# Classification of Quick Draw Images

**Team Name: AlphaClassifier**

Pavithra Parthasarathy (Kaggle Name: pavithrarajasekar) UdeM Matricule: 20182678

Arka Mukherjee (Kaggle Name: arka161) UdeM Matricule: 20182625

## INTRODUCTION:

In this Kaggle Competition, we have attempted to build an effective Image Classification Algorithm that works well on unseen data. The given dataset contains two .npz files, one for training and one for testing. The categories are 6 unique categories - ant, spider, flower, dolphin, lobster, bulldozer. To beat the required baselines, we have divided our work into the following main stages:

1. Data cleaning and feature-design.
2. Selecting Algorithms.
3. Hyperparameter Tuning for optimal performance.
4. Model Training and Cross-Validation.
5. Predicting over the test npz dataset, saving result, and trying the output CSV on Kaggle.

The goal in our problem is to not just have good predictions on the public leaderboard, but also have a good classification algorithm that performs well on unseen data. We tried two broad approaches – one approach using classical Machine Learning Algorithms, and another approach using Convolutional Neural Networks (normal CNNs, and Transfer Learning Models in Keras/Torch). We have observed that classical Machine Learning algorithms perform a lot worse than Deep Learning models and doing Feature Extraction is trickier and generally not as effective as the features automatically extracted by DL techniques. For tuning hyperparameters, we have relied on a manual search and the Keras Fine Tune API.
The final technique that worked the best was an ensembled model of VGGs, with each VGG good for a particular class or subset of the problem. We had checked the confusion matrix of each VGG, and we tried to create a merged model that handled the overall problem without overfitting. It had an accuracy of 89.171 in the private leaderboard of the competition (rank 11/65), and regular CNN models without an ensemble had fetched us around 82-83%.

## FEATURE DESIGN:

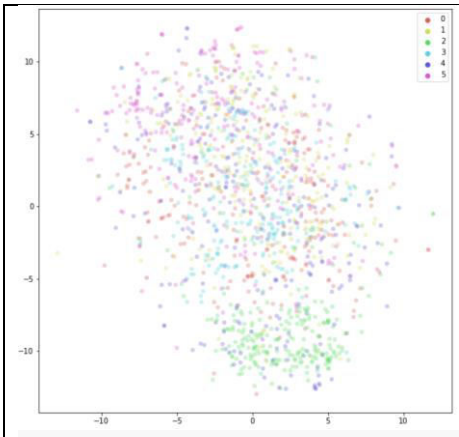
The first step was to investigate the dimensions and the channels of the image dataset. We tried to visualize a few images in each class and tried to find their common points. We tried a couple of filters such as darkness filter, mean filter, median filter, gaussian filter to see if the models have better prediction. The pixels were normalized by 255 and flattened to arrays for some of the preprocessing. We have noticed that normalizing the image Numpy array by 255 had improved the accuracy by over 20%[1].

Feature Reduction:
The first approach we tried was to use information from all the pixels, after reducing the dimensions. We applied the following techniques:

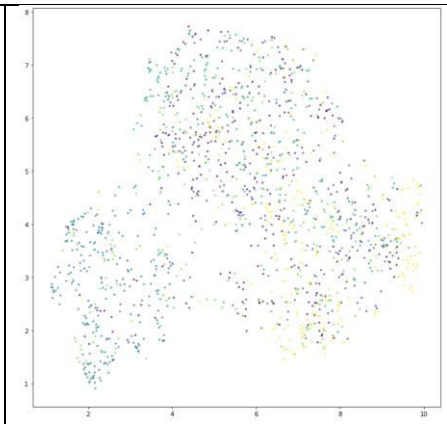
- TSNE (t-distributed stochastic neighbor embedding)[2]
- UMAP[3]
- PCA (Principal Component Analysis)
- Truncated SVD (Single Value Decomposition)

A GridSearch for the number of PCA components was found to be 20 which is also the minimum of the number of samples and the number of features based on the svd_solver. But this does not capture the 95% explained

variance. So, it was not fruitful for the results. UMAP and TSNE were tried to see if there were any natural clusters present in the images. However, there were none. All classes as seen below are intermixed.



