Simplifying Pathfinder: Automatically Calculating the Sum of Rolled Dice

Hana Lee
Department of Computer Science
Stanford University
`leehana@stanford.edu`

Abstract

Pathfinder RPG and many other tabletop games rely on rolling dice, which can become difficult and slow to add together in large quantities. I design a system capable of deciphering a picture of multiple rolled dice and calculating the sum of the values shown on the dice. Using a CNN to predict the value shown on a die using multi-class classification, I achieve 42.39% accuracy on a test set of unseen examples, with considerably higher accuracy for some classes and die types than others.

1 Introduction

Pathfinder RPG is a tabletop roleplaying game, similar to Dungeons and Dragons, that relies on dice-rolling for many of its core mechanics, such as combat, social interactions, and skill checks. When the player characters reach high levels, the amount of dice they must roll and add up during each round of combat quickly becomes unwieldy. As the Game Master for my group, I often need to roll more than 20 six- or eight-sided dice simultaneously and quickly total them in my head, dozens of times per game session. This task can be particularly challenging for players who struggle with counting pips or keeping track of sums in their heads.

In an effort to tackle this problem, I design a system that can decipher a 2D picture of multiple rolled dice and calculate the sum of the numbers displayed on the dice. The inputs to the system are pictures taken using a cell phone camera. These images are segmented into individual pictures of dice using an object detection and localization algorithm, and the values on the dice in the individual pictures are classified using a CNN.

The object detection and localization portion of this project was done for CS231A: Computer Vision. The CNN classification portion was done for CS230: Deep Learning. The dataset, as well as some processing and utility code, were shared between the two classes.

2 Related work

The problem of automatically detecting dice and computing their value using computer vision appears relatively unexplored. Some attempts have been made to automatically detect dice in the context of casino gambling, but the systems produced by these attempts were extremely rigid (i.e. relying on the user to input the number of dice rolled, and relying on a certain uniform type of die such as a white six-sided die with black pips) [1]. Such systems are for the most part designed to mitigate the need for human supervision in very specific applications, such as a game at a casino. They do not possess the speed and flexibility required to handle different types and quantities of dice.

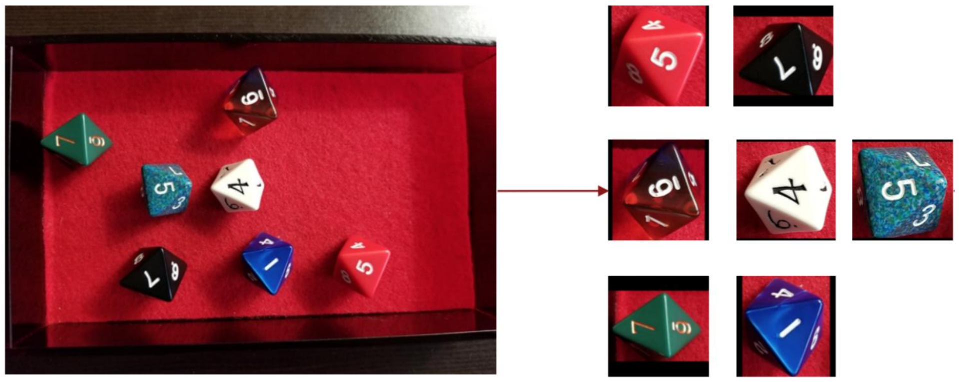


Figure 1: *Left*: a picture from the initial dataset, containing 7 d8s. *Right*: 7 training examples cropped from the picture.

Class	Train	Dev	Test	Class	Train	Dev	Test
0	12.74%	12.43%	14.13%	10	1.59%	1.62%	0.0%
1	12.42%	15.14%	14.13%	11	1.90%	2.16%	4.34%
2	11.54%	12.97%	11.96%	12	0.51%	0.0%	0.0%
3	11.98%	9.73%	10.87%	13	0.19%	0.54%	0.0%
4	9.26%	14.59%	10.87%	14	0.63%	0.0%	0.0%
5	9.77%	5.41%	9.78%	15	0.44%	0.0%	0.0%
6	6.72%	3.78%	5.43%	16	0.44%	0.0%	0.0%
7	6.79%	5.95%	9.78%	17	0.38%	0.0%	0.0%
8	5.77%	5.95%	3.26%	18	0.25%	1.08%	0.0%
9	6.15%	7.57%	4.35%	19	0.51%	1.08%	1.09%

Figure 2: Breakdown of dataset by class. For each example, the class k is equal to the value shown on the die minus 1. For d100s, a 10-sided die which displays multiples of 10, the value is interpreted as $\frac{x}{10}$.

