CSC 732

Final Project Part 2 Analysis: Dice Classification Imbalance, Performance Evaluation, Ensembling

Team Members: Dominic Klusek, Jonathan Rozen

Image classification is a supervised learning task that involves interpreting a class label from an image and is often used to identify many images. More traditional methods involve the creation of features that were derived from pixel data, histogram, textures, and shapes; building a robust model utilizing these features was a time consuming and challenging prospect even for skilled experts. Machine Learning is becoming a fast-rising method to instead allow the system to learn and create features with the only input being data. Supervised Machine Learning is a subset of Machine Learning in which models are trained on data that consists of both the original data features and the class labels that the models should predict for those features. One of the main architectures used for supervised image classification is the Convolutional Neural Network(CNN); a model that utilizes filters and the convolutional operator to find and extract certain edge data; and then passing the found edges and passing this data to a classifier such as a Multilayered Perceptron(MLP) or a Support Vector Machine(SVM). With these models there is a large amount of modularity in the form of changeable parameters such as the number of layers, size of layers, optimization strategy, etcetera, and etcetera. Images in themselves have a large amount of variation between one another such as the position of the object, background behind the object, ambient lighting, camera angle, and camera focus. This sort of variation is the reason that CNNs are so popular when it comes to image tasks; with their modularity and their ability to calculate and learn hundreds if not thousands of filters to capture different edge data makes it so they can learn to distinguish between and images of different classes. However in some cases a single classifier may be unable to perfectly learn a problem due to limitations in its design; for this reason in some cases multiple models are run in tandem and utilizing the predictions from all these slightly different or same models it is possible to create a model which performs similarly or even better than each of the models separately. This is called ensemble learning, and it is a powerful way to normalize and possibly boost performance of a system, though at the cost of more calculations and memory.

The image classification task that we took on was to classify between different polyhedral dice; these dice are commonly used in a variety of boardgames, tabletop wargames, or pen-and-paper role playing games. Being able to distinguish different dice from one another could be helpful to be in part of a larger system such as one that automatically tallies up dice rolls or simply ensures the correct dice are being used. The dataset that was utilized was the “Dice: d4, d6, d8, d10, d12, d20 Images” from Kaggle.com which consists of ~16,000 image of polyhedral dice. Polyhedral dice are simple dice which have several sides equal to the number after the letter *d* in their name; each of their shapes is based on a polyhedron; the d4 is a tetrahedron, the d6 is a cube, the d8 is an octahedron, and so on and so forth; each side has a different number from 1 to *n* where *n* is the number after the *d*. The d4, d8, and d20 utilize triangles, the d6 utilizes squares, and the d10 utilizes a kite shape. Since these dice are used in so many different scenarios there is a huge variety of different dice to choose from in stores and online; dice sets have different colors, different forms or highlights edges, different materials, they can be translucent, and some have marbling affects on them to add personability to different dice. Due to this large amount of variety in dice composition color becomes a poor choice of feature to tell some dice apart from others unless you were creating a system for a specific game with each die group having a different color. Therefore, the best way to visually distinguish between these dice is usually the shape of them, however depending on the angle at which you look at the die it may be difficult to tell the exact shape of the die due to lighting or the coloring of the die.

