Exploring Boosted Neural Nets for Rubik's Cube Solving

Alexander Irpan[*]
Department of Computer Science
University of California, Berkeley
`alexirpan@berkeley.edu`

Abstract

We explore whether neural nets can learn good actions in the Rubik's Cube setting when given minimal domain knowledge. In doing so, we experiment with integrating neural nets with AdaBoost to dynamically weight important instances from the data set at training time. Inspired by FilterBoost, we develop a variant of AdaBoost that is better suited to problem settings with a sampling oracle, while making fewer calls to that oracle. We then test that variant using neural nets as a weak learner. Experiments demonstrate that neural nets can acheive some success at Rubik's Cube solving with minimal domain knowledge, but our variant of AdaBoost does not work well with the current trend towards larger, deeper neural nets.

1 Introduction

Historically, the Rubik's Cube has been used as a test bed for measuring the performance of search algorithms. This culminated in the development of Kociemba's Two Phase Algorithm, which uses two phases of iteratively deepened A* search to first find a path to a subgroup of cube states, then a path from that subgroup to a solved state [4]. In practice, this algorithm gives near optimal solutions. Exhaustive computer search has shown every Rubik's Cube is solvable in 20 moves, or in 26 face turns if the quarter-turn metric is used [7].

Recently, there has been a trend of using neural nets trained offline to imitate the performance of online search. The appeal of this approach is that search algorithms often requiring starting computation from scratch for each input state. A neural net that outputs the same action as a search algorithm can allow for significant computation gains at evaluation time. Such neural net controllers have been trained for Atari games [6] and robotics control [5]. As demonstrated by AlphaGo, neural net controllers can also be used to augment search algorithms to further improve their performance [8]. These controllers have mostly used convolutional neural nets, although there is some progress on neural net controllers that use memory [10].

We chose to work in the Rubik's Cube domain because it is very cheap to simulate more data. This domain gives faster turnaround time on experiment, and lets us easily tweak both the problem difficulty and amount of available data.

2 Problem formulation

In human developed solving algorithms, there are common sequences of moves that are composed in different ways to apply the desired permutation to cube pieces. This suggests that a Rubik's Cube solver should exploit the temporal nature of the problem, and use the most recent moves to decide on next actions.

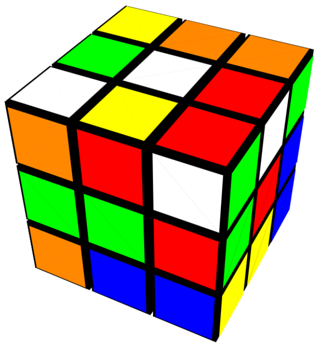
[*]Personal site at www.alexirpan.com

Submitted to 29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

Motivated by this, we model solving a Rubik's Cube as a sequence to sequence learning problem. First, we generate random episodes of length K. This is done by taking a solved cube, applying K random moves to get a sequence of K cube states and K moves, then reversing the episode. This gives a random starting cube and a sequence of K moves that solves the randomly generated cube.

2.1 Baseline methods

Before trying to use boosting algorithms with neural nets, we ran experiments to decide on the best neural net architecture for the weak learner. All architectures tested were trained on the task of predicting the correct move sequence when given a sequence of cube states. To encode each cube state, we use a one-hot encoding of sticker colors. There are 54 stickers, and each can be one of 6 colors. Indicator variable $X_{i,j}$ is 1 if the ith sticker is the jth color, and 0 otherwise. This gives a total of $54 \times 6 = 324$ features.



$$\longrightarrow \quad \text{6 colors}\left\{ \overbrace{\begin{pmatrix} 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & \cdots \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots \end{pmatrix}}^{\text{54 stickers}} \right.$$

Figure 1: Sample one-hot encoding of cube state

An alternative representation we did not try was representing the cube by the position and orientation of the 8 corner cubelets and 12 edge cubelets. The sticker-based representation was easier to implement, and initial results were promising enough to continue further work. Nevertheless, a cubelet representation may be worth pursuing.