Generally speaking, there are several promising techniques in the field of machine learning that can be applied to this relatively novel task. YOLO9000 is a state-of-the-art object detection and localization system that can be trained on user-supplied images to detect a specific object type; this can be used to extract die positions from images [2].

3 Dataset and Features

I collected an initial dataset of 266 pictures of dice rolled in dice trays. The initial dataset contains 8 different types of die: d4, d6, d8, d10, d12, d20, and d100. After segmentation using YOLO9000, the resulting dataset contains 1855 dice. This dataset is split into 85% training examples, 10% cross-validation, and 5% test.

Die Type	Train	Dev	Test
d4	10.53%	8.11%	11.96%
d6	15.09%	15.68%	14.13%
d8	14.90%	13.51%	14.13%
d10	18.90%	20.54%	15.22%
d12	15.28%	20.0%	17.39%
d20	9.07%	8.11%	7.6%
d100	16.23%	14.05%	19.57%

Figure 3: Breakdown of dataset by die type.

Parameters	Batches Trained	Train Accuracy	Dev Accuracy	Test Accuracy
No BN, $p = 1.0$, $\lambda = 0.0$	50,000	31.14%	12.43%	–
$\epsilon = 0.9$, $p = 1.0$, $\lambda = 0.0$	50,000	39.12%	17.30%	–
$\epsilon = 0.9$, $p = 0.5$, $\lambda = 0.0$	50,000	39.19%	18.92%	–
$\epsilon = 0.9$, $p = 0.5$, $\lambda = 0.01$	50,000	40.46%	23.24%	–
$\epsilon = 0.9$, $p = 0.5$, $\lambda = 0.05$	98,100	56.44%	49.73%	42.39%

Figure 4: Results of experiments in hyperparameter tuning.

Each training example is a 128×128×1 grayscale image. For any examples that are not square, a black border is added to ensure that the width and height are equal. A pixel-wise mean is computed for all examples in the training set and subtracted from each image; the same mean is subtracted from each example in the cross-validation and test sets.

The dataset is augmented by rotating each training example 90°, 180°, and 270°. Four random crops are taken from each of the rotated and original images; these crops are always square, roughly centered, and are no smaller than 60% and no larger than 120% of the original image size. All crops are scaled back to 128×128×1.

After augmentation, the training set contains 31,540 examples. The cross-validation and test sets consist of 185 and 92 images, respectively, that have undergone the same processing procedure, but no augmentation.

4 Methods

I use a CNN to perform softmax classification on each training example, using a multi-class cross entropy loss function with L2 regularization loss:

$$\mathcal{L}(y, \hat{y}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{j=1}^{k} y_j \log \hat{y}_j + \lambda ||W||_2^2$$

The CNN architecture is detailed below:

1. Convolutional layer. 16 5×5 filters with stride 2. Batch normalization (ϵ), ReLU activation, and max-pooling with stride 2.
2. Convolutional layer. 32 5×5 filters with stride 2. Batch normalization (ϵ), ReLU activation, and max-pooling with stride 2.
3. Convolutional layer. 64 5×5 filters with stride 2. Batch normalization (ϵ), ReLU activation, and max-pooling with stride 2.
4. Fully connected layer. 512 hidden units. Batch normalization (ϵ), dropout (p), and ReLU activation.
5. Fully connected output layer. $k = 20$ units. Softmax activation.

5 Results

I experimented with running classification with and without batch normalization, dropout, and L2 regularization. For all experiments, I began with a learning rate of $\alpha = 0.001$ and decreased it to $\alpha = 0.0001$ after training on about 50,000 batches; through experimentation, this seemed to help learning while keeping variance low. The minibatch size was 16 and batches were randomly sampled without replacement, so that the model trained on the entire training set.

I used accuracy on the validation set as my metric. Every 1,000 batches, I evaluated accuracy on the validation set. If it was higher than the previous highest validation accuracy, the weights were checkpointed. Every 10,000 batches, the weights were re-initialized from the last checkpoint and the learning rate was tuned if needed. This helped reduce overfitting.

