University of Missouri
Saint Louis, MO

Project Report - Deep Learning (CMP_SCI_4390)

Image Classification Using CNN on American Sign Language Dataset

Harcharan Singh Kabbay
M.A. Mathematics with Data Science Emphasis

Supervised by
Dr. Badri Adhikari

Contents

Abstract

This paper discusses various phases in the life cycle of a Deep Learning model. The Deep Learning models, specifically Convolutional Neural Networks (CNN) or ConvNets, are famous for solving Image classification problems. Many documents and easy-to-use code are available on the Internet, and many books are already written on this subject. This report uses the techniques learned from the book Deep Learning with Python, Chollet, F. (2021) for model development and evaluation.

An introduction to the CNN architecture is included to set a stage for the terminology used in the document and make it easy to understand the model development process. The report also covers the approach taken to split the data set to comply with industry best practices. The document includes information on some common types of Callbacks that can help the audience to save time and compute during the model training and validation phases.

We start by building an over-fitting model and observing the impact of changes in model layers. We try techniques like Data Augmentation and Regularization to see the effect on the metrics like training and validation accuracy. The report also covers the try-outs for some advanced model architectures like ResNet and VGG16 to understand the use and power of the pre-defined models and observe the model accuracy and runtimes. Each phase contains information on what is being tested, how we try it, and observations from the testing. Wherever applicable, plots or tables are included from the observations. This report provides an easy-to-use recipe to develop models for any Image Classification.

I Introduction

The central theme of the project is to design a **Convolutional Neural Network** architecture to predict the output labels for a given image data set. Any attempt at the design will be incomplete without describing the Convolutional Neural Network.
A Convolutional Neural Network, a.k.a **CNN** is a Machine Learning algorithm that takes input images and learns (calculate weights and biases) at each hidden layer to classify the input image into an output class. A very generic format of a CNN model can be represented using Figure 1.

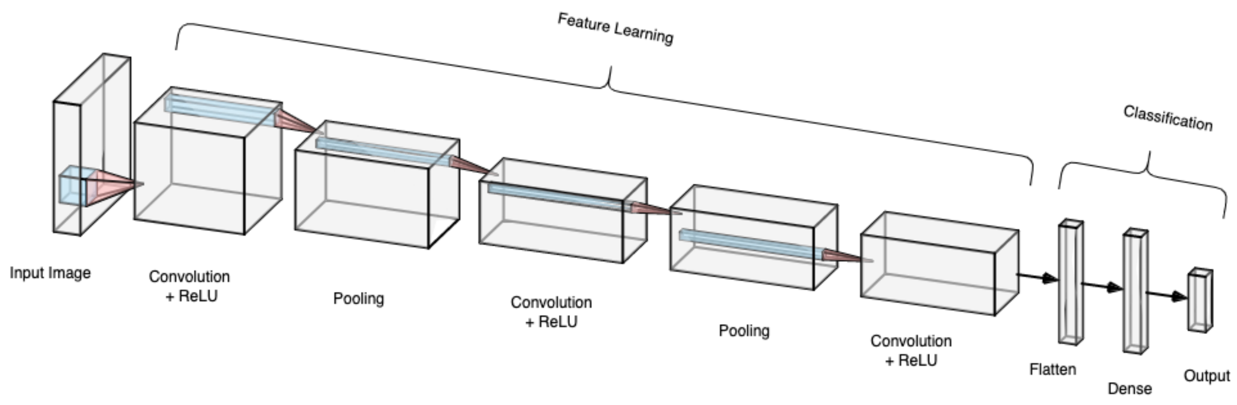


Figure 1: A Generic CNN layout

The CNN model has three parts, namely Input, Feature Learning, and Classification. A basic CNN model could be a stack of Convolution and Pooling Layers in feature learning. We will go through the details on attributes for each learning and classification layer throughout this report. This report contains different stages of Deep Learning model building and evaluation techniques.