Every neural net architecture used ends with 12 outputs, corresponding to the 12 possible face turns - each of the 6 faces can be rotated clockwise or counterclockwise.

Different architecture choices (fully connected, recurrent, etc.) were tested while keeping the number of parameters in the model constant. The best results were acheived with LSTMs. Full results are in Section 5.

3 Boosting

Boosting algorithms are a general class of ensemble methods that convert several weak learners into a single strong one. They have theoretical guarantees on convergence rates for error, but occasionally fail in practice.

We start by reviewing boosting algorithms from the literature, before proposing and testing our own variant. For the rest of this paper, f_t denotes the weak learner for the tth iteration, D_t is the distribution over data that f_t is trained on, and F_t is the strong learner at the end of the tth iteration. We also use standard classification notation, where X is the data space and Y is the label space.

3.1 Multiclass AdaBoost

AdaBoost is an algorithm developed by Freund and Schapire. Notably, it adaptively adjusts the weak learners to better classify data misclassified by previous weak learners. It is a batch booster, meaning all data is given before training begins.

There are multiple variants of AdaBoost, but for our purposes we use the AdaBoost.M2 variant [3]. AdaBoost.M2 expects the weak learner to output its confidence in each class. In other words, every weak learner is some $f : (X, Y) \to [0, 1]$, where X and Y are the data and labels respectively. Neural nets output a probability distribution over their outputs, so we have a natural measure of confidence in each label. As an aside, AdaBoost.M2 does not require the output to be a probability distribution, it only requires the confidence to be in $[0, 1]$.

The learning algorithm is detailed in [3]. It first reduces multiclass classification into several instances of two-class classification. Every instance (x_i, y_i) is transformed into $|Y| - 1$ tuples $\{((x_i, y_i), y)\}$, one for every wrong y. These are known as *mislabelings*. AdaBoost.M2 expectes the weak learner to discrimate the correct label y_i from some incorrect label y. For a given mislabeling, the pseudoloss f achieves is

$$\text{ploss}(f, (x_i, y_i), y) = (1/2)(1 - f(x_i, y_i) - f(x_i, y))$$

Note that a f that outputs constant confidence over all labels achieves pseudoloss $1/2$. This corresponds with a neural net whose output is a uniform distribution over all labels.

To converge, the weak learner must outperform a random guesser over any distribution D of mislabelings, or more formally, $\mathbb{E}_D[\text{ploss}(f, \cdot, \cdot)] < \frac{1}{2}$.

Algorithm 1 AdaBoost.M2

```
Input: Dataset size $n$, oracle $\text{Sample}()$
 1: Init $\{(x_i, y_i)\}$ with $n$ calls to $\text{Sample}()$
 2: For $i = 1, \dots, n, y \in Y \setminus \{y_i\}$, init $D_1(i, y)$ uniformly.
 3: for $t = 1$ to $T$ do
 4:     Train weak learner $f_t$ with distribution $D_t(i, y)$
 5:     $\epsilon_t \leftarrow \frac{1}{n} \sum_{(i,y)} D_t(i, y) \text{ploss}(f_t, (x_i, y_i), y)$
 6:     $\alpha_t \leftarrow \log \frac{1-\epsilon_t}{\epsilon_t}$
 7:     $F_t \leftarrow \frac{1}{Z_t} \sum_{s=1}^{t} \alpha_s f_s$, where $Z_t$ is a normalization constant
 8:     for $i = 1$ to $n$ do
 9:         $D_{t+1}(i, y) \propto D_t(i, y) e^{\alpha_t (1 + \text{ploss}(f_t, (x_i, y_i), y))}$
        Normalize to make $\sum_{(i,y)} D_{t+1}(i, y) = 1$
    return $F_T$
```

This has a corresponding convergence theorem.

Theorem 1 *Let ϵ_t be the error for the weak classifier at timestep t. Then*

$$\mathbb{E}[\text{0/1 loss of } F_T] \leq (|Y| - 1) \prod_{t=1}^{T} 2\sqrt{\epsilon_t(1 - \epsilon_t)}$$

where this expectation is over the empirical distribution of sampled data.

