Exploring Boosted Recurrent Neural Nets For Rubik's Cube Solving

Alex Irpan

May 1, 2016

1 Abstract

We explore integrating AdaBoost with neural net training. AdaBoost focuses training on difficult examples, which could train neural nets faster than uniform weighting. It also gives a natural ensemble for further boosting performance of the learned network. We examine whether AdaBoost achieves these results, using the problem domain of solving a Rubik's Cube.

2 Background

Historically, the Rubik's Cube has been a good test bed for measuring the performance of search algorithms. The best search algorithm is Kociemba's Two Phase Algorithm, which uses iteratively deepened A* search guided by domain knowledge [3]. Exhaustive computer search has shown every Rubik's Cube is solvable in either 20 moves or 26 face turns [6].

We choose to use a Rubik's Cube because it is very cheap to simulate. This lets us easily tweak both the problem difficulty and the amount of available data.

Prior work on neural net controllers has been successful in Atari [5], Go [7], and robotics control [4]. Much of this work uses convolutional neural nets, although there is some progress on neural net controllers with memory [9].

3 Problem Formulation

Following in the footsteps of AlphaGo, we reduce Rubik's cube solving to a supervised learning problem. we treat solving a Rubik's Cube as a sequence to sequence learning problem. To generate training sequences, let K be the desired sequence length. From a solved cube, apply K random moves. Reversing the outputted states and taking the inverse of the applied moves gives a sequence of K cube states and a sequence of K moves that solves the randomly generated cube. We train a neural net to output a move sequence when given a cube sequence.

We use a one-hot encoding of sticker colors. Each of the 6 colors is represented by a basis vector of $\mathbb{R}^6$. There are 45 stickers, giving $45 \times 6 = 324$ features in total.

An alternative representation we did not try was representing the position and orientation of each cubelet. The sticker-based approach was easier to implement, and initial results were promising enough to continue trying it. Nevertheless, alternate representations will likely have lower dimension, and are worth pursuing.

The last layer of the neural net has 12 outputs, corresponding to the 12 moves for rotating one of the 6 faces clockwise or counterclockwise.

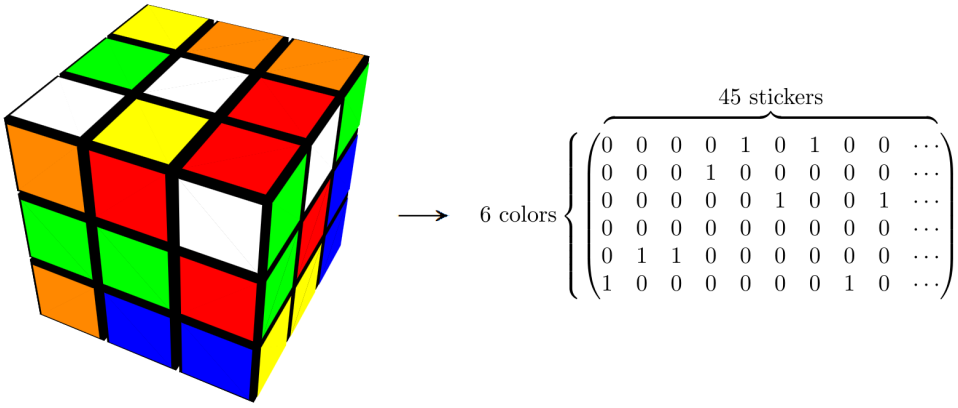

Figure 1: Sample One-Hot Encoding of Cube State

4 Unique Properties of Domain

The Rubik's Cube domain has some interesting properites. Unlike many supervised learning problems, we have access to an arbitrary amount of labelled data. Furthermore, by tweaking K we can change how difficult the classification problem is. Furthermore, it is very easy to simulate moves on a Rubik's Cube. In essence, this domain has a *sampling oracle*, which can be queried to generate more data in negligible time.

This motivates us to look towards boosters designed for the filtering setting. In this setting, boosters sample data from the oracle, and reject samples based on the error of the current boosted classifier [1]. Done appropriately, the accepted samples will come from the desired distribution D_t.