II Data set

The data set is crucial to the success of any Deep Learning model. You can build, debug, compare, or enhance the models if you understand what the data set represents, what we expect the model to predict, and the baseline accuracy. For the sake of this project, we choose the first four characters (A, B, C, and D) of **American Sign Language** data set. We used the *sklearn.model_selection.train_test_split* function to split the data set. Development set is often the name used for a train set and validation set combined. The test data set is not used in any model development but only for testing the model. (Overview in **Figure 2**)

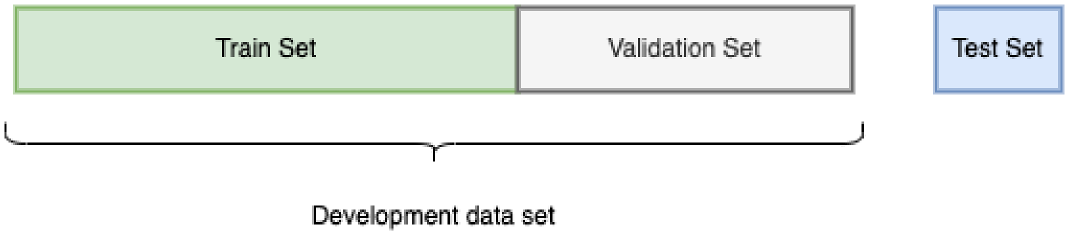


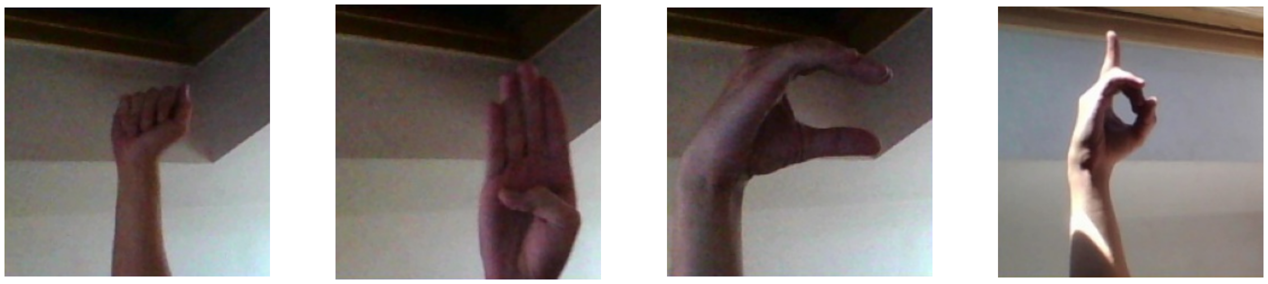
Figure 2: Data set Overview

We first split the Data into the Development set and Test Set in a ratio of 9:1. Then further broke the Development set, Train set, and Test set with a percentage of 6:4. The details on the number of image files for each class or character are given in **Table 1**.

Class	A	B	C	D
Train	1620	1620	1620	1620
Validation	1080	1080	1080	1080
Test	300	300	300	300

Table 1: Data set split

Some of the sample images are displayed in Figure 3.



(a) Sign - A (b) Sign - B (c) Sign - C (d) Sign - D

Figure 3: Sample Images from American Sign Language Dataset

III Tools for Model Development

We use Keras Deep Learning API for model development on a Google Colab environment. Keras provides an easy way to build and run models from beginners to advanced use cases. Google Colab environment allows GPUs and a high memory environment (on a paid plan) to save some compute time. Python plotting library *matplotlib* is used for plotting the metrics from the models. Tensorboard web-suite from Tensorflow is used for enhanced visualization like checking weight distributions, histograms, etc.

IV Callbacks

Model Training is a time-consuming task and overutilizes many compute resources if the process is not monitored. Babysitting and watching the model train and validation cycles is not always possible. To overcome this problem, Keras offers a variety of Callbacks. Callbacks allow us to control the runs of model training iterations. We use the following three callbacks on our models: -

1. **Early stopping**- This allows us to interrupt the model run when the validation loss is not improving. We use the parameter patience to wait for some epochs before interrupting the model *fit()*.

2. **Model checkpoint**- Model checkpoint allows saving the state of the model continually during the training phase. You can save the best model that has the best performance so far. These callbacks are typically used to optimize the compute cycles and save the best-known version of the model. Here is the code example on how to define these callbacks and to use it in the *model.fit()*

3. **LearningRateScheduler**- This Callback allows to adjust the learning rate for the epoch dynamically. We used this callback to change the learning rate as the number of epochs increases. Table 2 shows how the learning rate callback was used during the model training.

