Q-learning Applied To Convolutional Neural Network Design

Machine Learning 156 Final Project
Creighton Anderson-Soria
Theo Teske
Jacob Brinn
Andreas Boulios

Objectives

Convolutional Neural Networks are popular and powerful machine learning tools that can yield impressive results while performing image classification tasks (LeCun et al., 2015). These networks generally work by feeding an image through a sequence of filtering, pooling and fully connected layers and outputting the probabilities that an image belongs to a set of predefined classes. By feeding multiple epochs of training data through the network, the model's parameters are repeatedly updated to minimize a loss function, and thus the model learns. However, the hyperparameters related to the model's architecture are not automatically updated. That is, the number of layers, layer type ordering and type of filtering/pooling layer, to name a few, are manually defined hyperparameters. As a result, we aim to automate the discovery of powerful and effective CNN architectures using Q-learning. The project will involve creating a reinforcement learning environment where an agent interacts with the environment to create neural network designs, subsequently receiving a reward that it uses to find more effective architectures.

Figure 1 - Traversing the state-action space

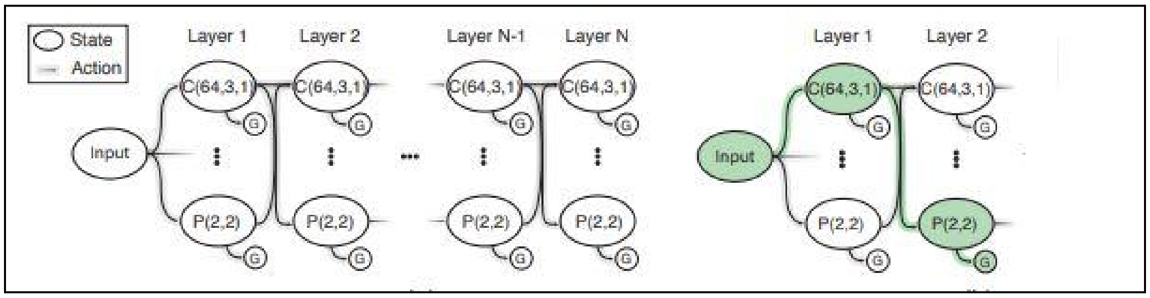


Figure 1 - Conceptually, the agent is choosing a path through an acyclic graph, where each vertex is a layer in the CNN it is building.

(Baker et al,. 2017) As demonstrated above, we will loosely think of the environment state as the current layer and the action as the choice of the next layer to define a finite Markov decision process where each episode results in a fully constructed architecture. We plan to set the reward given to the agent to be such that the agent is incentivised to find structures that have high accuracy when classifying images in datasets such as MNIST and SVHN.

The Datasets

For context, MNIST is a dataset consisting of 70,000 28x28x1 images of handwritten digits from 0-9. It is divided into 60,000 images for training and 10,000 for testing. See below for a sample:

Figure 2 - MNIST sample

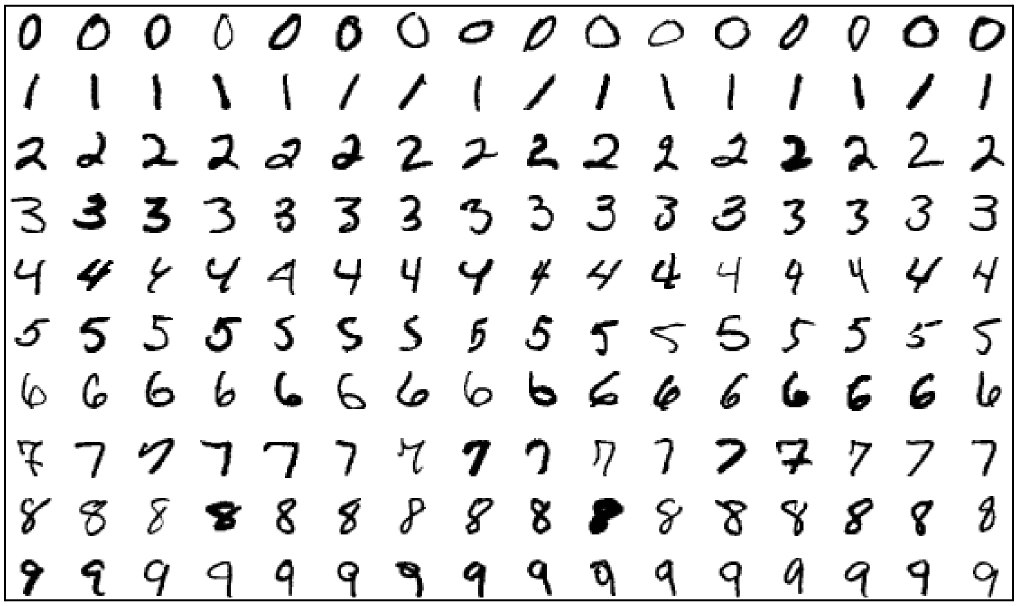


Figure 2 - 16 examples of each digit 0-9 contained in the MNIST dataset.