5 Boosting

5.1 FilterBoost

FilterBoost is an algorithm for the filtering setting that achieves similar results to AdaBoost. It does not run any faster than AdaBoost, but because while sampling only as many data points as needed to train the weak learner to sufficient accuracy [1].

Like AdaBoost, on each iteration t we train a weak learner against a distribution D_t that depends on the error of the previous weak learners. However, the sampling oracle gives samples from an unknown distribution P. To get samples from D_t, we run the boosted classifier F_t on samples from P, and keep samples with probability based on the error of F_t.

We did some experiments with FilterBoost, but ultimately decided it was not suitable for our use case. Recall that our weak learner is a neural net, which makes the boosted classifier F_t an ensemble of neural nets. Sampling new data (x, y) is cheap, but evaluating the ensemble of neural nets F_t on every new sample is too expensive to make experiments feasible.

5.2 Multiclass AdaBoost

AdaBoost has multiple extensions to multiclass classification, but the most natural variant is the AdaBoost.M2 variant. In this variant, the weak learner $f : (X, Y) \to [0, 1]$ outputs its confidence in each class [2]. Neural nets naturally output a probability distribution over outputs, so this fits

well. 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 AdaBoost.M2 extension turns (x_i, y_i) into $|Y| - 1$ mislabelings $\{((x_i, y_i), y)\}$, one for every wrong y. AdaBoost then asks the weak learner f to discriminate y_i from y. This reduces the problem to the two class class. For the loss, we define the pseudoloss as.

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

Note that a constant f achieves pseudoloss $1/2$. To converge, the weak learner must outperform random guessing over any distribution D of mislabelings. Formally, $\mathbb{E}_D[\text{ploss}(f, \cdot, \cdot)] < \frac{1}{2}$.

Algorithm 1 AdaBoost.M2

Input: Dataset size n, oracle Sample()

1: Init $\{(x_i, y_i)\}$ with n calls to Sample()
2: For $i = 1, \ldots, n, y \in Y \setminus \{y_i\}$, 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*

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

This can be proved by a reduction to the two class case. See the original paper for details [2].

5.3 Refreshed AdaBoost

AdaBoost is a well-tested method, but again it is not suitable for our use case. It assumes a fixed dataset, so it cannot use the full advantage a sampling oracle gives us.

This motivates the modification we call *refreshed AdaBoost*. In refreshed AdaBoost, a random portion of the dataset is thrown out and replaced with new samples every iteration. The new samples are then weighted the way they would be if they had been in the dataset from the beginning, in a manner similar to FilterBoost. To balance the cost of evaluating new weights, we add a refresh parameter p, which controls how often data is resampled.

The algorithm below is the same as AdaBoost.M2, except for lines 10-15, which add a refreshing step each iteration. Additionally, instead of normalizing weights D_t to sum to 1, we normalize it to sum to n, the number of samples. Why we do this will be more clear in the next section.

Algorithm 2 Refreshed AdaBoost

Input: Dataset size n, oracle Sample(), $p \in [0, 1]$

Init $\{(x_i, y_i)\}$ with n calls to Sample()
For $i = 1, \ldots, n, y \in Y \setminus \{y_i\}$, $D_1(i, y) = \frac{1}{|Y|-1}$
for $t = 1$ to T **do**
    Train weak learner f_t with distribution $D_t(i, y)$
    $\epsilon_t \leftarrow \frac{1}{n} \sum_{(i,y)} D_t(i, y) \operatorname{ploss}(f_t, (x_i, y_i), y)$
    $\alpha_t \leftarrow \log \frac{1-\epsilon_t}{\epsilon_t}$
    $F_t \leftarrow \frac{1}{Z_t} \sum_{s=1}^{t} \alpha_s f_s$, where Z_t is a normalization constant
    for $i = 1$ to n **do**
        $D_{t+1}(i, y) \propto D_t(i, y) e^{\alpha_t(1+\operatorname{ploss}(f_t,(x_i,y_i),y))}$
    Normalize to make $\sum_{(i,y)} D_{t+1}(i, y) = n$
    for $i = 1$ to n **do**
        Keep sample (x_i, y_i) with probability p.
    $m \leftarrow$ num samples lost, $W \leftarrow$ weight lost
    Call Sample() m times to get new (x_j, y_j)
    $D_{t+1}(j, y) \propto e^{\sum_{s=1}^{t} \alpha_s \operatorname{ploss}(f_s,(x_j,y_j),y)}$
    Normalize to make $\sum_{(j,y)} D_{t+1}(j, y) = W$
return F_T

