Metabuilder: Learning the Optimal Deep Network Architecture

Kiko Ilagan
B.S. Biology
Stanford University
ilaganf@stanford.edu

Anoop Manjunath
B.S. Biology
Stanford University
amanjuna@stanford.edu

Neel Yerneni
B.S. M.C.S
Stanford University
nyerneni@stanford.edu

Abstract

Though deep neural networks have gained increasing importance, designing network architectures remains a time-intensive task that requires human expertise. Effective automation of this task remains an important goal in machine learning research. In this paper, we develop Metabuilder, a reinforcement learning agent that automatically designs high-performance convolutional neural network (CNN) architectures for an image classification task. Using an ϵ-greedy exploration strategy, experience replay, and an LSTM for function approximation, Metabuilder searches a vast state space of possible architectures and designs a network that has comparable performance to human-designed networks on the same task.

1 Introduction

1.1 Motivation

Deep neural networks are powerful models with myriad practical uses. However, designing networks and setting hyperparameters that will perform well on a given task is a time- and skill-intensive process. High-performing networks are often created after much painstaking experimentation or by modifying a few known high-performance models.

Clearly, there is room for optimization in this process. If we formulate the task of designing a neural network as a sequential decision-making process, then we can apply reinforcement learning techniques to craft an algorithm that will be able to automatically generate a high-performing network architecture given a particular problem.

1.2 Problem Statement

Given an image classification task and associated training, validation, and test data, our goal is to train a reinforcement learning algorithm that can design a deep CNN architecture that achieves above 70% classification accuracy on the test set while only limiting itself to convolutional, max pooling, average pooling, and fully connected layers.

2 Prior Work

For this project, we were initially inspired by attempts to apply learning methods to hyperparameter optimization and tuning. There is extensive literature that studies the optimization of parameters like learning rate, regularization parameters, and momentum parameters. Many of these methods use either some kind of random search, gradient descent based method, or sequential Bayesian model to obtain near-optimal hyperparameter configurations. This problem can be inherently viewed as searching over the space of possible instantiations for a fixed model or network. However, in our method, we wanted to directly explore the space of all possible networks. As hyperparameter optimizations are fundamentally tied to certain aspects of the model space, making small changes to the model could create an entirely new set of optimal hyperparameters, requiring another search. As a result, we wanted our agent to be able to build the network itself from scratch while choosing hyperparameters along the way- similar in a high-level to the method introduced and proved in Li et. Al [5].

This type of "meta-learning" for automating model design has been somewhat explored but has not been consistently shown to outperform approaches involving hand-crafting networks, and comparable results were difficult to implement efficiently as they involved comprehensive searching. We were interested by potential applications to the space of image modelling using deep learning. The convolutional networks used in these problems typically involve a long series of different types of layers of varying sizes- an enumeration of possibilities that would be very difficult to search to determine the optimal architecture for a given problem. We believed that it may not necessarily be the case that hand crafted methods could produce an optimal architecture given the size of the state space. Many approaches have already validated methods to optimize hyperparameters for convolutional networks [3]. However, a recent paper from Baker et. Al. [1] actually directly explored the idea of building an optimal convolutional neural network for a given problem using a reinforcement learning agent to efficiently traverse the search space. They obtained results that outperformed hand-crafted methods on their experimental tasks.

3 Data

For this problem, we chose to use the CIFAR-10 image classification data set [4]. CIFAR-10 is a well-known image data set comprised of 60,000 32x32 color images separated into 10 distinct classes of 6000 images each. We split the data into 49,000 training images, 1,000 validation images, and 10,000 test images. Example images can be seen in Figure 1.

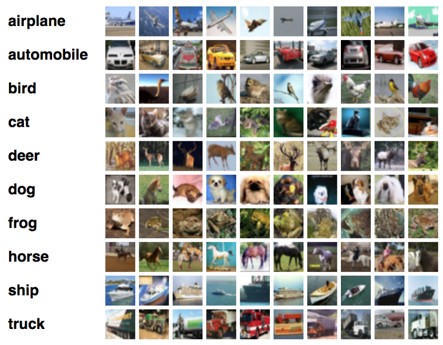


Figure 1: Example images and their associated labels from the CIFAR-10 image data set

We chose the CIFAR-10 image classification task because it is a relatively small data set comprised of small images, which greatly speeds training time. We had access to limited computational resources, so this simple data made training Metabuilder more feasible.

4 Approach

Convolutional neural networks were generated using the general process illustrated in Figure 2. A reinforcement learning agent was used to design an architecture for the neural network one layer at a time. From the perspective of the agent, the neural network is a list of layers. The reinforcement learning agent's action space at any time step consisted of layers that could be appended to the network. The layers, along with their possible parameters settings, are enumerated in Table 1.