Similarly, SVHN is a dataset consisting of 99,289 32x32x3 colored images of digits 1-10, as found in photographs of street view house numbers. It is divided into 73,257 training images and 26,032 images for testing. See below for examples whereby each row corresponds to a distinct digit, in increasing order:

Figure 3 - SVHN Samples

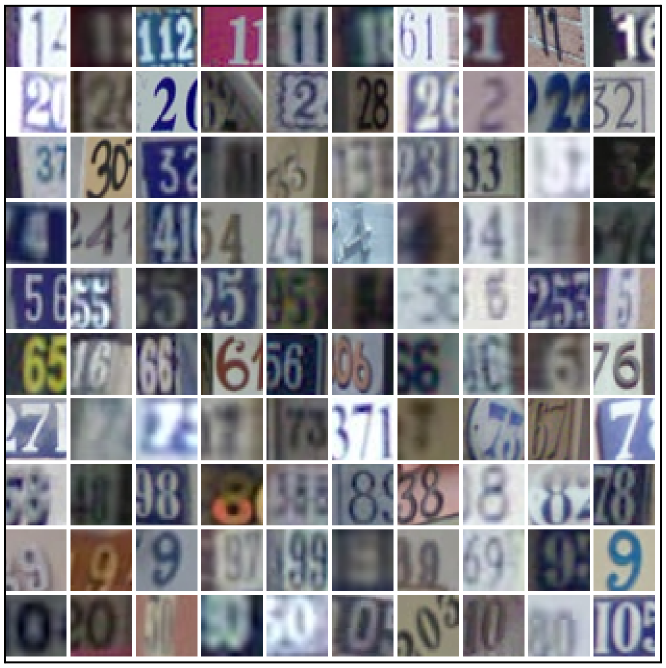


Figure 3 - 10 examples of each digit 1-10 contained in the SVHN dataset with digit value increasing by row.

Why is this interesting?

The proposed topic is interesting because, for one, it occurs at the intersection of two important realms of machine learning and cornerstones of the MATH156 course - neural networks and reinforcement learning. Convolutional neural networks and Q-based learning are both powerful methods by themselves, so we'd like to see how they can be combined to create an even more powerful method.

Also, as compared to other neural networks such as regular feed-forward neural networks, CNNs are particularly dependent on their network topology. That is, due to the increased number of layer type choices, CNN accuracies are quite sensitive to their architectures. This is relevant because the method we propose essentially automates the creation of CNN architecture, thus alleviating the need for expert intuition to inform structure choice. Our proposed method will theoretically allow users with little ML knowledge to easily create CNNs that can compete with those designed by ML experts.

What are other approaches used?

CNN architectures are often determined by the intuition and experience of the researchers who are designing the model. In popular architectures, it is commonplace that the pooling layers follow convolutional layers and fully connected layers are implemented towards the output of the model.

Figure 4 - AlexNet architecture

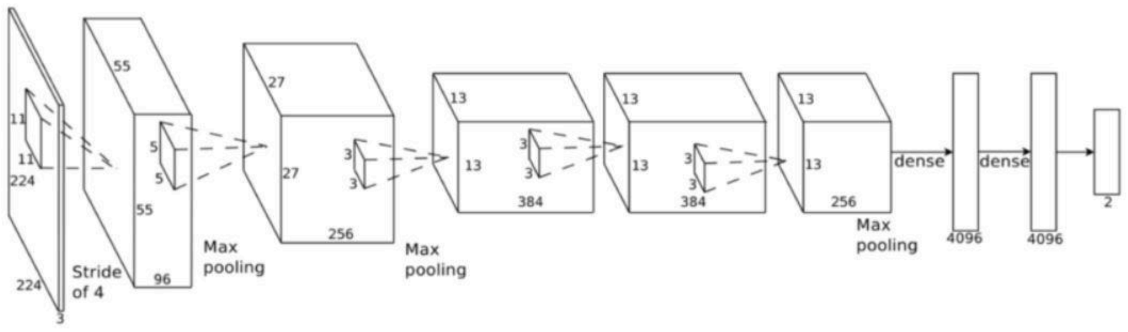


Figure 4 - In the above figure, we see the architecture of the AlexNet CNN (Alex Krizhevsky et al., 2012) following such a structure. This model was designed via the intuition of ML experts and was a state-of-the art image classification model when it was released in 2012.