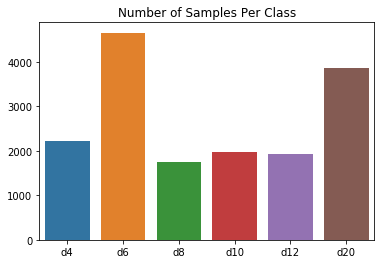
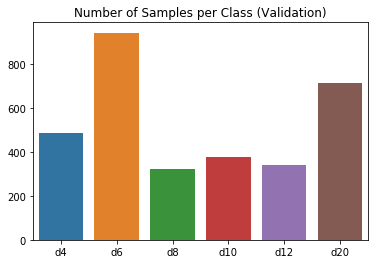
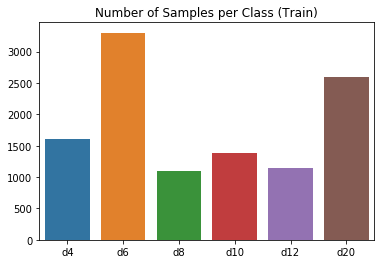
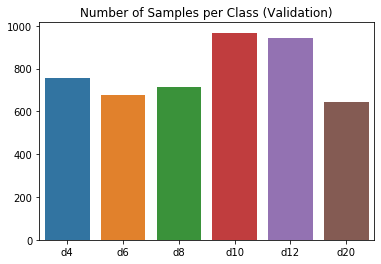
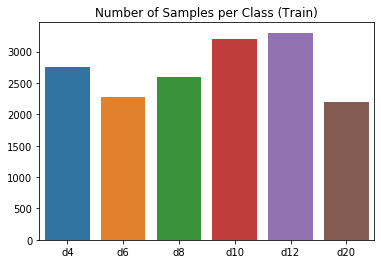


Figure 1. Distribution of class frequencies in original dataset



Figures 2-3. Distribution of training and validation subsets after dataset splitting

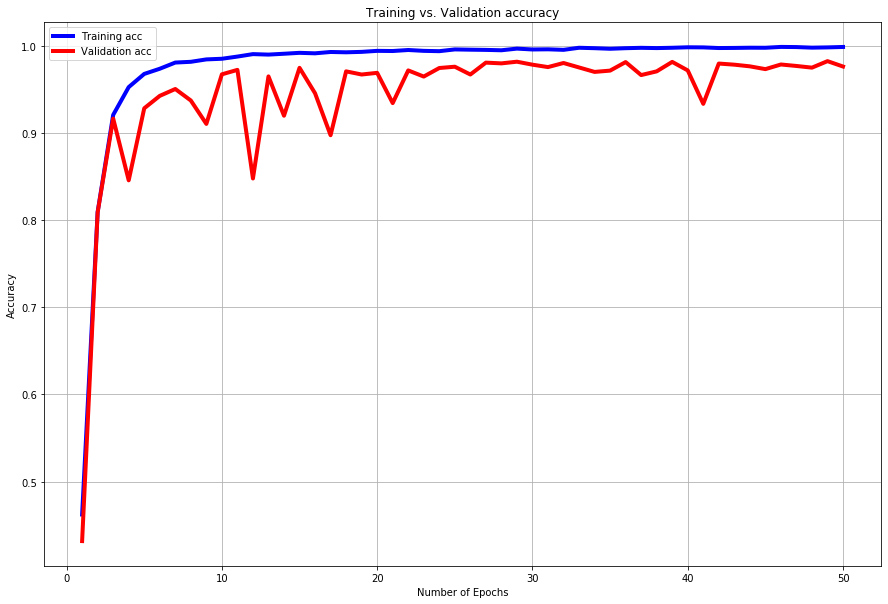
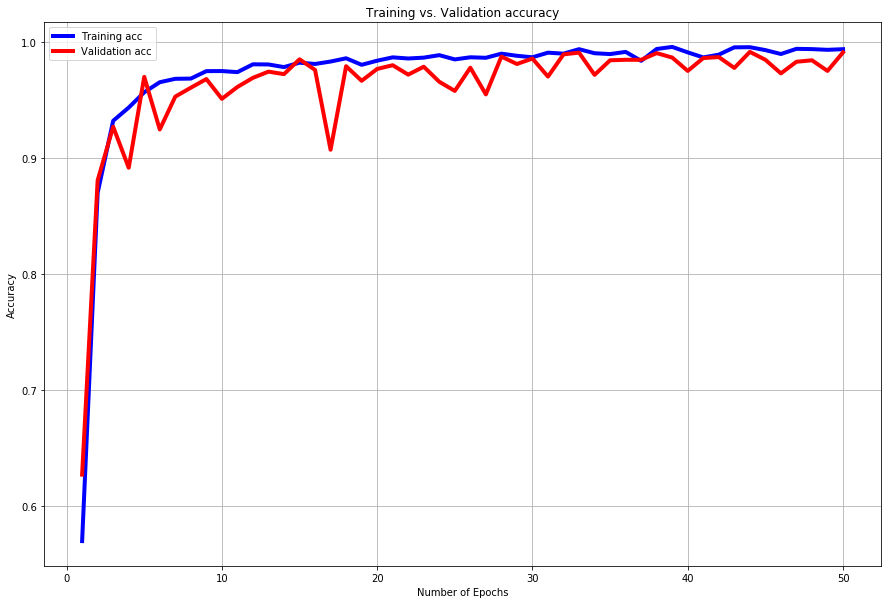
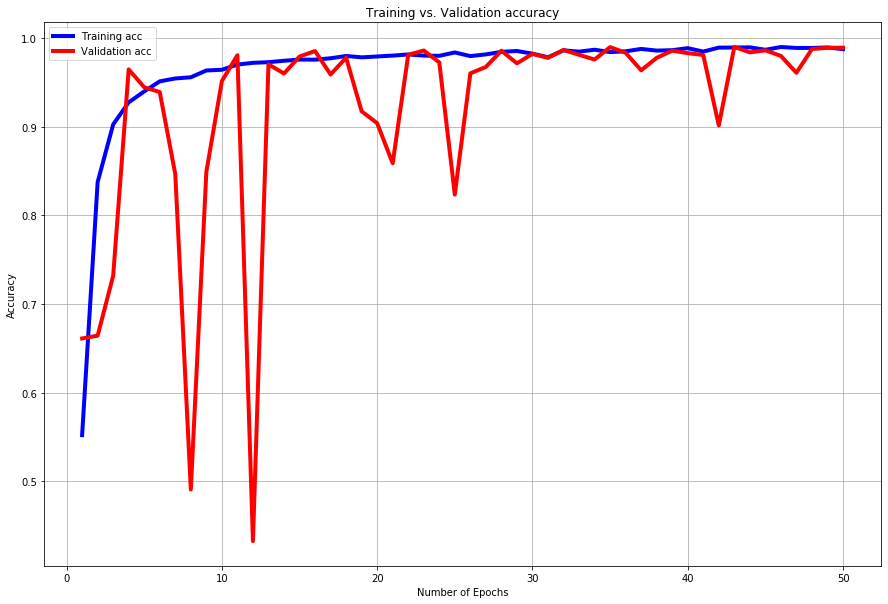