This can be proved by a reduction to the two class case, and we refer to the original paper for details [3].

3.2 Dealing with a sampling oracle

The Rubik's Cube domain has some interesting properties that influence the boosting algorithm we should use. Unlike many supervised learning problems, we have access to an arbitrary amount of labelled data which can be generated on demand. In essence, this problem has a *sampling oracle*, which can be queried to receive an episode sampled from some fixed distribution. This is known as the boosting-by-filtering setting.

FilterBoost is a boosting algorithm developed for the filtering setting. Informally, at time t FilterBoost trains a weak learner, then uses the sampling oracle to generate data for time $t + 1$. Samples from the oracle are rejected with probability based on the error of the strong learner so far, in such a way that generated samples come from the distribution D_t we want the weak learner to optimize over [2].

We did some experiments with FilterBoost, but ultimately decided it was not suitable for boosting large neural nets. The strong learner is an ensemble of deep neural nets, which makes evaluating it computationally expensive. Although sampling new episodes is cheap, evaluating the strong learner on every newly sampled episode was not worth the computation time.

These experiments motivated the development of a new AdaBoost variant.

3.3 Refreshed AdaBoost

In this subsection, we first present the motivation and derivations for the algorithm design, then present the pseudocode itself. As an aside, we call it refreshed AdaBoost to distinguish from AdaBoost where the dataset is resampled from distribution D_t every iteration. We assume the weak learner can be trained on weighted instances, which is true for neural nets.

We would like to sample new data for each weak learner, but at the same time we do not want to replace all the data in the dataset for computational efficiency. Part of AdaBoost's design is that the weight of each instance is multiplicative - that is,

$$D_{t+1}(i,y) \propto D_t(i,y)\exp(\alpha_t(1+\mathrm{ploss}(f_t,(x_i,y_i),y)))$$

Reweighting existing data only requires evaluating the weak learner, while weighting new data requires evaluating the entire ensemble. This makes existing data cheap and new data expensive.

The algorithm should balance decreasing generalization error by generating new samples and keeping computation time low by keeping old samples. There may be ways to do this adaptively, but for now this is done through a hyperparameter p. At each iteration, an instance is kept with probability p. New instances are then sampled to replace any instances lost.

By applying this iteratively, we get a closed solution for $D_{t+1}(i,y)$.

$$\begin{aligned}
D_{t+1}(i,y) &\propto \prod_{s=1}^{t} \exp(\alpha_s(1+\mathrm{ploss}(f_s,(x_i,y_i),y))) \\
&= \exp(\sum_{s=1}^{t} \alpha_s \,\mathrm{ploss}(f_s,(x_i,y_i),y) + \sum_{s=1}^{t} \alpha_s) \\
&\propto \exp(\sum_{s=1}^{t} \alpha_s \,\mathrm{ploss}(f_s,(x_i,y_i),y))
\end{aligned}$$

Algorithm 2 is the same as AdaBoost.M2, except for added lines 10-15. This is the refreshing step that adds new samples to the dataset. New data is sampled from the oracle, then weighted to match distribution D_t. At the end of each iteration, two invariants hold: the number of samples in the dataset is n, and the total weight of the dataset is n.

The same convergence bound from the batch version of AdaBoost.M2 still holds, because at the end of resampling all data is still sampled from distribution D_{t+1}. To see why, note the dataset at time $t+1$ is essentially formed by combining the two parts below.

- Generate $n-m$ samples, weighted based on the strong learner.
- Generate m samples, weighted based on the strong learner.

The only difference is that the first part, the part based on existing data, can be computed with clever shortcuts.

Thus, the same convergence theorem holds.

$$\mathbb{E}[\text{0/1 loss of } F_T] \leq (|Y|-1)\prod_{t=1}^{T} 2\sqrt{\epsilon_t(1-\epsilon_t)}$$

Importantly, this expectation is only over the empirical distribution *at time* T, not over all samples ever seen in the dataset. In the analytical bound, we still have the same generalization error as the batch version of AdaBoost, but in practice seeing more data during training reduces test error.