In general, the state spaces of CNN architectures are extremely large, too large for an exhaustive search, so researchers have used methods such as Grid Searches (Andonie et al,. 2020) and Genetic Algorithms (Dominic et al,. 2020) to automate structural choices. We see that although hyperparameter tuning methods such as Grid Search are effective for simple tuning tasks, they tend to be less computationally efficient and less effective than reinforcement learning based methods when it comes to optimizing CNN architectures (Eimer et al,. 2023).

Methodology

The RL agent is trained using Q-learning, an algorithm which learns an action-value function Q that directly approximates the optimal action-value function q*, independent of policy followed. The update rule in Q-learning is as follows (Sutton and Barto, 2018):

$$Q(S_{t.}, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathrm{max}_{a \in \mathcal{A}(s)} Q(S_{t+1}, a) - Q(S_t, A_t)\right]$$

In our model, we set α=0.01 and γ=1. We select a policy from the action space using an ε-greedy policy. This means that stochastically, we either select the action which maximizes the learned action-value function Q, or we uniformly select an action from among all possible actions given the state. The ε-value determines how likely the agent is to select the maximizing action; for example, when ε=1, the agent samples uniformly from among all possible actions every time, but when ε=0.1, there is only a one in ten chance that the agent does not select the maximizing action. The idea is to begin with ε=1 for the initial training episodes to allow the agent to explore the state-action space, then gradually reduce the value of ε over the course of training. This way the agent becomes more likely to choose the maximizing action as it approaches the end of the training. The epsilon schedule we employ is shown below:

Figure 5 - MNIST-Simple - Iterations per epsilon

ε - value	**1**	**0.9**	**0.8**	**0.7**	**0.6**	**0.5**	**0.4**	**0.3**	**0.2**	**0.1**
Iterations per ε	50	30	10	8	8	8	8	8	10	20

Figure 5 - The agent completes the greatest number of iterations of META-CNN with ε=1 to allow for sufficient exploration. Also, we increase the iterations of META-CNN with ε=0.1 so that the agent can hone in on the optimal learned action-value function Q at the end of training.

The implementation of the META-CNN algorithm is demonstrated below:

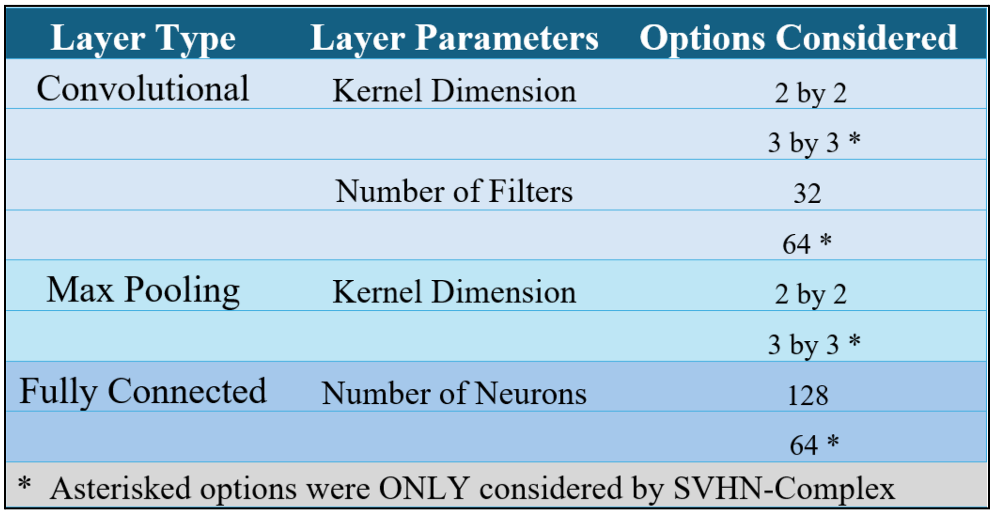
```
Algorithm 1 Q-learning for CNN Topologies - Group 11
Require: $\varepsilon \in [0,1]$, $Q$, $M$, $k$, topologies
 1: replay_memory ← []
 2: $Q \leftarrow \{(s,u)\ \forall s \in \mathcal{S}, u \in \mathcal{U}(s)\ :\ 0.5\}$
 3: for episode = 1 to $M$ do
 4:     $S$, $U$ ← SAMPLE_NEW_NETWORK($\varepsilon$, $Q$)
 5:     accuracy ← TRAIN($S$)
 6:     topologies.append(($S$, accuracy))
 7:     replay_memory.append(($S$, $U$, accuracy))
 8:     for memory = 1 to $k$ do
 9:         $S_{\text{SAMPLE}}$, $U_{\text{SAMPLE}}$, $\text{accuracy}_{\text{SAMPLE}}$ ← Uniform{replay_memory}
10:         $Q$ ← UPDATE_Q_VALUES($Q$, $S_{\text{SAMPLE}}$, $U_{\text{SAMPLE}}$, $\text{accuracy}_{\text{SAMPLE}}$)
11:     end for
12: end for
```