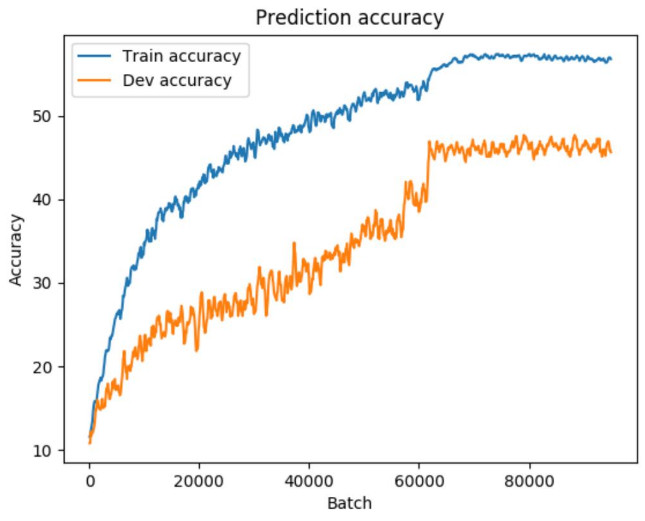


Figure 5: Accuracy of final model on train and validation sets throughout training. Values are averaged over the last 500 batches for a smoother curve.

The results of my experiments are listed in Figure 4. The highest dev set accuracy I was able to achieve was 49.73% after 98,100 epochs, using a batch normalization decay rate of $\epsilon = 0.9$, dropout keep probability of $p = 0.5$, and L2 regularization weight parameter of $\lambda = 0.05$. This model achieved a test accuracy of 42.39%.

6 Conclusion

Examining the confusion matrix from evaluation on the test set (Figure 5) provides some interesting insights. For example, in the test set, the model always mislabels dice with the value 20 as showing the value 10 instead. This is likely because the 10 class has much more support in the training set (6.15%) than the 20 class (0.51%) (see Figure 2), and since they both consist of a single character before a 0, the model assigns higher probability to the more likely class (10). Another common error the model makes is mistaking a 10 for a 1; the reason for confusion in this case is obvious. The model also mislabels dice with the value 1 as showing the value 7 in 23% of test set examples - an understandable error, considering I sometimes have trouble telling the two numbers apart on my own dice.

While the train and dev set confusion matrices are not presented in the main body of this paper for the sake of conciseness (see the Appendices if you're curious), they tell a similar tale. The model most accurately identifies values with high support in the training set: in general, the lower values. It often mis-identifies higher values as the lower value with which they share a digit (for example, identifying a 19 as a 9). These findings suggest that a promising method of improving the model's accuracy is simply to introduce more of these examples into the training set. Because high values only appear on the d12 and d20, this will require adding more examples of these types of dice.

It is also likely the case that human-level performance in this task is not 100%. I had difficulty labeling some examples in the dataset, particularly pictures of d20s where it was not clear which side was facing up (see Figure 7 for an example). Certain die types, such as the d20, are much more difficult to classify than others due to the number of faces that are showing in any given image. The breakdown of accuracy by die type (see the Appendices) supports this conclusion. Fortunately, in my experience playing Pathfinder, it is rarely the case that one must roll multiple d20s at the same time and add the results together; I only added d20s to the dataset for the sake of completeness. In future work, it may be best to remove d20s from the dataset altogether to reduce human error in the ground truth labels.