Algorithm 2 Refreshed AdaBoost

```
Input: Dataset size $n$, oracle $\mathrm{Sample}()$, $p \in [0,1]$
 1: Init $\{(x_i, y_i)\}$ with $n$ calls to $\mathrm{Sample}()$
 2: For $i = 1, \ldots, n, y \in Y \setminus \{y_i\}$, $D_1(i,y) = \frac{1}{|Y|-1}$
 3: for $t = 1$ to $T$ do
 4:     Train weak learner $f_t$ with distribution $D_t(i,y)$
 5:     $\epsilon_t \leftarrow \frac{1}{n} \sum_{(i,y)} D_t(i,y)\,\mathrm{ploss}(f_t, (x_i, y_i), y)$
 6:     $\alpha_t \leftarrow \log \frac{1-\epsilon_t}{\epsilon_t}$
 7:     $F_t \leftarrow \frac{1}{Z_t} \sum_{s=1}^{t} \alpha_s f_s$, where $Z_t$ is a normalization constant
 8:     for $i = 1$ to $n$ do
 9:         $D_{t+1}(i,y) \propto D_t(i,y) e^{\alpha_t (1+\mathrm{ploss}(f_t,(x_i,y_i),y))}$
        Normalize to make $\sum_{(i,y)} D_{t+1}(i,y) = n$
10:     for $i = 1$ to $n$ do
11:         Keep sample $(x_i, y_i)$ with probability $p$.
12:     $m \leftarrow$ num samples lost.
13:     Normalize such that total weight of kept data is $n - m$.
14:     Call $\mathrm{Sample}()$ $m$ times to get new $(x_j, y_j)$
15:     $D_{t+1}(j,y) \propto e^{\sum_{s=1}^{t} \alpha_s \,\mathrm{ploss}(f_s,(x_j,y_j),y)}$
16:     Normalize such that total weight of new data is $m$.
    return $F_T$
```

4 Neural net training and boosting

For efficiency reasons, it is too expensive to retrain a neural net from scratch each iteration. Instead, we do the following.

- Initialize a neural net at time $t = 0$.
- For each t, iterate through the current weighted dataset once, updating with stochastic gradient descent.
- Save a copy of the neural net at time t for the boosted classifier F_t.
- Reweight and refresh data, then continue training from the current neural net parameters.

Note that even though the weak learner (the neural net) has heavy dependence on previous timesteps, the convergence bound analysis still holds, and new weak learners are still guided towards classifying high-error examples well. None of the proof requires independence between the weak learners, and as long as each weak learner acheives small enough error, the strong learner's error will still converge.

For SGD, the core idea is that we can treat a weighted datapoint (x_i, y_i) as having a fractional count of (x_i, y_i) instances. If a minibatch contains (x, y) twice, and we run SGD on that minibatch, the gradient is effectively applied twice. We could achieve the same gradient update by multipying the gradient by 2 before applying the update.

This has a natural extension to fractional weights. If the weight of (x_1, y_1) is 1.5, and the weight of (x_2, y_2) is 0.5, we effectively have 1.5 instances of (x_1, y_1) and 0.5 instances of (x_2, y_2). The gradient update will be 1.5 of the gradient for (x_1, y_1) and 0.5 of the gradient for (x_2, y_2)

Initially, the dataset is n examples all with weight 1, so the total weight is n. At each iteration, we normalize such the the dataset continues to have total weight n, then multiply the gradient for each (x_i, y_i) by that weight. Normalizing total weight to n ensures we apply n "gradients" each timestep.

5 Experiments

All code was written in Torch. Initial architecture experiments were done before applying boosting. We experimented with a one hidden layer fully connected model, a two hidden layer fully connected model, a one hidden layer RNN, and a one hidden layer LSTM. Architectures were tested both on classification accuracy, and on the solve percentage of the argmax policy of the neural net output.

Neural nets were trained on episodes of length K, then tested on episodes of length $K + 1$. The LSTM architecture performed best.