In our implementation of the Q-learning algorithm, we use an *experience buffer*, denoted replay_memory in the above pseudocode. This stores all network topologies that the agent has built during a given execution of META-CNN. At the conclusion of each episode, the algorithm uniformly samples a given number k of network topologies from the experience buffer, and uses the accuracy associated with each topology to update the Q-table. Doing so allows the Q-table to be updated k times each episode, speeding up convergence. This is especially crucial considering the computational intensity involved with training a single convolutional network.

Also, in the META-CNN algorithm, we keep track of the network topologies that have already been trained and the accuracy that was achieved with the given topology. If the agent then chooses a network topology that has already been trained, instead of training the network again, the TRAIN function returns the stored accuracy for that topology. Again, this is a crucial step to reduce computational intensity. The pseudocode for SAMPLE_NEW_NETWORK, UPDATE_Q_VALUES, and TRAIN can be found in Appendix

The action space for the MNIST-Simple model consists of one convolutional layer, one max pooling layer, one dense layer with ReLU activation function, and another dense layer with a softmax activation function. This last softmax layer is the terminal layer, so it outputs 10 probabilities, each of which corresponds to the likelihood that the image in question is in a given class according to the neural network. The action space for the SVHN-Complex model consists of four possible convolutional layers, two max pooling layers, two dense layers with ReLU activation function, and again a terminal dense layer with a softmax activation function. The action space described above is depicted in detail below.

Figure 1b - Action space available to the META-CNN agent.



Layer Type	Layer Parameters	Options Considered
Convolutional	Kernel Dimension	2 by 2
		3 by 3 *
	Number of Filters	32
		64 *
Max Pooling	Kernel Dimension	2 by 2
		3 by 3 *
Fully Connected	Number of Neurons	128
		64 *
* Asterisked options were ONLY considered by SVHN-Complex		

Figure 1b - Described above are all possible layers that the agent can choose in building a CNN. Note that in certain states, only a subset of the above actions are available to the agent.

Importantly, not every action is available to the agent at each state; for example, if the agent has just chosen a dense layer, it should not select a convolutional layer as the next action, as this would break the resulting neural network. To obviate this problem, each state is a tuple of the form (layer-number, layer-type, indicator), where layer-number indicates how many layers are already in the in-progress neural network (i.e., layer-num for the input layer is zero), layer-type is a string indicating one of the layers described in the action space table above, and indicator is an integer which determines whether or not the action space available in the given state should be restricted. In further efforts to prevent the agent from creating a neural network which will fail to compile, if the agent selects either a convolutional or a max pooling layer and the input is smaller than the kernel utilized in the layer, we apply any necessary padding so that the input is equal to the size of the kernel. Ideally, this should create a poorly performing neural network and the agent will learn not to attempt to apply a convolutional or max pooling layer when the input is too small.

Every neural network is trained with batch size 128, learning rate 0.01, and for the simple model 5 epochs were used, while the complex model required 10 epochs. We employ the AdaDelta optimizer and the ReLU activation function for the convolutional and intermediate dense layers. We did not optimize for these hyperparameters and simply chose them beforehand. We also implement a dropout layer after every two layers chosen by the Q-learning agent, each with a dropout probability of 0.2, in order to reduce the risk of overfitting.