Epochs	Learning Rate
1 to 2	0.001
3 to 15	0.0001
16 -	0.00001

Table 2: LearningRateScheduler - Callback

4. **Tensorboard Callback**- This Callback enables saving the *events_file* from each model training and validation run, which can later be visualized using TensorBoard.

V Model Training

We start the model training with a base model having 12 layers

- Conv2D - 5 layers
- MaxPool2D - 4 layers
- Flatten - 1 layer
- Dense - 2 layer

and then keep on modifying the number of layers and filters. Each model was trained for ten epochs to check the learning (loss and accuracy during the training phase). Once the model is trained, additional metrics like Accuracy, Precision, Recall, and F1 score were calculated based on model testing for a sample size of 1000 from the training data. We do not use any validation or test data set during this phase of model training because we target to find an over-fitting model.

Figure 4 is a comparison plot for the performance observed from all the models. The X-axis shows the model (Model_name, Layers, and Params), and the Y-axis shows the performance metrics (Accuracy, Precision, Recall, and F1-score).

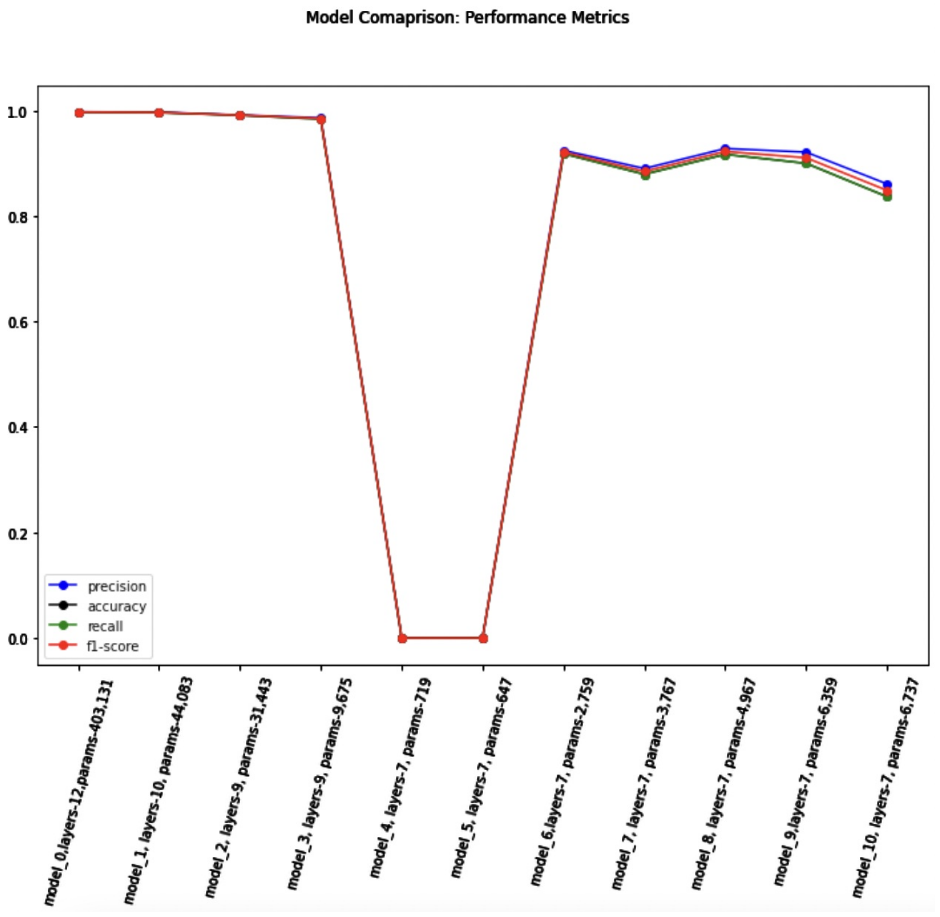


Figure 4: Model Comparison

V.I Observations

A total of 11 **CNN** models are tested to learn and predict the data from the American Sign Language data set for alphabets A, B, C, and D. Following are the observations from the tryouts:-

1. **Size and Shape of Model defines the number of parameters** - The number of parameters are controlled via input size, the number of layers, features per layer in case of Conv2D, and window size in MaxPool2D layer, neurons in Dense layers. Overall, we had a range of parameters from 719 to 403,131, with layers from 7 to 12, respectively.