In order to ensure valid neural networks, the first layer (action) was restricted to a 2D convolution while the last layer was restricted to being a layer of 10 fully connected neurons followed by a softmax activation function - the standard for image classification tasks such as CIFAR-10. A ReLU activation follows every convolutional and fully connected layer. The reinforcement learning agent specifies the learning rate for training the convolutional neural network as its final action. The agent

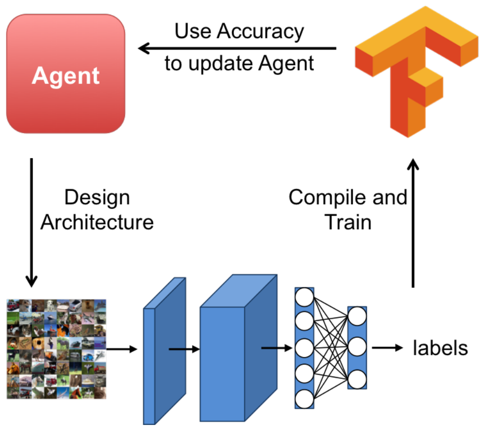


Figure 2: Process schematic for architecture exploration and testing

is also forced to flatten the input once before the output layer. All layers following this flatten layer are required to be fully connected. A maximum number of layers was set beforehand to manage computational complexity. With a maximum number of layers of 10, the number of possible action plans is on the order of 450×68^7.

Table 1: General action space for reinforcement learning agent

Layer	Parameter	Options
2D Convolution	Filters	1, 8, 16, 32, 64
	Kernel Size	3x3, 5x5, 7x7
	Stride	1, 2, 3
Maximum Pooling	Pool Size	2x2, 5x5, 7x7
	Stride	1, 2, 3
Average Pooling	Pool Size	2x2, 5x5, 7x7
	Stride	1, 2, 3
Fully Connected	# Neurons	25, 50, 75, 100
Batch Normalization	—	—
Flatten	—	—
Learning Rate	—	$10^{-1}, 10^{-2}, \cdots, 10^{-10}$

Once the architecture of the network has been decided, it is compiled using the Keras sequential API [2] and trained for one epoch on the training set of CIFAR-10 dataset with a batch size of 16 images. Once the model has been trained, the accuracy of its prediction on the validation is determined and passed back to the agent as the reward. To speed computation, we made the simplifying assumption that better networks would outperform inferior ones even after 1 epoch of training.

4.1 Baseline: Random Model

For a base of comparison, we created an agent that does not learn and simply designs a sequential CNN by choosing layers randomly, subject to the stated placement constraints.

In order to make the state space somewhat more manageable, we also pruned which layers could be chosen at a given time step given some domain knowledge: we disallowed two pooling layers in a row, we did not allow convolutional layers after flattening, and we discretized the parameters associated with each layer. These pruning assumptions were applied to all subsequent models as well.

4.2 Linear Model

As a first pass at an actual learning agent, we implemented a Q-learning agent with Temporal Difference control that used linear function approximation for the value function. Given weights $\mathbf{w} \in \mathbb{R}^n$, a state s, action a, reward associated with that action r, learning rate α, discount factor

γ, and feature function $\phi : S \times A \to \mathbb{R}^n$, we updated the learned weights of the model with the equation:

$$\mathbf{w} = \mathbf{w} + \alpha(r + \gamma(V(s) - Q(s,a)) * \phi(s,a)$$

We featurized the state space by creating a vector of aggregate statistics for the layers of the network so far: we counted each layer type, and we also counted the number of parameters associated with each layer. The Q value for a particular state could then be calculated as a simple dot product between the featurized state and the learned weights of the model: $Q(s,a) = \mathbf{w}^T \phi(s,a)$. The optimal value for a given state can be calculated by $V(s) = \max_a Q(s', a)$

4.3 Fully Connected Network for Value Approximation

One of the main weaknesses of the previous approach was, of course, its inability to handle non-linearity. To remedy this weakness, instead of performing a dot product to estimate Q values, we used a 2 layer fully connected neural network with 128 hidden units in the first layer and 64 in the second. Given the same feature vector $\phi(s,a)$ as above, the network outputs a reward $r \in (0,1)$ that represents the predicted classification accuracy of the proposed network the agent is building, approximating Q. The network is updated at every time step (batch of 1) to better predict the Q value (minimize temporal difference). In particular, at every time step, the "ground truth" value for prediction is $r + \gamma V(s)$, where $V(s) = \max_a Q(s', a)$, and $Q(s', a)$ is the output of the MLP for the new (s', a) pair. The network is fit to this value using gradient descent, with the gradient with respect to the mean squared error back-propagated through the network. An Adam optimizer with a learning rate of 0.01 is used.