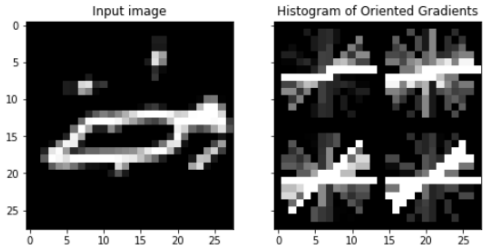
Visualization with TSNE and 2 components



Visualization with UMAP and 2 components

**HOG Transformation:**

We felt doodles are mostly about rough shapes/edges. Since the HOG (Histogram of Gradients)[4] descriptor focuses on the shape of the object and provides the edge direction as well, we gave it a shot. It performed decently when the extracted features were used in some ML models. Since the doodles are quite low in resolution, we experimented with the number of localized portions. The best was observed at 14 x 14 local space and 15 orientations. Using a space with an even lower dimensionality more noise than useful gradients, and a larger space was not very useful too.




## CNN + Image Data generator:

We tried to use a CNN to extract the useful features along with data augmentation done by Keras' ImageDataGenerator API to get some variance in the images. The changes made in the ImageDataGenerator hyperparameters affected the features extracted from CNN and thereby affecting prediction. Some parameters which did not help were vertical flip, feature wise and sample wise normalization. Few of them improved the model prediction like zoom, rotation, width and height shift. We manually tuned the parameters and created batches of images for each training and validation epoch. Since we had very low-resolution images (28, 28, 1), we kept all the params to very small variations, ranging from 0.2 – 0.3.

**ALGORITHMS:**

We investigated the following algorithms in combination with some feature designs.

***Support Vector Machine (SVM) – SVM + PCA, CNN + last layer as SVM, VGG + last layer as SVM:***

A Support Vector Machine is a classifier that operates on the principle of maximizing the margin from the classifier decision boundary to the support vectors. We applied SVM to the features extracted from PCA and a CNN/VGG. The features extracted from CNN were more useful, though it was high dimensional. SVM is independent of dimensionality of the feature space. Since the classifier is based on maximum margin, we were expecting good results. We also experimented with different kernels. Since was not linearly separable, we went with an RBF kernel. The problem was with whatever feature extraction we used, there was always a conflict between class 1 and 4. On closer look, some lobsters were incomplete and in some cases the ants were incomplete. So, the extracted features were not the right representation. In the case of CNN+SVM, we replaced the last softmax layer, with an SKlearn SVM model. We have tried this with transfer learning as well, in which the softmax was replaced with the SVM[5]. The results were mixed, and our final VGG models had softmax last layer for all models and an SVM only for one of the models (seemed to have the best confusion matrix with that combination).

***Random Forest Classifier:***

We used Random Forest Classifier from SKlearn with n_estimators as 180 and max_depth as 20. This proved to work the best with HOG descriptors. If max_depth was increased, there was overfitting and the validation accuracy dropped. The Random Forest Classifier could create arbitrary classification boundaries. So, it did not suffer from curse of dimensionality. Since this is an ensemble model, it worked better than many single models like KNN or SVM one vs all with PCA feature reduction. This gave us ~44% in the leaderboards.

***Convolutional Neural Networks (CNNs):***

CNN was one of the obvious choices for image classification. The principle of a CNN is having locally connected Neural Networks with a shared kernel and sliding the kernel across the image and then performing a pooling and subsampling operation. We built a network of Convolutional layers, with max pooling and dropouts, and analyzed hyperparameters. We added a dense layer after that and used Softmax Activation Function at the tail. Different minor configuration changes in the CNN performed better on different classes. So, we used 5 different CNNs (VGGs), trained the dataset on the ensembled model. Finally, we selected the class predicted with maximum probability. More conv layers had issues of overfitting. So, we restricted to using just 4 conv layers followed by max pooling. Similarly, we also used fewer number of nodes in each layer (32, 32, 64, 128) layer. Increasing the number of nodes also cause overfitting. We also experimented with the kernel weight initializer as we read that Kaiming performed better than glorot_uniform, due to the nature of the ReLU activation function used in each layer except the last dense layer. But in our case, we used Kaiming only in the dense layers[6], which led to slight improvement in accuracy.