2. **Bigger models not necessarily the best** - We can always add more layers and filters to increase the size of the Model. Still, you may achieve the same performance from a relatively more minor model as well. Moreover, the bigger Model may overanalyze the data and be over-fitting.

3. **Model predicts output labels if passed as additional channel** - In one of the use cases, we passed the output labels as another channel to input. The Model learned quickly by using merely seven layers and 647 parameters.

We can achieve maximum accuracy (99.76%) with model_0, which has 12 layers and uses 403,131 parameters. Model_1 can deliver a relative accuracy of 99.63% using ten layers and 44,083 parameters. Learning curves and additional metrics would show a degradation in the performance of a model if we reduced the parameters by trimming the layers or reducing the filters any further. Model_3 is the most miniature model that provides 98.3% accuracy with nine layers and 9,675 params.

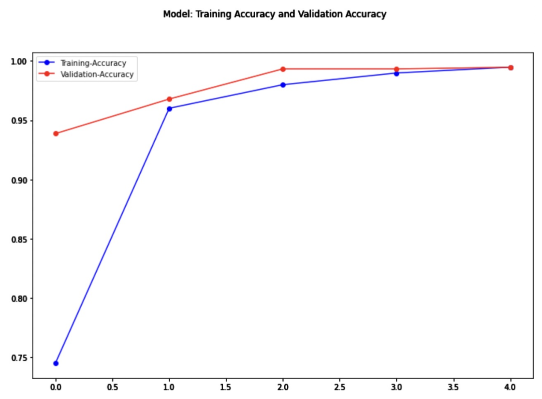
VI Data Augmentation

Now, we use the techniques of data augmentation (Image generators) to study the effect on the model's accuracy. Image generators provide an easy way to read images from memory data sets or disks and make them available for model processing. Image generators can generate data in batches with real-time augmentations like re-scale, resize, rotate, zoom, etc. These augmentations help control the problem of over-fitting by generating more training data without repeating the same image. Image generators use *ImageDataGenerator* method from *tensorflow.Keras.preprocessing.image* library.

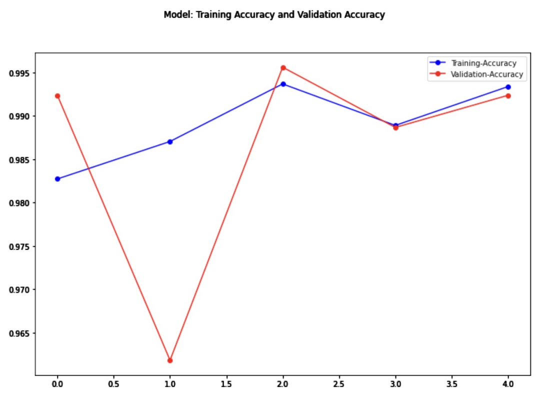
We start with a base model to test the impact of Data augmentation. We enabled different data augmentation parameters in steps and captured the impact on Training and Validation Accuracy. The details of the testing are as follow: -

1. **Base (Rescale)** - This is the base Image Data generator with just the *rescale* option, which divides each pixel by 255 to get the values in 0-1 range. The model can achieve a training accuracy of 99.48% and a validation accuracy of 99.47%. (Figure 5.a)

2. **Adding rotation_range** - this image generator has the rotation_range option to rotate images by angle. We also add the fill_mode to fill the blank space created by rotating the image. The model can achieve a training accuracy of 99.34% and a validation accuracy of 99.24%. (Figure 5.b)