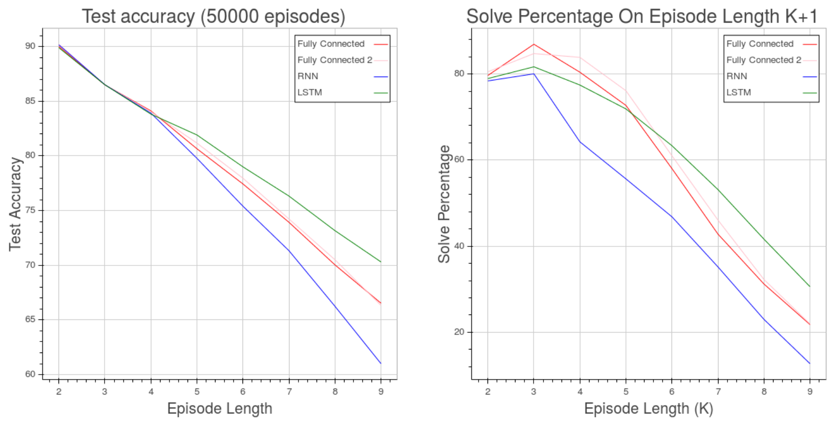


Figure 2: Classification acc. and solve % of architecture tests.

The final architecture used is an LSTM with 100 hidden units. We tested with and without boosting, with $p = 0.8$ and $p = 0$. All methods were run for 20 epochs with $n = 50000$ episodes in the dataset. Episode length used was $K = 9$; although episodes of length $K = 26$ are required to see every possible cube state, prior experiments showed $K = 9$ achieved similar performance with significantly less training time. Each was trained with a learning rate of 0.1.

Overall, we obtained resounding negative results. Refreshed AdaBoost had worse classification accuracy than a baseline SGD method that did no boosting. Furthermore, refreshed AdaBoost took longer to train. Solve percentages were generated by running the argmax policy on 10000 random cube states, halting on solve or on failure to solve after 50 actions. The baseline method continued to have better solve percentages across all episode lengths.

Table 1: Experiment results

Method	**Accuracy**	**Run Time (min)**
Baseline	72.26%	237
Boosted, $p = 0.8$	67.56%	366
Boosted, $p = 0$	67.91%	439

6 Analysis

We have some conjectures for why boosting failed to improve performance. We conjecture that boosting works better when weak learners are cheap to train and evaluate. This allows us to generate an ensemble of many hundreds or thousands of learners, instead of tens. Additionally, weak learners should disagree on the optimal action when given the same input.

Large neural nets are not cheap to train, so we do not have the time to train many weak learners - in testing, we only had time to train 20 weak learners. Furthermore, because all gradient updates were applied to the same neural net, the neural net was regularized against changing too much each iteration. Upon manual inspection, the weak learners almost always agree on the optimal action. We were unable to find an input on which the argmax action differed (although the confidences varied between learners.)

Theoretically, a boosting approach will work with enough iterations, but boosting large neural nets appears to be less efficient than simply feeding all data into a single neural net.