***Transfer Learning (with Keras VGG-16 pretrained using ImageNet Weights):***

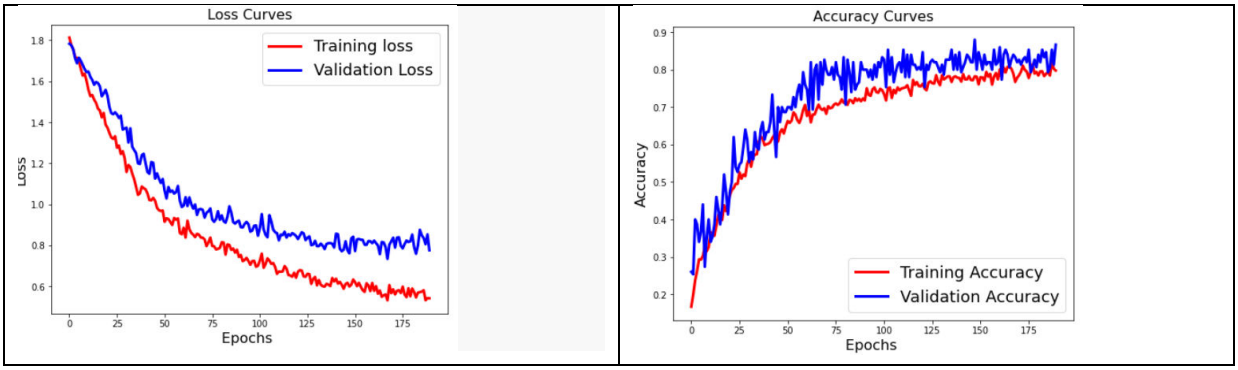
Apart from the above, we also tried KNN, Voting classifier, MLP, CNN+Xgboost. But all of them gave poorer result in comparison to the CNN+SVM or Transfer Learning methods. Transfer Learning means using domain knowledge of a problem in another. We used a Neural Network trained on ImageNet weights (Keras' VGG-16), and froze some layers with existing weights, and re-trained the result after feature extraction using our data. Freezing till Block 3 in the architecture gave us the best results, although freezing till Block 4 is very competitive/comparable too[7].

**METHODOLOGY:**

This section could be split into Machine Learning (ML) models and Deep Learning (DL) models. For the ML models, we tried different filters, in combination with PCA or HOG transformations. We used GridSearch for the different parameters used in SVM and Random Forest models. We used ***Cross Validation*** to analyze our models. The train, validation set was split in the ratio of 80:20. The validation technique used was K-Fold since it had unbiased estimation of loss. The final model (used for generating the CSV) was trained on all the data. For DL models, the basic strategy used was to train the model with 90% of the training data available. We adjusted the ImageDataGenerator several times to see performance on the validation set (val_acc and val_loss). We trained for

more epochs than required and then monitored the loss and accuracy curves to decide on epochs to train. We also used *EarlyStopping* and *Dropouts* when we saw that the validation loss was increasing, detecting an overfit situation. We added dropout (regularization) of 0.2 after each maxpooling layer. Dropout basically is a ***Regularization*** technique that refers to rate of neurons ignored during training. In our case, Batch Normalization did not help. Finally, let us elucidate our final (best) algorithm's methodology:

- We check the VGG-16's prediction with a confusion matrix and the val_loss, val_acc.
- Choose the model based on the balance of scores across different classes.
- Then, we save the weights.
- Re-train with small tweaks in different params until there is a better performance on a different class.
- Ensemble all the weights and use it for final prediction.




**RESULTS:**

| Algorithm | Leaderboard Type / Accuracy | Accuracy (%) |
|---|---|---|
| SVM | Public Leaderboard | 25.955 |
| | Private Leaderboard | 25.478 |
| Random Forest (with Histogram of Gradients) | Public Leaderboard | 44.077 |
| | Private Leaderboard | 43.921 |
| CNN (Keras Fine Tune) | Public Leaderboard | 82.416 |
| | Private Leaderboard | 82.888 |
| CNN ensemble (weak CNNs x 15) | Public Leaderboard | 80.611 |
| | Private Leaderboard | 80.714 |
| VGG-16 | Public Leaderboard | 83.705 |
| | Private Leaderboard | 83.914 |
| VGG-16 Ensemble (Best) | Public Leaderboard | 88.500 |
| | Private Leaderboard | 89.171 |
| VGG + SVM (Ensemble) | Public Leaderboard | 85.300 |
| | Private Leaderboard | 85.678 |
| TorchVision Resnet18 | Training Accuracy | 58.513 |
| | Validation Acc | 62.538 |