Figures and Presentations

The graphs below correspond to a less complex model run on MNIST, dubbed MNIST-Simple, that has a maximum number of 5 layers. Each layer is picked from a list of only four possible types; convolutional, pooling, dense, and terminal. This allowed our code to run in a more manageable time frame as it was successful over fewer epochs and utilized a smaller state-action space. However, the idea behind this code can be further expanded to more complex models and other data sets such as SVHN. We implemented an example of this more complex model dubbed SVHN-Complex.

Unfortunately, we ran into the issue that more complex CNNs required more epochs to successfully reach high enough accuracies. High accuracy was necessary with our implementation because the new state action value is computed as a weighted average of its previous value and the accuracy. As a result, if the initialized values in the Q table are larger than most of the accuracies expected (given our number of epochs), then all strategies will get penalized. This is clearly problematic, since for the last few iterations of our code the states deemed optimal are expected to run more, and thus decremented further. Even after updating the initialization we would be comparing between low accuracies, and it is not guaranteed that the optimal strategy with a few epochs remains the optimal once we allow for more.

We discovered this issue by running the complex model using only 5 epochs, and while the code did finish and find an approximately optimal topology, since the number of epochs was so low, all the accuracies were also low, and thus our agent recommended a non-sensible “optimal” structure. When bumping up the epochs to a more reasonable number, 10, we immediately noticed accuracies that were far greater than before, but these episodes took far longer to run, and the compute power needed was outside our available resources for this project. By observing the first few episodes using the higher epoch value, we confidently hypothesize that our agent would find a true optimal CNN structure for the layer types and total number of layers we offered to it.

MNIST-Simple - Figure 2a - Accuracy by Iteration Number

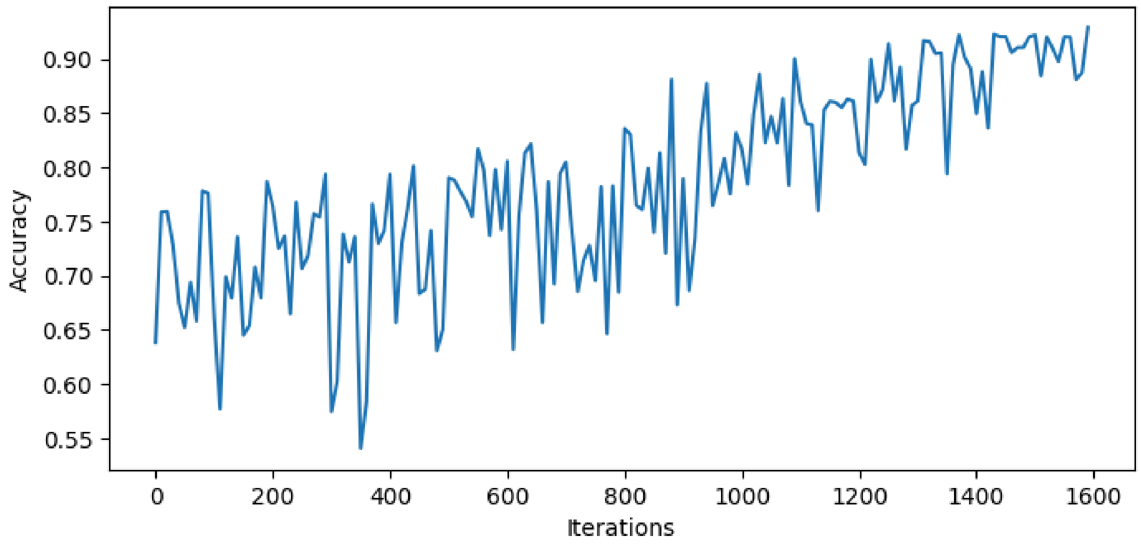


Figure 2a - For the first 500 iterations we have ε=1 and for the rest of the iterations ε is incrementally lowered until ε=0.1. This explains why the accuracy does not immediately increase - the model is exploring during the first 500 iterations and only increases when we start to lower ε and take greedy actions. Our model converged to an accuracy of 92.01%.