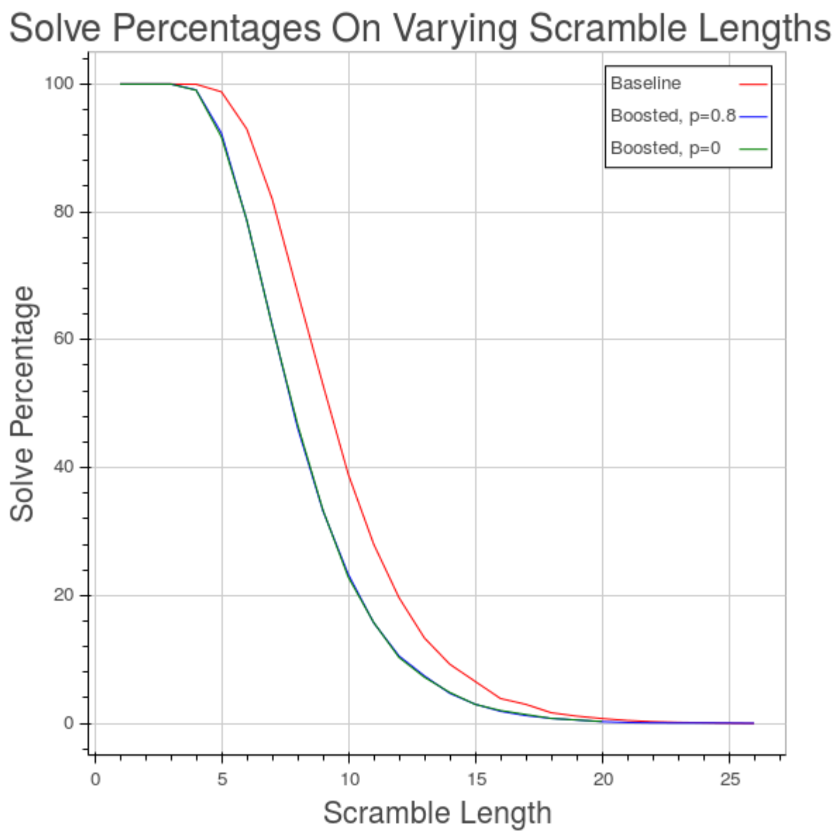


Figure 3: Solve percentages of boosted and unboosted neural nets.

7 Further work

Although we did not achieve good performance with boosting large neural nets, it is worth testing whether the refreshed AdaBoost algorithm works with other weak learners, like decision stumps.

More specifically to Rubik's Cube solving, the current approach does not deal well with fuzziness in episode labels. A given cube has many possible solutions, and reducing the problem to classification assumes there is a single fixed solution. Further work will need to address this issue. One option we propose is to verify outputted sequences during training time, updating the loss and labels appropriately if the output is a valid solution.

Reinforcement learning seems very well suited to the Rubik's Cube problem. The sample inefficiency of RL is less of a concern here due to very fast simulation time, and much like prior work the neural net policy can be further improved with policy gradient methods [8]. Prior work showing replaying previously sampled actions with high error decreases training time could be of use [9].

Finally, curriculum learning could be applicable [1]. Tweaking episode length K during training lets us increase the difficulty of the problem over time. Doing so could lead to less training time.

8 Conclusion

We find negative results for combining our proposed AdaBoost variant with large neural nets. Although boosting methods are useful, they do not appear to mix well with a hammer as general and powerful as deep neural nets. They seem better suited when the weak learner is truly a weak learner.

References

[1] Bengio, Yoshua, et al. "Curriculum learning." Proceedings of the 26th annual international conference on machine learning. ACM, 2009.

[2] Bradley, Joseph K. "FilterBoost: Regression and Classification on Large Datasets." 2007.

[3] Freund, Yoav, and Robert E. Schapire. "A decision-theoretic generalization of on-line learning and an application to boosting." 1995.

[4] Kociemba, Herbert. "Two-Phase Algorithm Details." 2014. Web. 27 Apr. 2016.

[5] Levine, Sergey, et al. "End-to-end training of deep visuomotor policies." arXiv preprint arXiv:1504.00702 (2015).

[6] Mnih, Volodymyr, et al. "Playing atari with deep reinforcement learning." arXiv preprint arXiv:1312.5602 (2013).

[7] Rokicki, Tomas, Herbert Kociemba, Morley Davidson, and John Dethridge. "God's Number Is 20." God's Number Is 20. N.p., Aug. 2014. Web. 29 Apr. 2016.

[8] Silver, David, et al. "Mastering the game of Go with deep neural networks and tree search." Nature 529.7587 (2016): 484-489.

[9] Schaul, Tom, et al. "Prioritized Experience Replay." arXiv preprint arXiv:1511.05952 (2015) .

[10] Zhang, Marvin, et al. "Policy learning with continuous memory states for partially observed robotic control." arXiv preprint arXiv:1507.01273 (2015).