In our Support Vector Machine, we had observed that the results were quite poor both on the public and private leaderboard. Our normal CNN gave us a decent score of over 82% on both the leaderboards, and the model had a configuration that was chosen by Keras Fine Tune after a Hyperparameter search and some manual tweaks. We observed that the CNN did well on all classes, except for class 1 and class 4. After that, we tried to ensemble 15 different kinds of CNNs. Even though the confusion was a little more balanced, our accuracy seemed to gone down (at 80.611%), as 15 CNNs were depending on a degree of Randomness in the ImageDataGenerator API. Then, we tried a VGG-16 pretrained model with some layers having ImageNet weights, and some being trained. We were able to achieve an accuracy of 83.705. Our final model was a combination of several VGGs, and the accuracy was 88.5% in the public leaderboard.

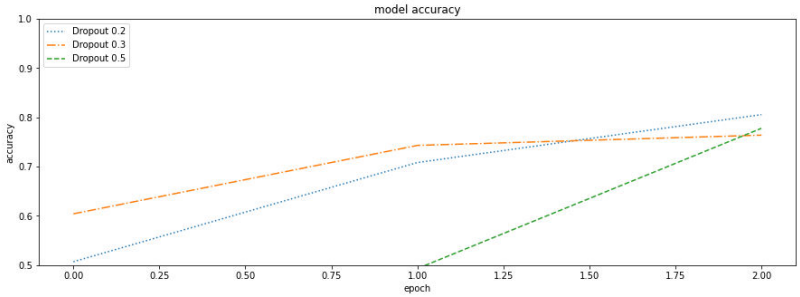
**ImageDataGenerator in Keras analysis (for Data Augmentation):**

We have observed that changing parameters here have a drastic effect on which class is predicted better. If we had set the parameter "samplewise_std_normalization" to True, our confusion matrices indicated that class 4 was predicted better. And, if we set "zca_whitening" to True, class 1 would be better. If we ignored both or set both to true, we would get an overall balanced confusion matrix for all the classes. For our final VGG ensemble model, we had the following config: 1 model good for class 4, 1

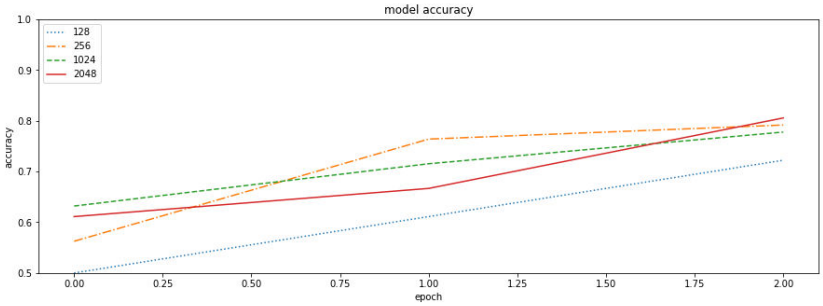
model good for class 1, 3 models balanced amongst all the classes (shown in Appendix). We had this sort of split as class 4 and class 1 seemed relatively harder to predict. Setting Horizontal Flip (Mirror Flip) to true helped us here, as mirror images of the classes are usually the classes themselves. Vertical/Y Axis flip made the results worse. We had also zoomed into the images and done a small height shift and width shift[8].

**CNN Hyperparameters and model analysis[9]:**

1. **Dropouts:** Dropouts are generally used for preventing overfitting in CNNs. The technique basically means the rate of neurons that are randomly ignored during training. This tends to reduce the model complexity and hence reduce overfitting. We had noticed that if we do not use dropouts in our VGG model, there was some overfitting present, and the validation accuracy was not as great compared to the training accuracy. We had first tested our model without dropout, and in more tests, the validation accuracy was around ~10% lower than the training accuracy. Then, we gradually started adding dropouts, and our best model had a dropout rate of 0.2 after the first broad layer (VGG with top removed), and 0.3 before the last dense layer. We have a higher dropout rate before the dense layer as intuitionally it seems like a layer that might increase the model capacity more. The graph for the first dropout layer is shown below. We had selected 0.2.