(a) Using Re-scale

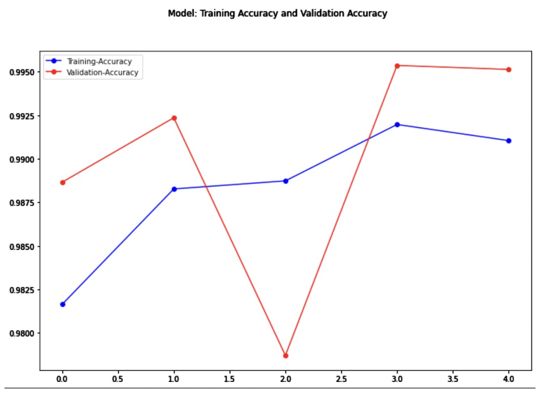


(b) Using rotation_range

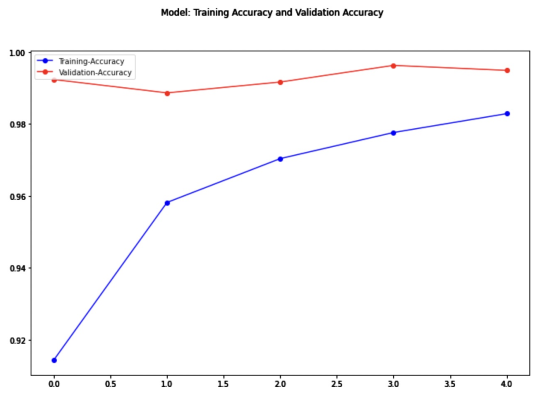
Figure 5: Impact of Data augmentation

3. **Adding width and height shift ranges** - Next, we add width and height shift ranges to the Image data generators. The model is able to achieve a training accuracy of 99.10% and a validation accuracy of 99.51%. (Figure 6.a)

4. **Adding shear, zoom and horizontal_flip**. We enhance the augmentation by adding shear, zoom and horizontal flip options to the Image data generator. The model is able to achieve a training accuracy of 98.29% and a validation accuracy of 99.48%.(Figure 6.b)



(a) Using Width and height shift ranges



(b) Using shear, zoom and horizontal_flip

Figure 6: Impact of Data augmentation 3, 4

VI.I Observations

We tried out four **Data Image Generators** to feed images for model learning. There are the following observations from these tryouts:-

1. **Ease of use** - Image Data generators are easy to use. Once defined, these can be referenced in Image generators for each train, validation, and test category and fed into the model for Image inputs. The *ImageDataGenerator* method provides a simple way to enable, disable or tune the data augmentation techniques, like rescale, flip, zoom, etc.

2. **Control over-fitting problem** - Image Data Generators ensure that the same image is not fed into the model for training, hence increasing the training data. The use of Image Data Generators helps control the problem of over-fitting. Though we didn't suffer the pain of over-fitting for the given data set and the chosen model. Accuracy was already high even with basic rescale augmentation. One noticeable difference in the last image data augmentation (maximum options used) is the validation accuracy starts high (99.24%) from the first epoch run and stays high.

VII Regularization

After testing the impact of Data augmentation, we move on to try the regularization techniques and study the effect on the accuracy of the model. Regularization provides a simplistic approach to control the problem of over-fitting by using some constraints on the weights to choose a smaller value. Some of the standard regularization methods are **BatchNormalization, Dropout, and L2 regularization**. Similar to what we followed in previous phases, we start with a base model and test the impact of regularization by enabling one parameter at a time. The metrics like accuracy, loss, and convergence rate are noted for each regularization change for the base model.

1. **BatchNormalization** - We modify the model by adding *BatchNormalization* layer after each Conv2D layer. This layer would normalize the data coming out from the Conv2D. The model is able to achieve a training accuracy of 97.75% and a validation accuracy of 98.80%. (Figure 7)