MNIST-Simple - Figure 2b - Accuracy by Epsilon-Value

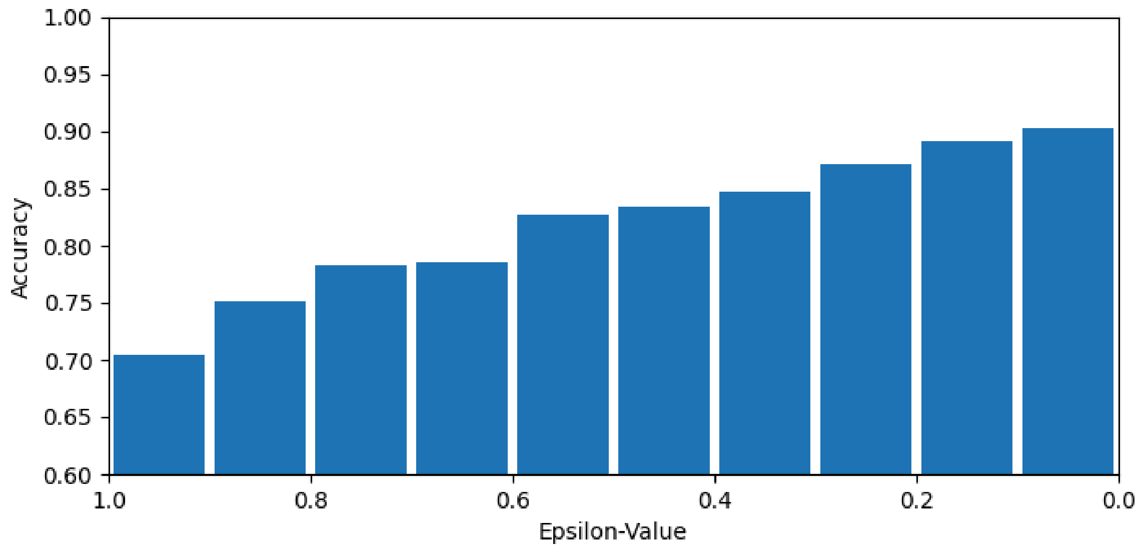
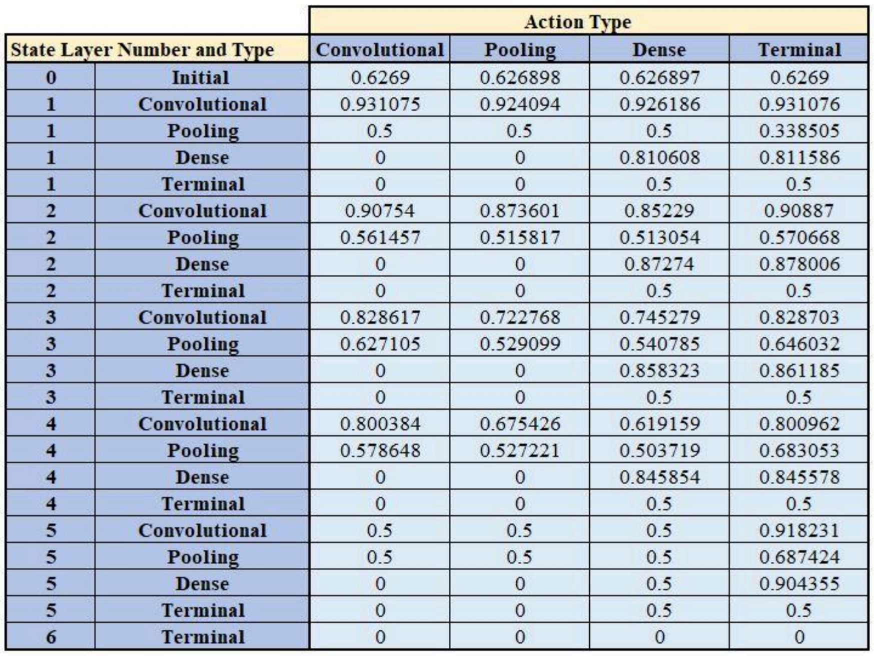


Figure 2b - We see how as we lower ε, accuracies tend to increase. This is, once again, expected and due to the fact that when ε is higher the model does more exploring, and when ε is lower the model does more exploiting of the information it gained by exploring.