2. **BatchNormalization:** It is a technique used to make CNNs more stable by normalizing the input layer, through re-centering and re-scaling. In our case, we noticed that adding a BatchNormalization layer after the VGG main layers did not have a noticeable impact, and in some tests, the Cross Validation Accuracy had dropped a little bit. We had dropped BatchNorm from our VGG model after some tests.
3. **Dense Layer Configuration:** We have tried a number of dense layers, including 128, 256, 1024 and 2048

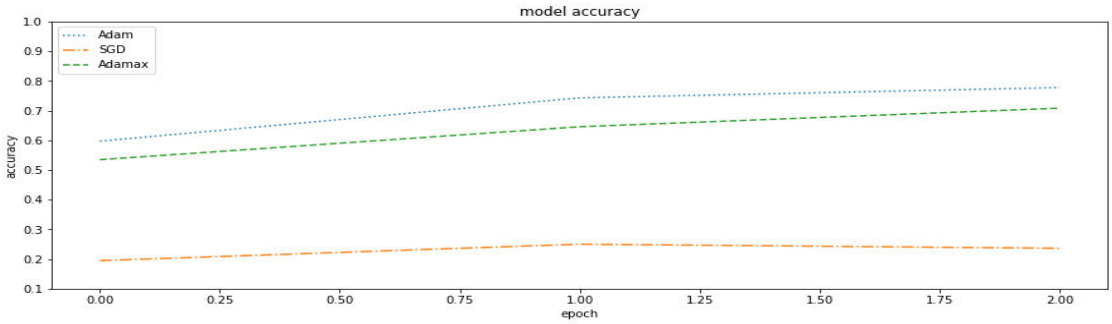



Based on the below graphs, and our test results, 1024 and 2048 seemed to perform the best. We finally decided to go for 1024 for a trade-off in performance vs accuracy. 256 seems promising in our graphs, but it seems to generalize a little worse compared to 1024 neurons in the dense layer. A point to note is that each epoch in the above graph is a sub-epoch for the VGG-16.

4. **Image Resize (for VGG-16's Input Tensor):** We noticed that re-sizing the input tensor to 80 x 80 performed better than just re-sizing to 32 x 32 (minimum size for VGG-16). We used pillow and TF's array_to_img for our transform, which seemed to work better than cv2's transform by a small degree.
5. **Pre-trained models tested:** We have tried a number of pre-trained models including InceptionNet V2, MobileNet, VGG-19, Resnet50, Resnet-18, and we observed that in our code, VGG-16 performed the best,

closely followed by VGG-19. We chose VGG-16, as it is one of the simplest model out of all the available options in PyTorch and Keras.

6. **Layers Frozen:** In VGG, if we do not want to use ImageNet weights, we can retrain the some of the layers. In our model, we have trained the last 2 major layers in the VGG-16 archiecture. We noticed that training the last 2 layers (i.e Block 4 and Block 5) performs the best, and training more layers increases computation cost and makes performance worse. We chose "block4_conv3" as the stopping point in our loop where we set layers as not trainable [10].
7. **Optimizer:** We ran a GridSearch for KerasClassifier for our model. We had observed that the Adam optimizer had performed the best. The below graph was obtained with LR = 0.001. Adam performed the best in all CV tests even after fixing a more apt learning rate for the SGD Optimizer.




8. **Learning Rate:** Our SKLearn GridSearch had returned that the learning rate 0.001 for the Adam optimizer performs the best. We could observe good results with that learning rate in our cross validation and leaderboard accuracy as well.