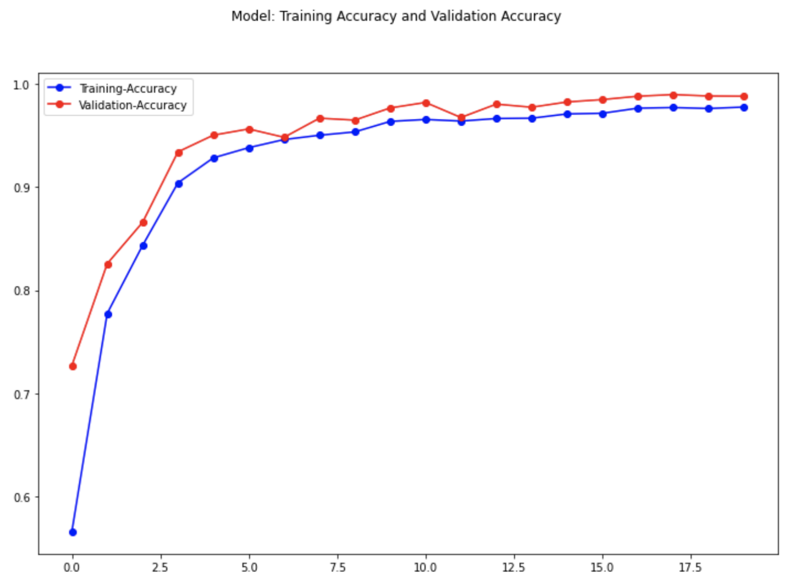


Figure 7: Using BatchNormalization

2. **Adding L2 Regularization** - We further modify the model by adding **L2 regularization** in each Conv2D layer. L2 regularization forces the model to take smaller weights. The model is able to achieve a training accuracy of 98.04% and a validation accuracy of 98.89%. (Figure 8)

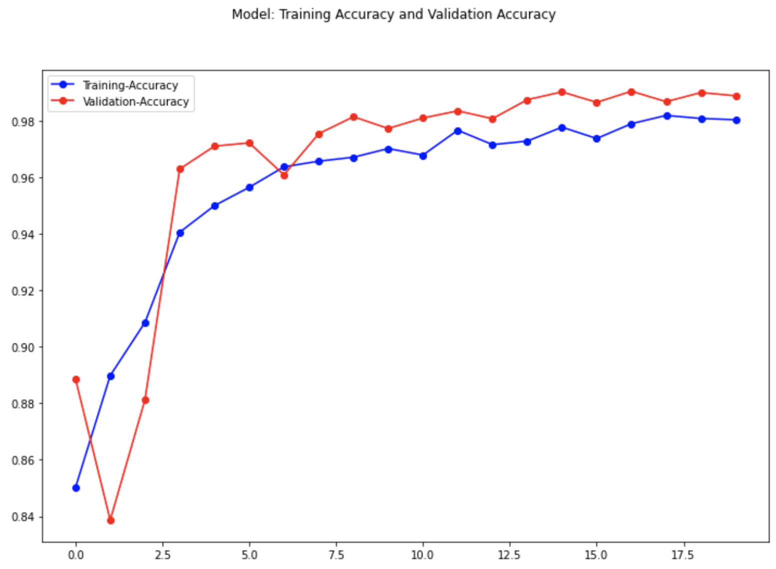


Figure 8: Using L2 Regularization

3. **Adding Dropout** - Finally, we add the **Dropout** layer with a rate=0.5 after the Flatten layer. The model can achieve a training accuracy of 94.60% and a validation accuracy of 97.52%. (Figure 9)

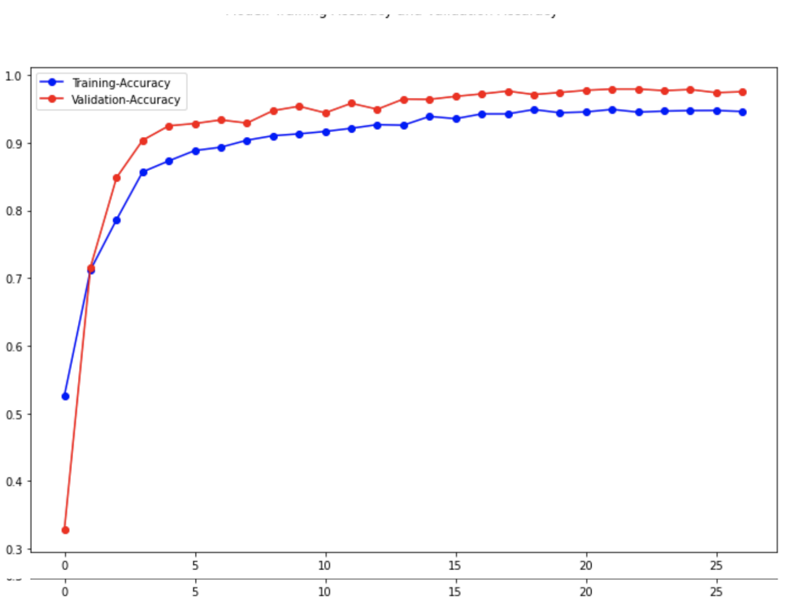


Figure 9: Using Dropout

VII.I Observations

We tried out three **Regularization** techniques, namely BatchNormalization, L2 Regularization, and Dropout. There are the following observations from these tryouts:-

1. **Slow convergence** - The final model with all the regularization layers was slow to converge as compared to the initial/base model, which is expected behavior.