First, we argue why $D_{t+1}(j, y) \propto e^{\operatorname{ploss}(F_t,(x_j,y_j),y)+\sum_{s=1}^{t} \alpha_s}$. The weights from AdaBoost.M2 are defined exponentially, with

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

Applying this iteratively to get a closed solution to $D_{t+1}(i, y)$. Note that since we only want something proportional, we can cancel $\exp(c)$ for constants c that do not depend on i or y.

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

By weighting new samples (x_j, y_j) in this way, we replace the removed data points with samples from the same distribution D_{t+1}. Thus, in expectation the same convergence theorem should hold.

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

6 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, none of the convergence bound analysis requires independence between weak learners, so all theorems still hold.

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.

7 Experiments

Initial experiments were done to determine which type of architecture worked best. Architectures were tested both on classification accuracy, and on the solve percentage of the argmax policy, defined by applying the move with largest output weight.

We trained a fully connected net, a deeper fully connected net, a recurrent neural net, and an LSTM on varying episode lengths. When normalized to have the same number of parameters, the LSTM architecture performed best.

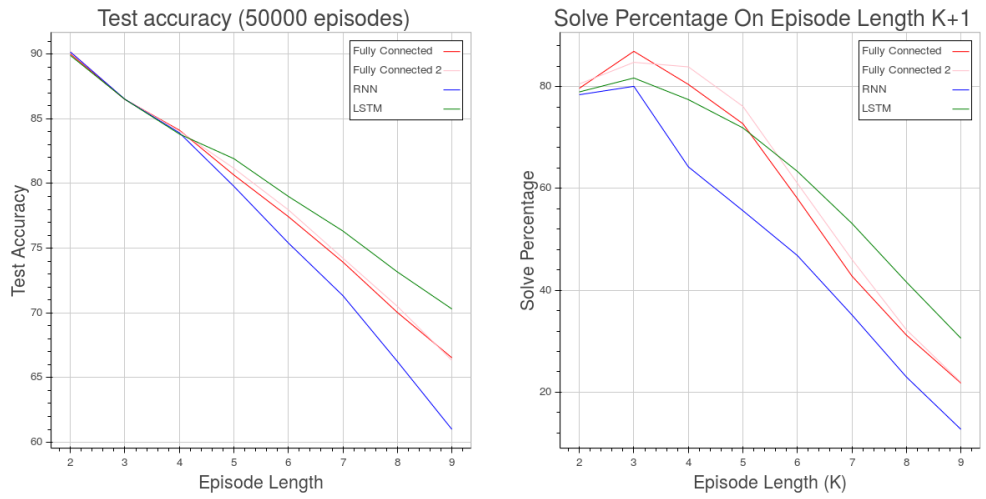

Figure 2: Classification Acc. and Solve % of Varying Architectures

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

Table 1: Experiment Results

The final architecture used is an LSTM with 100 hidden units. Each method was run for 20 epochs of $n = 50000$ episodes each, with episode length $K = 9$.

The results showed the net trained with Refreshed AdaBoost has worse accuracy than the baseline SGD method, and took longer to train.

The baseline method also has better solve percentages across the board. There is some generalization of solving ability past scrambles of length $K = 9$, but accuracy drops off quickly.