**DISCUSSION:**

In this competition, we started out with a generic CNN in Keras. Then, we added data normalization by 255, and fine-tuned various CNN parameters (dropouts, layers, neurons in each layer, max pooling, etc). Then, we saw it was hard to push beyond 82 or 83, and we moved on to Transfer Learning. The model that worked the best for us overall was a VGG model with the above hyperparameter choices. Each VGG in the Soft Voting Classifier was the same, and the only difference was in the Image Augmentation parameters, as they make some classes better than others. Furthermore, ensembling made us ensure we are not just overfitting on one class (since we do not know the class distribution of the public leaderboard), and our overall model is balanced.
We had also tried out classical Machine Learning Algorithms, and FeatureExtraction using various feature maps. Some algorithms scored far worse on unseen data than validation data, and they did not generalize well (eg: Support Vector Machines). On the other hand, our Random Forest with HOG model had very similar CV and Kaggle Leaderboard accuracy. We believe that our model has scope for improvement, and we can train it more intelligently with different image re-scaling, and dropout/dense layer configurations. Furthermore, there may be better pretrained CNN architectures for this problem than a VGG-16 with ImageNet weights. Class 1 and Class 4 are the hardest to classify, and even as a human being, it is sometimes easy to make a mistake while looking at sample images from those classes. Even after a lot of effort in ensembling, class 4 still had the lowest value in the prediction confusion matrix. A more balanced matrix can improve overall predictions/accuracies. Overall, we could conclude that CNN features extracted are better than classical techniques, and CNN works the best here as we could attain a lot of data using Data Augmentation using Keras' ImageDataGenerator.

**STATEMENT OF CONTRIBUTIONS:**

Both team members contributed equally and worked together for each task - data analysis, developing methodology, coding and optimizing, result and graph analysis, and writing the report. We hereby state that all the work presented in this report is that of the authors. All work is unique and our own, and referenced works used has been cited.

**REFERENCES:**

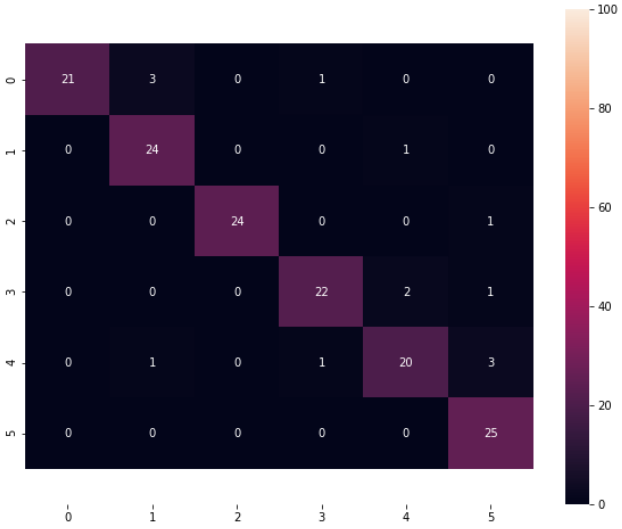

1. https://machinelearningmastery.com/how-to-manually-scale-image-pixel-data-for-deep-learning/
2. https://scikit-learn.org/stable/modules/generated/sklearn.manifold.TSNE.html
3. https://umap-learn.readthedocs.io/en/latest/basic_usage.html
4. https://en.wikipedia.org/wiki/Histogram_of_oriented_gradients
5. https://github.com/AFAgarap/cnn-svm
6. https://towardsdatascience.com/weight-initialization-in-neural-networks-a-journey-from-the-basics-to-kaiming-954fb9b47c79
7. https://towardsdatascience.com/a-demonstration-of-transfer-learning-of-vgg-convolutional-neural-network-pre-trained-model-with-c9f5b8b1ab0a
8. https://keras.io/api/preprocessing/image/
9. https://www.kaggle.com/cdeotte/how-to-choose-cnn-architecture-mnist
10. https://arxiv.org/abs/1409.1556


**APPENDIX:**

To get better results with ensembling, we used the models' weights that provided the following confusion matrices. We provide the confusion matrix and the accuracy scores of the VGG models we had ensembled. We have a total of 5 models that contribute to a soft/probabilistic vote.