MNIST-Simple - Figure 2c - Final Q-table



		Action Type			
State Layer Number and Type		Convolutional	Pooling	Dense	Terminal
0	Initial	0.6269	0.626898	0.626897	0.6269
1	Convolutional	0.931075	0.924094	0.926186	0.931076
1	Pooling	0.5	0.5	0.5	0.338505
1	Dense	0	0	0.810608	0.811586
1	Terminal	0	0	0.5	0.5
2	Convolutional	0.90754	0.873601	0.85229	0.90887
2	Pooling	0.561457	0.515817	0.513054	0.570668
2	Dense	0	0	0.87274	0.878006
2	Terminal	0	0	0.5	0.5
3	Convolutional	0.828617	0.722768	0.745279	0.828703
3	Pooling	0.627105	0.529099	0.540785	0.646032
3	Dense	0	0	0.858323	0.861185
3	Terminal	0	0	0.5	0.5
4	Convolutional	0.800384	0.675426	0.619159	0.800962
4	Pooling	0.578648	0.527221	0.503719	0.683053
4	Dense	0	0	0.845854	0.845578
4	Terminal	0	0	0.5	0.5
5	Convolutional	0.5	0.5	0.5	0.918231
5	Pooling	0.5	0.5	0.5	0.687424
5	Dense	0	0	0.5	0.904355
5	Terminal	0	0	0.5	0.5
6	Terminal	0	0	0	0

Figure 2c - Here we see the final Q-table. By reading off the table, we can observe that the final architecture contains a single convolutional layer and fully connected soft-max layer, or terminal layer. The reason our final model contains so few layers is due to the fact that the MNIST dataset is of low-fidelity. That is, many layers are not needed to identify the underlying patterns in the data, and adding more layers can induce overfitting.

Future Questions

One topic of note that arose in our post-analysis discussion of questions for the future was Double-Q learning. We discussed the possibilities of a future extension of this project that initiates two Q-tables, Q1 and Q2. Then, within the algorithm, when the function UPDATE_Q_VALUES is called, it would choose randomly and equally between updating Q1 or Q2 according to the respective update rules:

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathrm{max}_{a \in \mathcal{A}(s)} Q_2(S_{t+1}, a) - Q_1(S_t, A_t)\right]$$
$$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \mathrm{max}_{a \in \mathcal{A}(s)} Q_1(S_{t+1}, a) - Q_2(S_t, A_t)\right]$$

The only other change to implement this new algorithm would be within the ε-greedy policy approach. We would now take an ε-greedy approach with respect to the sum of Q1 and Q2. This version of Q-learning may be superior to the original for its ability to avoid maximation bias and converge faster (Sutton and Barto, 2018). We would like to explore whether this method would create a more efficient algorithm. If so, we would be able to use a larger state-action space or number of iterations without increased computational costs.

Another topic of discussion pertained to the other hyperparameters existent in a CNN. Our model addresses the structure of an overall CNN, including the order and number of different possible layers. However, we were still interested in the idea of fine-tuning the rest of the hyperparameters that we did not test for in our models, such as stride-length, activation functions, learning rate, etc. We could do so by including a greater number of possible actions for our agent to choose from. Unfortunately though, including these would exponentially increase the necessary computational power and time to run the model, but it is hypothetically possible for the agent to learn and optimize for these values under ideal conditions.

We also had questions about how well Q-learning based hyperparameter tuning can be generalized to models other than CNNs. We recognise that other models such as Graph Neural Networks are also highly dependent on the network's topology, so we'd be interested to see how well our hyperparameter tuning method would apply. We predict that it would, once again, generalize better than other methods such as Grid-Search that are widely used for optimizing scalar hyperparameters such as learning rates, rather than entire network topologies. To explore this, we would have to reconfigure our state-action space, but in general, the method would remain the same.

Reproducibility

This project should be fairly easy to reimplement for others who have the computer power, time, and desire to discover the optimal CNN structure for their favorite data set. In fact, within this project we ourselves created multiple different models that observed various numbers of layers across different data sets. If the reader wishes to observe the hyperparameters tuned in our models using our chosen layers and datasets, all that is required is loading our notebook, running all the cells, and finding a good book to read as the code runs. (We recommend Pattern Recognition and Machine Learning by Christopher Bishop).

However, should one wish to test their own custom datasets or layers, there are a few important details to consider. Ensure that the desired dataset is loaded into the notebook correctly and resized if needed so that it follows (number of examples x height x width x depth) and reorganized so that it matches the dimensions of the predictions on which the model will be tested. Then, move down to the **train** function and ensure that in the line **model.add(keras.Input**), so that the shape corresponds to the shape initiated above. For choosing custom layers and layers numbers, one should change the dictionary **states**, where the keys are tuples of three items; the layer number, the type of layer, and an indicator value, and the values are unique indices corresponding to the row associated with a given state tuple in the Q-table. For each state, update the indicator value in the function **permissible_actions** to determine the action space given your current state. For example, in our models we use the indicator value of 1 in **permissible_actions** to allow any layer to be added, while a value of 0 only allows dense layers or the termination (softmax) layer. We also use these indicator values when initializing the Q-table so that impermissible actions for a given state have the initial value of 0. Next, ensure that the dictionary **actions** below **states** contains all of the non-input layers you desire. Finally, implement the indicator values mentioned above in the functions **transition** and **permissible_actions** so that the flow of layers matches whatever pattern or rules you may want. (Just be careful, as certain combinations of layers may result in errors). Finally, run the rest of the code, resume reading a book, and view the results a few hours later.