Figures 4-5. Distribution of training and validation subsets after image augmentation

Before training the model we split the data into training, validation, and testing sets; 70%, 20%, and 10% respectively to effectively train and gauge the performance of each model. One problem with the dataset is shown in Fig 1; there is a class imbalance with classes d6 and d20 both having close to or more than double the number of examples of the d4, d8, d10, and d12 classes. As can be seen in Figs 2-3 this imbalance is also evident after separating the dataset into the three subsets and must be solved otherwise the model can develop a class bias which will affect the end performance of the model. To solve the class imbalance, we used used the ImageDataGenerator class from Keras which can be set to random transformation for image data; the transformations we chose were random horizontal flipping, shearing from [0,0.2], zooming from [0,0.2], rotating from [0,40]°, and height/width shifting from [0,0.2]; image augmentation was applied to the training and validation subsets to create a more balanced dataset the end result is the distribution in Figs 4-5.Since there is a large variety of different dice and backgrounds we went for a simple augmentation strategy of creating a single augmented image per image in the subset; rather than perfectly balancing each class perfectly. One strange problem that we encountered came from splitting the dataset into the three subsets. When saving the images to their respective class folders some images were saved into the incorrect class folder and as such when trying to display examples of incorrectly and correctly classified dice there were inconsistencies that could not be ignored. We initially tried to remedy this by carefully debugging the script that we made to split the dataset, but could not find any error in it; instead, to remedy this problem we sifted through each class folder in each subset and deleting the misplaced images.