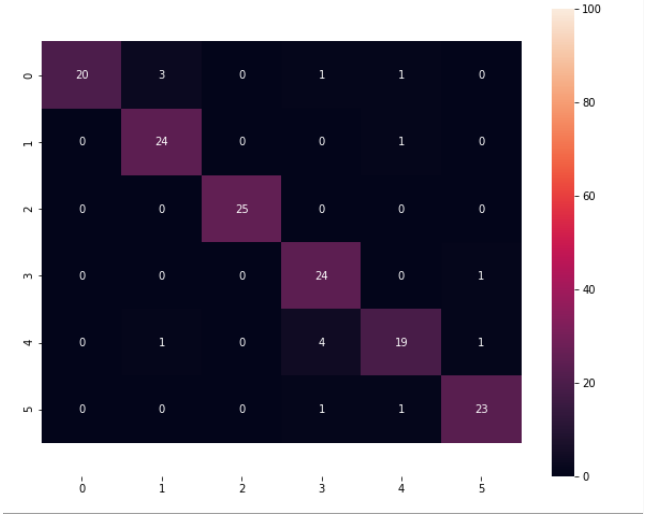
1. **Model with high accuracy (balanced model - 1):**

```
Accuracy Score is displayed below:
0.9066666666666666
[18]: array([[21,  3,  0,  1,  0,  0],
       [ 0, 24,  0,  0,  1,  0],
       [ 0,  0, 24,  0,  0,  1],
       [ 0,  0,  0, 22,  2,  1],
       [ 0,  1,  0,  1, 20,  3],
       [ 0,  0,  0,  0,  0, 25]], dtype=int64)
```



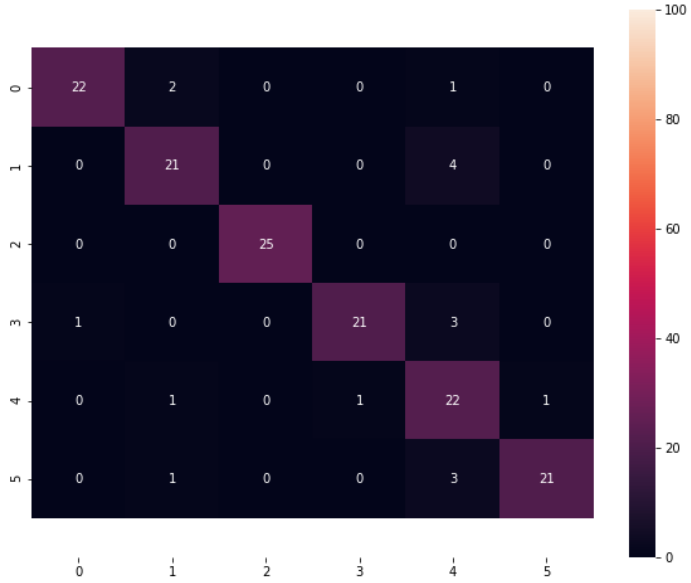
## 2. Model with high accuracy (Balanced Model 2):

```
Accuracy Score is displayed below:
0.9
array([[20,  3,  0,  1,  1,  0],
       [ 0, 24,  0,  0,  1,  0],
       [ 0,  0, 25,  0,  0,  0],
       [ 0,  0,  0, 24,  0,  1],
       [ 0,  1,  0,  4, 19,  1],
       [ 0,  0,  0,  1,  1, 23]])
```



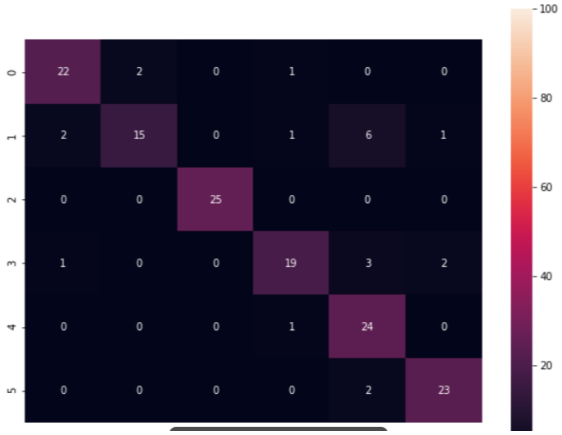

## 3. Model with high accuracy (balanced model – 3):

```
Accuracy Score is displayed below:
0.88
array([[22,  2,  0,  0,  1,  0],
       [ 0, 21,  0,  0,  4,  0],
       [ 0,  0, 25,  0,  0,  0],
       [ 1,  0,  0, 21,  3,  0],
       [ 0,  1,  0,  1, 22,  1],
       [ 0,  1,  0,  0,  3, 21]])
```