4.4 LSTM for Value Approximation

A major limitation of the our initial Q-learning agent models is that the state representation of the network (aggregate statistics on number of neurons or numbers of different layers) fails to capture vital information regarding the relative position of layers (e.g. the identity of the previous layer) that is intuitively important for the construction of a neural network. The agent builds the network as a sequence of layers of indeterminate length, much like words in a sentence. Recurrent neural networks, as models well-suited to reading in sequences data of variable lengths and leveraging positional information within a sequence for prediction, stood out as strong candidates for global approximation of the Q-function.

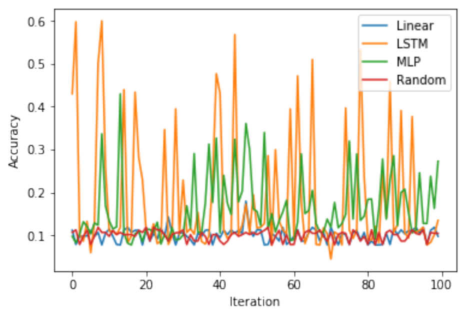
We re-featurized the state and action space in order to represent the current network as a sequence of layers. A state action pair (s, a) is represented by a $(N + 1)$x16 matrix, where N is the number of layers in the network so far. Each layer is represented by a 16 dimensional vector, permitting an indicator for each layer type, as well as a value for each parameter for each layer. Actions are represented as a vector of length 16 appended to the matrix representation of the network so far.

After empirical testing, we settled upon an LSTM model with 128 units in the hidden layer as our recurrent model. The final time step of the LSTM outputs to a single output neuron with a sigmoid non-linearity. This sigmoid ensures the output of the model lies in the range between 0 and 1, allowing the agent to effectively predict the classification accuracy of the model. We trained this LSTM and used it to predict the Q values exactly as with the fully connected network.

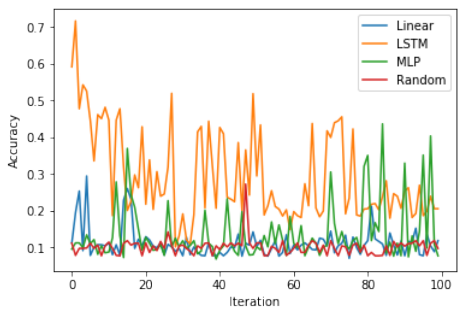
4.5 Offline Training

Since training the CNN with GPU acceleration takes on the order of 10 seconds for 3 layer networks and 90 seconds for 10 layer networks, it is inefficient to only train the models online. The MLP and LSTM models required long training times to achieve strong approximations of the Q function before model performance increased substantially. As a result, we generated 10,000 training examples with a random model and performed offline batch training on this data in order to ensure our models had robust estimates of the Q function before we evaluated their online performance. Offline training was performed in the same fashion as online training - we walk along a "history" of model construction and make actions as provided. Instead of generating and training a new network after the network is completed, we take the validation accuracy from the history. We performed 1,000 iterations of offline learning before assessing online performance. Further offline learning failed to make a significant difference in model performance.

5 Results



(a) Online training alone



(b) Online training following following 1000 iterations of offline training

Figure 3: Validation accuracy of 100 8-layer networks generated through online policy search by different reinforcement learning agents

Here, we started by training and testing the models online. As shown in figure 3a, running the models online incurred large bias in the model updates. Without any offline training, we can see that the models have extremely high variance in terms of their performance. However, the LSTM consistently outperforms the other approximation methods.

Offline training failed to rescue the performance of our random and linear approximation of Q, resulting in similar performance as with online training alone. We noticed slightly better results with our Multi-layer Perceptron (MLP), however it was difficult for this method to gain traction in training as it had trouble maintaining information on successes from previous iterations. As expected, we had consistently better results with the sequential LSTM model. It was able to learn successfully in an offline setting, and thus started off quite well. We noticed that poorer performances in an online setting significantly biased its estimates of Q, and hence its actions, making it difficult for this model to make progress on increasing accuracy in later iterations.