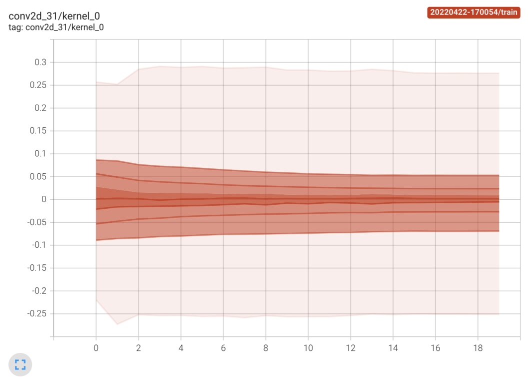
2. **Impact on Accuracy** - **Table 3** shows the impact on Training and Validation Accuracy as we applied regularization.

Regularization	Training Accuracy	Validation Accuracy
None	97.8%	98.2%
BatchNormalization	97.75%	98.80%
BatchNormalization, L2 Regularization	98.04%	98.89%
BatchNormalization, L2 Regularization, Dropout	94.60%	97.52%

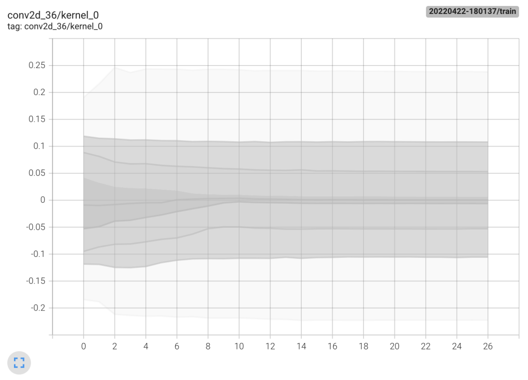
Table 3: Impact of Regularization

   As we can notice from the above table, the validation accuracy is maximum when we use both BatchNormalization and L2 Regularization

3. **Weight decay** - Introduction of L2 Regularization forces the model to lessen the value of weights. **Figure 10** represents the weights for the first Conv2D layer in the model. We can observe the regularized weights are in the range of -0.22 to +0.25, compared to non-regularized weights for the same layer in the range of -0.27 to +0.29.



(a) Non-regularized weights



(b) Regularized weights

Figure 10: Weights from First Conv2D in the model

VIII Advanced Model Architectures

After studying the impact of Data Augmentation and Regularization techniques, we decided to check the accuracy of this data set using Advanced Architecture like ResNet and check the performance of pre-defined models like VGG16.

ResNet stands for Residual Neural Network, which allows merging the original input and the output of a convolutional block using a skip connection.

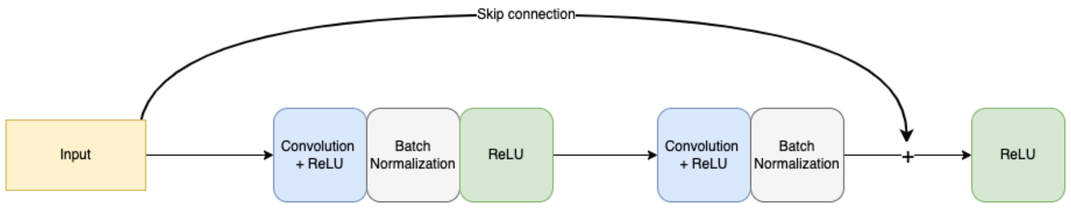


Figure 11: Residual Neural Network - Overview

In this phase, we build a resnet model with five blocks of Convolution, Pooling, BatchNormalization, Activation, and different Dropout rates and batch sizes. In addition, we tried the pre-defined models ResNet50 and VGG16 by following the following approach.