Sources

A. Krizhevsky and I. Sutskever and G. Hinton, (2012). *ImageNet Classification with Deep Convolutional Neural Networks*

B. Baker and O. Gupta and N. Naik and R. Raskar, (2017). *Designing Neural Network Architectures Using Reinforecment Learning*

L. Den. (2012). *The MNIST Database of Handwritten Digit Images for Machine Learning Research*. IEEE Signal Processing Magazine, 29(6), 141–142. **

N. Aszemi and P. Dominic, (2020). *Hyperparameter Optimization in Convolutional Neural Network using Genetic Algorithm*. International Journal of Advanced Computer Science and Applications

R. Andonie and A.C. Florea, (2019). *Weighted Random Search for CNN Hyperparameter Optimization*

R. Sutton and A. Barto, (2018). *Reinforcement Learning: An Introduction* second edition

T. Eimer and M. Lindauer and R. Raileanu, (2023). *Hyperparameter Tuning in Reinforcement Learning is Easy, Actually*

Y. LeCun Y and Y. Bengio and G. Hinton, (2015). *Deep learning*. Nature. doi: 10.1038/nature14539. PMID: 26017442.

Y. Netzer, T. Wang, A. Coates, Al. Bissacco, B. Wu, A. Y. Ng, (2011). *Reading Digits in Natural Images with Unsupervised Feature Learning.* NIPS Workshop on Deep Learning and Unsupervised Feature Learning **

**cites the datasets SVHN and MNIST uses throughout this project

Appendix - Pseudocode

Below is the implementation for the algorithms SAMPLE_NEW_NETWORK, UPDATE_Q_VALUES, and TRAIN. Note that TRAIN references the list topologies which has been appended to by the algorithm META-CNN. Also, the function TRANSITION simply outputs the state tuple which results from the agent being in a state and choosing an available action, which is deterministic (i.e. given a state and a chosen action in that state, the next state is completely determined).

Algorithm 2 SAMPLE_NEW_NETWORK - Group 11

```
Require: $\varepsilon \in [0,1]$, $Q$
 1: state sequence $S \leftarrow [s_{\text{INIT}}]$
 2: action sequence $U \leftarrow [\,]$
 3: while $U[-1] \neq$ terminate do
 4:     $\beta \sim \text{Uniform}[0,1)$
 5:     accuracy $\leftarrow$ TRAIN($S$)
 6:     if $\beta > \varepsilon$ then
 7:         $u \leftarrow \text{argmax}_{u \in \mathcal{U}(S[-1])} Q[S[-1], u]$
 8:         $s' \leftarrow \text{TRANSITION}(S[-1], u)$
 9:     else
10:         $u \sim \text{Uniform}\{\mathcal{U}(S[-1])\}$
11:         $s' \leftarrow \text{TRANSITION}(S[-1], u)$
12:     end if
13:     $U$.append($u$)
14:     if $u \neq$ terminate then
15:         $S$.append($s'$)
16:     end if
17: end while
18: return $S, U$
```

Algorithm 3 UPDATE_Q_VALUES - Group 11

```
Require: $Q, S, U$, accuracy
 1: $Q[S[-1], U[-1]] = (1-\alpha)\, Q[S[-1], U[-1]] + \alpha \cdot \text{accuracy}$
 2: for $i = \text{length}(S) - 2$ do
 3:     $Q[S[i], U[i]] = (1-\alpha)\, Q[S[i], U[i]] + \alpha\, \text{argmax}_{u \in \mathcal{U}(S[i+1])} Q[S[i+1], u]$
 4: end for
 5: return $Q$
```

Algorithm 4 TRAIN - Group 11

```
Require: $S$
 1: for $(s, \text{acc})$ in topologies do
 2:     if $s = S$ then
 3:         return acc
 4:     end if
 5: end for
 6: acc $\leftarrow$ CNN($S$)        ▷ CNN($S$) trains a conv net with layers prescribed by the state sequence $S$
 7: return acc
```