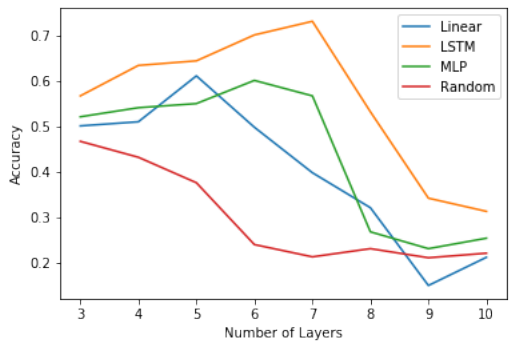


Figure 4: Test set classification accuracy of the best model produced by each agent versus the maximum number of layers allowed

Finally, we wanted to get a better understanding of the performance of our agents relative to the size of the architecture it had to learn to build. Considering every number between 3 and 10 as the maximum number of layers, we trained each agent offline for 1,000 iterations followed by 100 iterations of online search. In Figure 4 we present the test set accuracy of the best model (chosen by validation accuracy) produced by each agent during online training. After two layers, the linear approximation immediately started to drop off in accuracy, as it was unable to model more complex dependencies between multiple layers. Though the LSTM performed better on average, both the LSTM and MLP showed a decline in accuracy on models larger than 7 layers (not counting learning rate as a layer).

The optimal network determined by our agent consisted of 7 layers (counting flatten as a layer for the purpose of this problem). The network uses a 3x3 convolutional layer with 64 filters followed by a max-pooling layer of size 3. It then uses a 1x1 convolutional layer also of filter size 64, followed by an average pooling layer of size 7. The output of this layer was then flattened and passed through a dense fully connected layer of 100 units and finally output to represent a distribution over the 10 possible classes. An illustration of this final model is presented in 5. This model achieved 79% classification accuracy on the test set after 10 epochs of training, surpassing CNN designs that the authors of this paper had previously devised for CIFAR-10. This is a fairly impressive result considering the relative shallowness of the network and simplicity of the constituent layers.

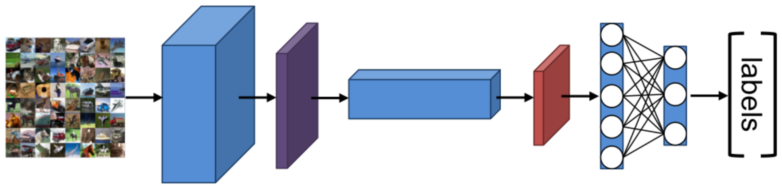


Figure 5: Schematic of the best CNN designed by the LSTM agent, which achieved 79% classification accuracy on the test set

6 Discussion

The ability for our model to capture long term sequential dependencies in a network's architecture ended up being pivotal in building a good learning agent. We noticed that both the linear and MLP model were heavily constrained by poor state representation as they were not able to effectively model things like spacing between layers or even synergies between certain longer term patterns of layers. This is likely why the MLP model showed minimal gain over the linear model in terms of the accuracy of the networks it produced. The recurrent nature of the LSTM model allowed it to learn these features and consequently gave us strong results. However, one drawback was that the LSTM model did require robust offline training to maintain its precision. As the model used to predict the Q-value of a state was the same one being used to predict the action, our predictions were fundamentally biased. In an online setting, this meant that bad results led to strong perturbations in the decision-making of the agent, causing a large degree of derailment in its previously trained decision-making.

We noticed that our agent performed better when we reduced the horizon of the model it was tasked with learning to build, with the best performance seen when the generated model was 7 or fewer layers. This is likely due to the fact that feedback received from optimal decisions made at later time steps could easily back-propagate to decisions made earlier given a smaller horizon. However, when tasked with constructing deeper networks, especially given our lack of resources to train the network for many epochs, we were unable to learn optimal architectures. This result manifests in Figure 4 as a steep drop in accuracy of the best model produced when the maximum number of layers increases. When the agent was able to construct a model with a significantly different accuracy, successive iterations would generally not see a corresponding momentous change. While this problem could certainly be tackled with greater computational power and longer training, we also believe there is potential for training extensively in lower dimensions and horizons as an effective approximation for policies in larger dimensions or horizons.

7 Conclusion

We have presented Metabuilder, a reinforcement learning agent that learned to construct a CNN architecture that achieved 79% classification accuracy on the CIFAR-10 image data set, a result that actually surpassed the accuracy of networks designed by the authors of this paper for other classes.

There is room for improvement in the value approximation and exploration of the model, but there is much potential for reinforcement learning approaches for the automation of neural network design.

Acknowledgments

We would like to thank Mykel and the rest of the course staff for all their hard work throughout the quarter.

References

[1] B. Baker, O. Gupta, N. Naik, and R. Raskar. Designing neural network architectures using reinforcement learning. *CoRR*, abs/1611.02167, 2016.

[2] F. Chollet et al. Keras. `https://keras.io`, 2015.

[3] T. Domhan, J. T. Springenberg, and F. Hutter. Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves. In *IJCAI*, volume 15, pages 3460–8, 2015.

[4] A. Krizhevsky. Convolutional deep belief networks on cifar-10, 2010.

[5] K. Li and J. Malik. Learning to optimize. *arXiv preprint arXiv:1606.01885*, 2016.