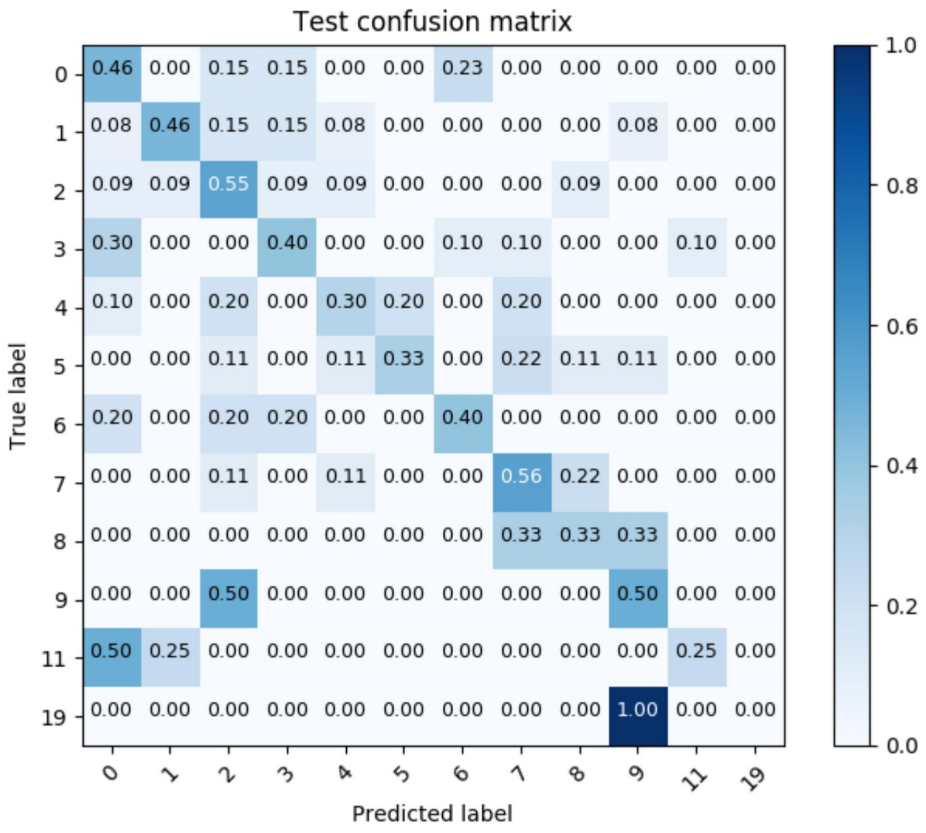


Figure 6: Normalized confusion matrix on the test set. (Labels correspond to the value shown on the die minus 1, so a die with the label 5 is actually displaying a 6, etc.) Not all classes are present in the test set, since test set examples were randomly selected.

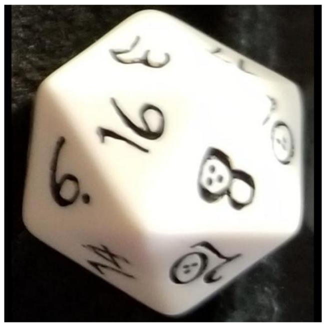


Figure 7: An example of an image where the ground truth (a.k.a. my own hand-labeling) may not be accurate due to the ambiguity of the die's orientation.

Finally, the gap between training and dev set accuracy in the final model suggests that the model is still having problems with overfitting, despite having bias issues as well. Increasing the amount of regularization may help, but not until the bias problem is corrected; it is evident from the evaluation curve in Figure 5 that the model currently plateaus at sub-60% accuracy on the training set, in part due to the challenges discussed previously (i.e. dataset imbalance and human error in labeling).

References

[1] Kuo-Yi Huang. An auto-recognizing system for dice games using a modified unsupervised grey clustering algorithm. *Sensors (Basel)*, 2008.

[2] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

A Code

All of the code for this project is available in a public repository on Github.

B Figures

Die Type	Train Accuracy	Dev Accuracy	Test Accuracy
d4	56.04%	33.33%	54.55%
d6	73.11%	65.52%	53.85%
d8	65.96%	72.0%	46.15%
d10	61.07%	57.89%	50.0%
d12	52.70%	48.65%	37.50%
d20	37.76%	13.33%	14.29%
d100	41.02%	26.92%	33.33%

Figure 8: Breakdown of accuracy by die type.