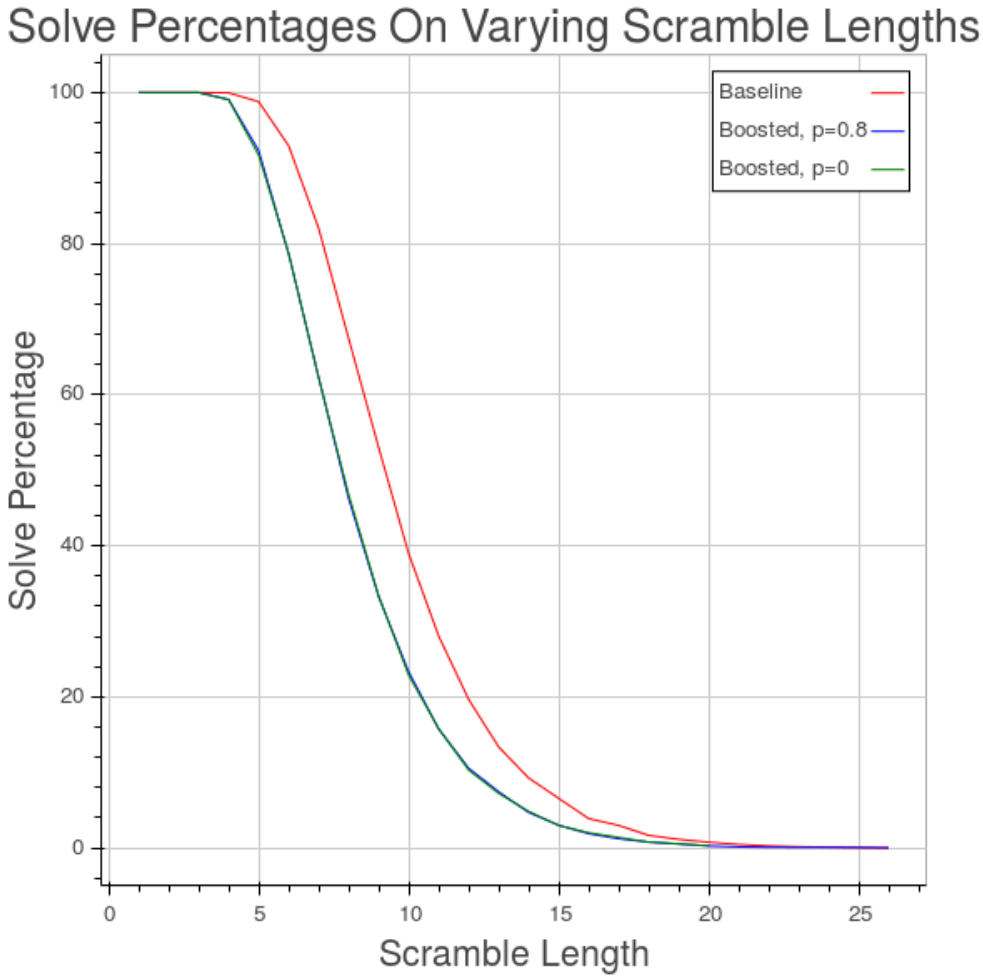

Figure 3: Solve Percentages of Best Model From Each Method

8 Analysis

Intuitively, AdaBoost should work better when

- Weak learners are cheap to train and evaluate.
- Weak learners propose different outputs when given the same input.

Neural nets are not cheap to train, so we do not have the time to train many weak learners. Further, because all gradient updates are applied to the same neural net, the neural net is regularized against changing too much each iteration. Manual inspection shows that when given the same input, the neural nets almost always agree on the best label.

Theoretically, a boosting approach will work with enough iterations, but it is too computationally expensive and there are not enough gains in performance.

The current approach does not deal with fuzziness in episode labels well. We assume a given cube has exactly one sequence of moves that solves it, but a given cube will have many possible solutions. Further work on this problem will need to address this issue.

9 Extensions

One extension is to use reinforcement learning to further optimize the solve percentage. This approach was used with AlphaGo, and nothing suggests it cannot work here. Additionally, a Rubik's Cube can be simulated very cheaply. The big downside of RL is that it is not very sample efficient, but in this case we have as many samples as required. There is prior work on replaying previously sampled actions with high error, which could be of use [8].

Another approach that may work is curriculum learning. By tweaking episode length K, we can change the difficulty of the problem. Instead of training with a fixed K, we could slowly increase K as the neural net achieves performance.

Finally, we propose one option for dealing with label fuzziness. During classification, if the neural net predicts a valid solution that differs from the expected one, the net should observe zero loss. We can try to identify this during training, then adjust the loss and labels appropriately.

10 Conclusion

We find negative results for combining boosting methods with neural nets. Although boosting methods are useful, they do not mix well with a hammer as general and powerful as deep neural nets, and seem best suited to other weaker machine learning methods.

11 References

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

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

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

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

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

[6] 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.

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

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

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