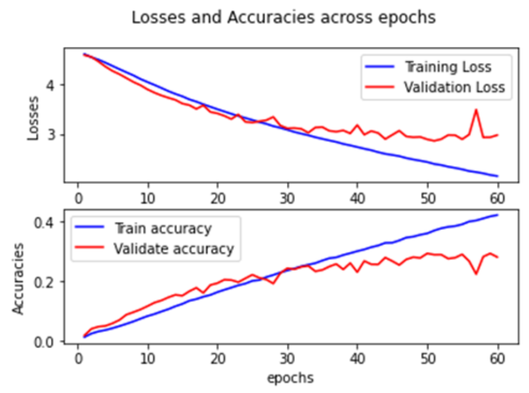

Figure 13c. APNet -CIFAR-100

Table 2 shows the comparison of the models on the MNIST Dataset. Table 3 shows the comparison of the models on the Fashion-MNIST Dataset. Table 4 shows the comparison of the models on the CIFAR-100 Dataset.

Model	**Train Accuracy**	**Validate Accuracy**	**Test Accuracy**
LeNet-5	99.97	99.03	99.03
AlexNet	99.76	98.92	98.90
VGG-16	99.92	99.23	99.36
GoogLeNet	99.81	98.53	98.77
APNet	99.93	98.78	99.09

Table 2. Comparison of performance on MNSIT dataset

Model	**Train Accuracy**	**Validate Accuracy**	**Test Accuracy**
LeNet-5	93.85	89.87	89.43
AlexNet	95.23	90.25	89.51
VGG-16	95.51	92.03	92.16
GoogLeNet	93.63	88.45	87.80
APNet	96.97	90.67	89.39

Table 3. Comparison of performance on Fashion-MNSIT dataset

Model	**Train Accuracy**	**Validate Accuracy**	**Test Accuracy**
LeNet-5	29.01	22.32	23.77
AlexNet	50.09	31.50	32.33
VGG-16	22.51	21.40	22.93
GoogLeNet	43.35	33.86	35.80
APNet	42.10	20.08	28.13

Table 4. Comparison of performance on CIFAR-100 dataset

VIII. CONCLUSION (A:50%, B:50%)

Comparing the existing architectures amongst themselves yields a few conclusions. All models perform well on the MNIST dataset, but the VGG architecture shows best results for it. VGG comes out on top again for the Fashion-MNIST dataset, clearly establishing its hype. However, the VGG-16 model seems to have the worst performance for the CIFAR-100 dataset, possibly due to the higher number of classes in it. In this arena, GoogLeNet easily takes the spot for top performer. Its comparatively better performance can be attributed to its ingenious architectural design. However, every model performs poorly on the CIFAR-100 dataset in contrast to their performance on the other 2 datasets. It makes sense because we need much more compute power and even complex architectures to be able to build a successful multi-class image classifier with 100 classes.

Our APNet architecture, along with the LeNet-5, comes very close to the performance of the VGG-16 on the MNIST dataset, overtaking the deeper models like AlexNet and GoogLeNet in this particular experiment. AlexNet, LeNet-5 and APNet are closely tied at second place after VGG's performance on the Fashion-MNIST dataset. APNet also beats LeNet-5 and VGG-16 on the CIFAR-100 dataset. We can see that APNet takes more epochs than the other architectures to converge on the minima, but given that its training is incredibly fast, this shouldn't be an issue. The dropout layer helps additionally with APNet not overfitting the training data. The performance of APNet is consistently good on both MNIST and Fashion-MNIST datasets.

IX. FUTURE SCOPE OF WORK (A:50%, B:50%)

APNet is a very lightweight architecture, with lots of possibilities to make it deeper. The empirical results already look promising on the MNIST and Fashion-MNIST datasets. There is scope to add more convolution layers. We could implement dimension reduction prior to the concatenation stage. Adding an additional fully-connected layer and splitting the dropout ratio between the two layers could help further increase performance. Future research into its architecture and behavior can lead to a stable deep CNN architecture and possibly eventually outperform some other well-known networks.

ACKNOWLEDGMENT

We thank Prof Feng Chen, our instructor for the course Artificial Intelligence (CS6364), and Kai Jiang, TA for the course. We are grateful for this learning opportunity to implement the knowledge gained into practical and real-world application. We hope our work is satisfactory and meets expectations.

REFERENCES

[1] R. C. Gonzalez and R. E. Woods, Digital image processing. New York, Ny: Pearson, 2018.

[2] M. V. Valueva, N. N. Nagornov, P. A. Lyakhov, G. V. Valuev, and N. I. Chervyakov, "Application of the residue number system to reduce hardware costs of the convolutional neural network implementation," Mathematics and Computers in Simulation, vol. 177, pp. 232–243, Nov. 2020, doi: 10.1016/j.matcom.2020.04.031.

[3] Y. Lecun, L. Bottou, Y. Bengio and P. Haffner, "Gradient-based learning applied to document recognition," in Proceedings of the IEEE, vol. 86, no. 11, pp. 2278-2324, Nov. 1998, doi: 10.1109/5.726791

[4] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," Communications of the ACM, vol. 60, no. 6, pp. 84–90, May 2012, doi: 10.1145/3065386.

[5] K. Simonyan and A. Zisserman, 'Very deep convolutional networks for large-scale image recognition', arXiv preprint arXiv:1409. 1556, 2014.

[6] G. Zeng, Y. He, Z. Yu, X. Yang, R. Yang, and L. Zhang, ''Going deeper with convolutions christian', in Proc. IEEE Conf. Comput. Vis. Pattern Recognit.(CVPR), 2015, pp. 1–9.

[7] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, 'Rethinking the inception architecture for computer vision', in Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 2818–2826.

[8] "The mnist database," MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges. [Online]. Available: http://yann.lecun.com/exdb/mnist/. [Accessed: 14-Dec-2022].

[9] https://www.tensorflow.org/api_docs/python/tf/keras/datasets/mnist

[10] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," arXiv:1708.07747 [cs, stat], Sep. 2017, [Online]. Available: https://arxiv.org/abs/1708.07747

[11] https://www.tensorflow.org/api_docs/python/tf/keras/datasets/fashion_mnist

[12] A. Krizhevsky, "CIFAR-10 and CIFAR-100 datasets," Toronto.edu, 2009. https://www.cs.toronto.edu/~kriz/cifar.html

[13] https://www.tensorflow.org/api_docs/python/tf/keras/datasets/cifar100

[14] https://www.tensorflow.org/api_docs/python/tf/keras

[15] https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html

[16] A. Alekseev and A. Bobe, 'GaborNet: Gabor filters with learnable parameters in deep convolutional neural network', in 2019 International Conference on Engineering and Telecommunication (EnT), 2019, pp. 1–4.

[17] L. Madhuanand, P. Sadavarte, A. J. H. Visschedijk, H. A. C. Denier Van der Gon, I. Aben, and F. B. Osei, 'Deep convolutional neural networks for surface coal mines determination from sentinel-2 images', European Journal of Remote Sensing, vol. 54, no. 1, pp. 296–309, 2021.

[18] https://www.tensorflow.org/api_docs/python/tf/keras/layers

[19] https://www.tensorflow.org/api_docs/python/tf/keras/optimizers/experimental/SGD