In terms of designing our CNN we followed a simple strategy of increasing the number of filters in the CNN block as the number of layers increased. In addition, we decided to utilize a small MLP block to act as our classifier as it is simple to connect and implement in Keras. The CNN block consisted of 4 sub-blocks; each sub-block consisted of a 2-D Convolutional layer, followed by a Relu activation layer, which was then followed by a Dropout layer with a 0.2 dropout ratio, and finally a Maxpooling2D to reduce dimensionality between sub-blocks. The number of filters for each 2-D Convolutional layer were [32, 64, 128, 128] respectively; we increased the number of filters to each consecutive 2-D Convolutional layer because the inclusion of the Maxpooling layers meant that the dimensionality increase would not be as extreme. We included Dropout layers because by randomly setting some weights to 0 the model is forced to learn more generalized features which reduces/eliminates overfitting. The MLP block consisted of 2 Dense layers of size [512,6] respectively; the first Dense layer takes the flatten output of the convolutional block, and then applies the Relu activation function. The second Dense layer utilizes the Softmax activation function as it calculates the probability distribution for all different classes and is most effective when applied to class labels that are in a sparse categorical format. In addition there is a Dropout and Dropout layer with ratio 0.2 and a BatchNormalization layer between the 2 Dense layers; BatchNormalization is a technique to control the size of updates across all updates during the backpropagation step, and effectively normalizes the updates and prevent overly large updates which can cause inconsistent learning.

Utilizing this base CNN model we built three models in which the only difference was the optimizer; we decided to use Adagrad, Adam, and RMSProp as they are all optimizers which have variable learning rates which will help with learning the problem more effectively. Each of the models was trained for 50 epochs, with a batch size of 32 utilizing the categorical cross entropy loss function; this was an acceptable number of epochs for the models to reach an acceptable accuracy of >99% and to stabilize; we did run a test with 100 epochs but rather than gaining any more performance, the system simply continued to stagnate around the original accuracies that model trained for 50 epochs achieves.

Figs 6-8 Training and Validation accuracies plots for Adam, Adagrad, and RMSProp (top-left to bottom-left)

From our previous tests with the architecture and dataset that was used we had a good idea of the expected performance but were hopeful with the larger training and validation sets would result in higher or similar accuracies. In our previous results where the training set only had 6,600 image and the validation set had 1,200 images the model achieved a max accuracy of 97.77% accuracy which is quite good considering we did not utilize the entire dataset, and instead utilized the ImageDataGenerator in Keras to create augmented images. The models on our new dataset which contains both augmented and non-augmented images and having a much larger variety of images achieved much better accuracies of 99.43% utilizing Adam, 99.11% utilizing Adagrad, and 99.49% utilizing RMSProp. A not insignificant increase of ~2% increase in performance on our testing set which is a good sign that our data is cleaned of an incorrectly placed images, and each model effectively learned to distinguish the different dice. Accuracy is good measurement of the performance of classifiers, but a classification report which consists of precision, recall, f1, and accuracy scores for each class provides deeper knowledge into where the shortcomings of a classifier derive from. The precision score is a confidence score that is a measure of the classifiers confidence that when predicting a certain class it is said class; recall is the classifier’s ability to effectively distinguish between the given class and all other classes; f1-score is an average of the precision and recall scores. Looking at the precision and recall scores for all our models we see that the precision and recall scores are usually close to one another and both were close to 1.00 which is a good indicator that our classifiers are not only confident when they make their predictions they are also fairly accurate. Another performance metric is plotting the Receiver Operating Characteristic Curve(ROC), and computing the Area under the Curve (AUC); the ROC plot is created by plotting the recall against the precision at various threshold values, and AUC is equal to the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one, and AUC is often utilized to compare the performance between different classifiers; with a classifier with a higher AUC is on average much more effective than one with a lower value. ROC is also helpful in finding which classes a classifier struggles to classify and can help to determine is a change in the overall system such as the architecture, dataset, or preprocessing of the dataset is needed. Since our models all achieved accuracies of over 99% the ROC curve showed lines that rise rapidly and then stagnate at a true positive rate of 1.0 regardless of the false-positive rate; meaning that for any amount of true positive and false positive examples in a set they are effective in distinguishing between one class and the other classes.