1. Download the pre-defined model- Predifned models can be checked at the Keras applications and be easily downloaded using **tf.Keras.application** API.
2. Freeze the trainable layers- Since we use the pre-learned models, we don't want to train the weights again. So, the layers in the model can be marked as *trainable=False*
3. Modify the last layer - The data set and the output classes are different in this case, so we modify the last layer per our requirement. As an example, we adjusted the number of classes in the *Dense* layer.

VIII.I Observations

We tried three models with advanced ResNet architecture and pre-defined ResNet50 and VGG16 models. There are the following observations from these tryouts:-

1. **Ease of use for pre-defined models** - The pre-defined model is easy to use by using *tf.Keras.applications* and following the standard procedure to freeze the learning layers and update the last layer. This cuts off the learning time; otherwise, the model would need to train with such a huge number of parameters.

2. **Big models may not always work** - ResNet50 and VGG16 are both heavy models, but we observe different accuracy from the model runs. Table 3 contains Training and Validation Accuracy and run-times for the first epoch for the testing in this phase.

Model	**Trg. Accuracy**	**Val. Accuracy**	**Epoch 1 - runtime**
ResNet (dropout=0.2, batch=32)	88.35%	91.57%	1.1 mins
ResNet (dropout=0.4, batch=64)	81.31%	86.64%	1.1 mins
ResNet (dropout=0.5, batch=96)	86.02%	90.09%	1.1 mins
pre-defined-ResNet50	54.12%	66.78%	67 mins
pre-defined-VGG16	92.62%	98.45%	94 mins

Table 4: Advanced Architecture Models - Performance

Based on Training and Validation Accuracy, the pre-defined model VGG16 is the best model in this experiment.

Conclusion

Throughout this exercise, we studied different phases of model development for an Image Data set, right from Data Preparation, to building models, leading to the impact of techniques like Data Augmentation and Regularization. Furthermore, we observed how to use the pre-defined models like ResNet50 and VGG16. In each phase, we had some observations that helped us decide when and where to use the models, parameters, or techniques used in each stage. First and foremost, it is vital to understand the Data set to be analyzed. Like in our case, we had the images for the first four characters **A, B, C, and D** from **American Sign Language**. Using a simple probability formula, we could determine the baseline accuracy of 25.0%. For any model to be considered good, it should be able to beat the baseline accuracy.

We primarily used the Convolutional Neural Network architecture to build and test the models to predict output labels for the data set. Most of the models performed well and delivered a Validation Accuracy above 90.0%. There are other metrics like Precision, Recall, and F1 score, which are essential when evaluating the performance of a model. The best model can provide a Validation Accuracy of 99.51%.

The impact of Data Augmentation is visual when using all the augmentation options in the *Image-DataGenerators*. This helps us expand the training set by ensuring that the same is never passed to the model for feature learning. Similar observations were noticed for the Regularization, which is another powerful method to control the problem of over-fitting. Callbacks help save time on model training and avoid burning unnecessary computing.

Last but not least, Keras provides a lot of pre-defined models that are fully trained and can be easily used on most common use-cases for Image Classification problems. While the training run time for the pre-defined models may be high, it still saves a lot of time on model development and provides a good comparison point for custom-build models.

Overall, the **CNN** models are pretty powerful algorithms for generic Image Classification problems. TensorFlow and Keras provide many methods, APIs, and tools to build models from fundamental to advanced architecture without writing much code. We can apply the practices from this report to develop models for an Image Classification problem.

References

- Chollet, F. (2021). Deep learning with python. Manning Publications.
- Stackoverflow - randomly distribute files into train and test directories, https://stackoverflow.com/questions/39210765/randomly-distribute-files-into-train-test-given-a-ratio
- TensorBoard — TensorFlow https://www.tensorflow.org/tensorboard/
- Deep residual learning for image recognition - arxiv.org. (n.d.). Retrieved April 27, 2022, from https://arxiv.org/pdf/1512.03385.pdf
- Team, K. (n.d.). Keras Documentation: VGG16 and VGG19. Keras. Retrieved April 30, 2022, from https://keras.io/api/applications/vgg/#vgg16-function