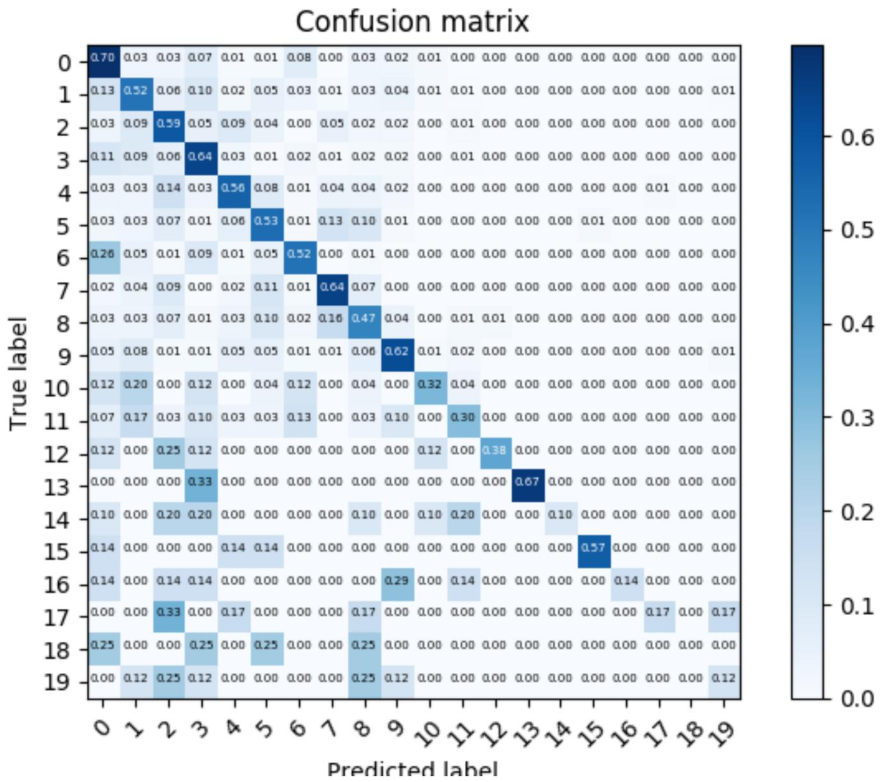
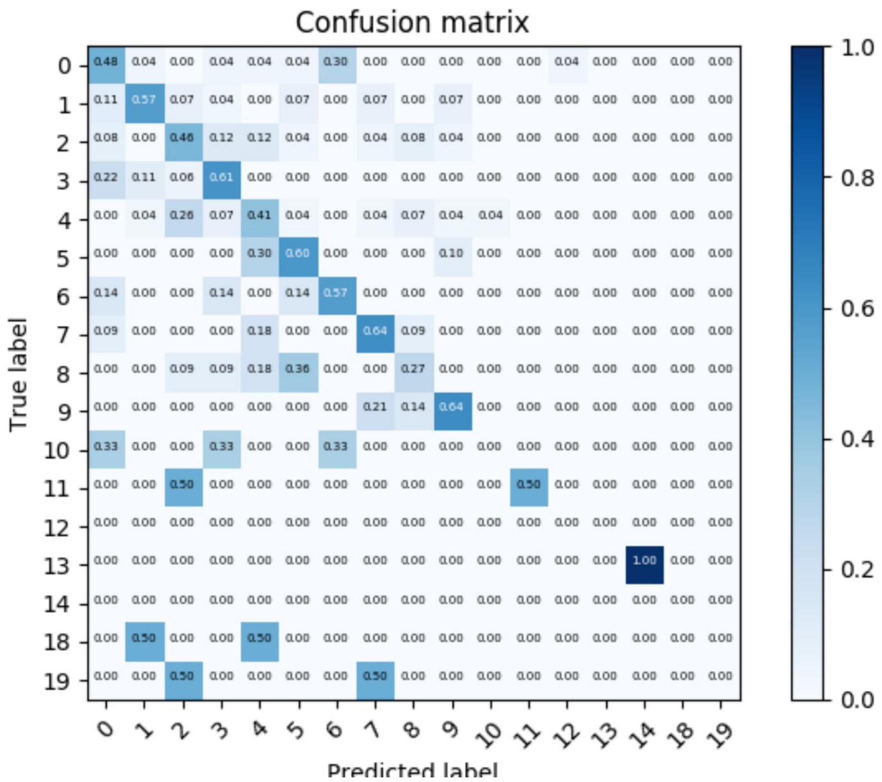


Figure 9: Normalized confusion matrix on the train set.



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	18	19
0	0.48	0.04	0.00	0.04	0.04	0.04	0.30	0.00	0.00	0.00	0.00	0.00	0.04	0.00	0.00	0.00	0.00
1	0.11	0.57	0.07	0.04	0.00	0.07	0.00	0.07	0.00	0.07	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.08	0.00	0.46	0.12	0.12	0.04	0.00	0.04	0.08	0.04	0.00	0.00	0.00	0.00	0.00	0.00	0.00
3	0.22	0.11	0.06	0.61	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.04	0.26	0.07	0.41	0.04	0.00	0.04	0.07	0.04	0.04	0.00	0.00	0.00	0.00	0.00	0.00
5	0.00	0.00	0.00	0.00	0.30	0.60	0.00	0.00	0.00	0.10	0.00	0.00	0.00	0.00	0.00	0.00	0.00
6	0.14	0.00	0.00	0.14	0.00	0.14	0.57	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
7	0.09	0.00	0.00	0.00	0.18	0.00	0.00	0.64	0.09	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
8	0.00	0.00	0.09	0.09	0.18	0.36	0.00	0.00	0.27	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.21	0.14	0.64	0.00	0.00	0.00	0.00	0.00	0.00	0.00
10	0.33	0.00	0.00	0.33	0.00	0.00	0.33	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
11	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00
12	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
13	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00
14	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
18	0.00	0.50	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
19	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.50	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00

Figure 10: Normalized confusion matrix on the dev set.