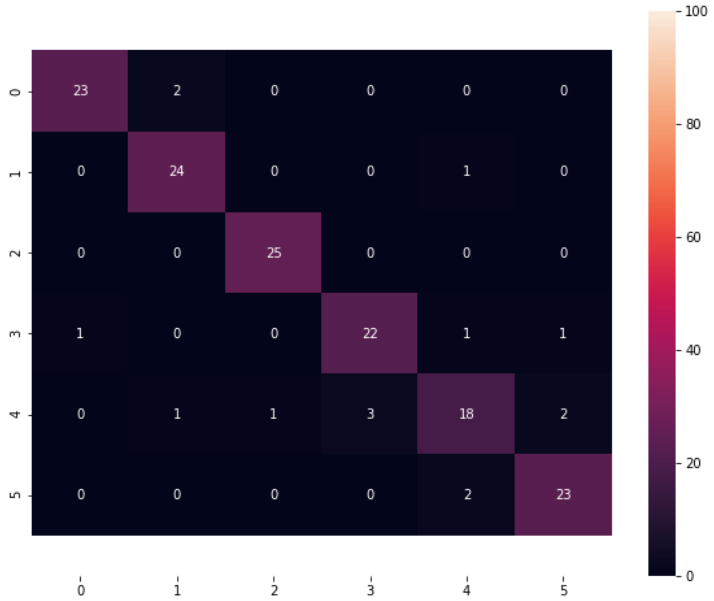
**4. Good model, but best for class 4:**

```
Accuracy Score is displayed below:
0.8533333333333334
array([[22,  2,  0,  1,  0,  0],
       [ 2, 15,  0,  1,  6,  1],
       [ 0,  0, 25,  0,  0,  0],
       [ 1,  0,  0, 19,  3,  2],
       [ 0,  0,  0,  1, 24,  0],
       [ 0,  0,  0,  0,  2, 23]])
```



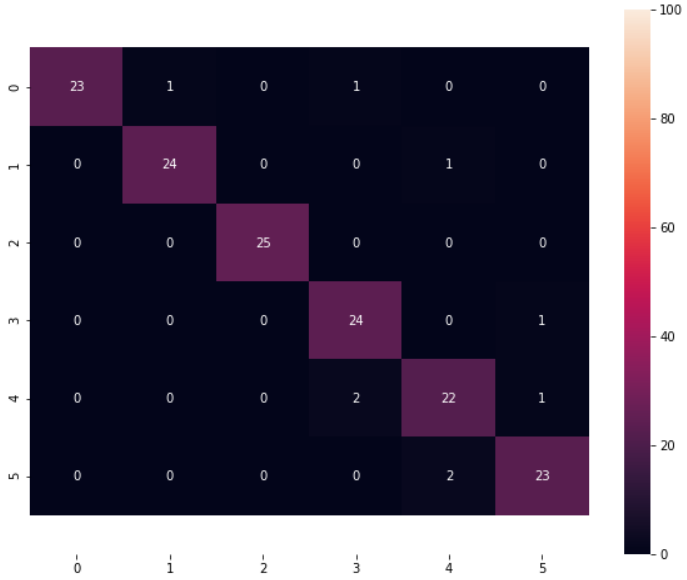

**5. Good Model for Class 1:**

```
Accuracy Score is displayed below:
0.9
array([[23,  2,  0,  0,  0,  0],
       [ 0, 24,  0,  0,  1,  0],
       [ 0,  0, 25,  0,  0,  0],
       [ 1,  0,  0, 22,  1,  1],
       [ 0,  1,  1,  3, 18,  2],
       [ 0,  0,  0,  0,  2, 23]])
```



6. **Final Soft Voting algorithm – combining all of the 5 above models:**

```
Accuracy Score is displayed below:
0.94
[35]: array([[23,  1,  0,  1,  0,  0],
       [ 0, 24,  0,  0,  1,  0],
       [ 0,  0, 25,  0,  0,  0],
       [ 0,  0,  0, 24,  0,  1],
       [ 0,  0,  0,  2, 22,  1],
       [ 0,  0,  0,  0,  2, 23]], dtype=int64)
```




**Additional Note:** To save RAM, we had converted the Numpy Array from Float64 to Float32. Even after a conversion to Float32, our result array resized to a higher scale and having 3 different channels was still occupying 10GB in our system memory. Furthermore, we used an RTX 2060 with 6GB of VRAM for training the VGG-16 models.