When constructing an ensemble classifier there are different strategies: bagging, Adaboost, voting, and stacked classifier. Each of these different strategies have different methods to form a final prediction; bagging creates separate classifiers which are trained on random subsets of the training set and then each of their predictions are aggregated to form a final prediction. Adaboost creates weak classifiers and trains them on repeatedly modified data and then their predictions are combined in a weighted majority vote to produce the final prediction. Voting consists of training similar or completely different classifiers and then combining their predictions in an unweighted majority vote to get a final prediction. Stacking consists of models being trained and then their outputs are stacked and passed to a final classifier which then forms the final prediction. We initially thought of using one of the ensemble methods built in Scikit-Learn, but the problem is that those methods such as VotingClassifier require that the models not be trained which we do not want to retrain our models. So instead we followed the hard-voting ensemble method outlined by Jason Brownlee in “How to Develop an Ensemble of Deep Learning Models in Keras”, which takes an average of the predictions of the 3 models and then picks the class that had the highest average to make our choice. This means that each classifier has an equal contribution to the final results of the ensemble classifier; which may not be the optimal solution; in some cases it may be better to have a hard voting system in which each classifier has a weight associated to it so that more reliable classifiers contribute the most and less reliable classifiers contribute little or nothing to the final result. The ensemble classifier achieved a final accuracy of 99.43% which is similar in performance to the model trained with the Adam optimizer. When comparing the macro average of the precision between the ensemble model and the Adam model; [0.9950, 0.9959, 0.9954] and [0.9949, 0.9959, 0.9954], which shows that the precision of the ensemble model is slightly better than the Adam model, but otherwise it is similar in performance. Most likely the model is being limited by the performance of the 3 models in conjunction as they probably each become confused on similar examples and thus even as a majority are unable to distinguish between them.

Although our ensemble classifier performed almost exactly like our CNN trained with the Adam optimizer our models achieved very high accuracies and were all able to learn the problem on their own. This problem was a good problem as the large variety of the dice in the dataset meant that models needed to focus more on the underlying shape of the dice rather than any color. The different backgrounds also mean that the models trained on this dataset will be able to effectively work on most common surfaces; image augmentation and the different image angles mean that the models will also be able to work at a multitude of camera angles.The largest problems we had was cleaning up our dataset to remove images that were improperly placed in incorrect class folders due to some strange performance of our dataset splitter; which caused some drop in performance due to incorrect class examples in incorrect folders. Afterwards mitigating some of the extreme class imbalance in the dataset was easily mitigated with image augmentation and allowed for our base classifiers to not develop a class bias towards the majority classes. The increased amount of data improved the performance of our models as compared to the previous model from part 1 of the assignment. Finally, the voting ensemble classifier was able to also achieve great performance, but it is a shame that it was not able to perform better than our base classifiers; perhaps a different strategy such as Adaboost may have been able to increase performance.

References:

* Brownlee, J., 2018. *How To Develop An Ensemble Of Deep Learning Models In Keras*. [online] Machine Learning Mastery. Available at: <https://machinelearningmastery.com/model-averaging-ensemble-for-deep-learning-neural-networks/> [Accessed 9 May 2020].
* TensorFlow. n.d. *Image Classification  |  Tensorflow Core*. [online] Available at: <https://www.tensorflow.org/tutorials/images/classification> [Accessed 9 May 2020].
* Lurig, M., 2018. *Dice: D4, D6, D8, D10, D12, D20 Images*. [online] Kaggle.com. Available at: <https://www.kaggle.com/ucffool/dice-d4-d6-d8-d10-d12-d20-images/> [Accessed 9 May 2020].
* Team, K., 2020. *Keras API Reference*. [online] Keras.io. Available at: <https://keras.io/api/> [Accessed 9 May 2020].