Landing on the Moon with Reinforcement Learning

Deep Q-Learning on OpenAI LunarLander Game

Cailyn Craven

cailyn.craven@colorado.edu

Github: https://github.com/cailyn-craven/OpenAIFinal

1. Overview of Project:

Reinforcement Learning (RL) is a type of artificial intelligence where agents take actions in an environment to maximize cumulative rewards. RL overlaps with many disciplines including psychology, neuroscience, game theory, and control theory.[1] Many classic RL algorithms like ***Temporal Difference Learning*** date back to the 1980s or earlier.[2]

In the past decade, RL research has soared given the popularity of deep learning techniques and the ability to pair the two. Using non-linear neural networks as function approximators for Deep Reinforcement Learning required some research breakthroughs since initial Naive Deep Q Learning attempts struggled with the ***deadly triad.***

The ***deadly triad*** includes the following problems: *Convergence is not guaranteed when using a function approximator with a non-linear model. * If a single network is simultaneously used to evaluate the maximum action and choose the maximum action while being updated at every step, it has the effect of chasing a moving target. The model being trained doesn't actually have a gradient. *Highly correlated samples.

In the 2015 *Nature* paper "Human-level control through deep reinforcement learning" Mnih et al. described how they utilized a deep-Q-network agent (DQN) that reached human level performance on Atari games. To overcome the deadly triad, the researchers utilized: *a ***replay buffer*** to give the agent a memory. States are randomly sampled in a different order before using gradient descent so samples are no longer correlated. *To solve the problem of a moving target Q-value making the regression target unstable, the implementation uses a separate target network and prediction network. From Atari games, DeepMind continued to take on harder challenges. Reinforcement Learning featured prominently in the AlphaGo program that defeated Go champion Lee Sedol in 2016.

One of the primary purposes of this project was to use ***OpenAI Gym,*** a toolkit for developing and comparing reinforcement learning algorithms.[3] To understand Reinforcement Learning better, I started with a simple implementation of the classic ***Temporal Difference Algorithm*** to solve the FrozenLake-v0 environment.

From there, I turned my attention to Deep Reinforcement Learning. Phil Tabor provides an interesting walkthrough of a ***Naive DQN*** that illustrates the problems that arise from the deadly triad in naive implementations of a Deep Q-Learning algorithm for the Cartpole-v0 environment. After seeing firsthand some of the challenges of utilizing Reinforcement Learning and Deep Learning together, I turned to a ***Deep Q-Learning*** implementation capable of helping an agent learn on how to land on the moon in the LunarLander-v2 environment.

Approach:

2. Environments and Game Rules

[1] "Reinforcement Learning," Wikipedia, https://en.wikipedia.org/wiki/Reinforcement_learning

[2] "Temporal Difference Learning," Wikipedia, https://en.wikipedia.org/wiki/Temporal_difference_learning.

[3] "OpenAI Gym Documentation, https://gym.openai.com/

Over the course of the project, I touched on three different environments:

- For the basic ***Temporal Difference Algorithm,*** I utilized the ***FrozenLake-v0*** environment. FrozenLake is a simple text-based ***gridworld*** environment where the agent starts off in the top left corner and needs to make it down to the bottom right corner to retrieve a frisbee. The agent is on a frozen lake so sometimes it will stochastically slide in a direction it didn't intend to go. The agent receives a reward of 1 for making it to the bottom right corner. Other steps have a reward of 0. The game terminates if the agent falls into a hole on the ice. The 4x4 grid means the ***state space*** is 16, and the agent has four possible ***actions:*** left, right, up or down. [4]
- To better understand the deadly triad, I utilized ***Cartpole-v0.*** This was the toy problem utilized in the notebook introducing this project so it was important to base the main project on another environment. Therefore, I will just touch on this one briefly. In this ***classic control*** problem, a pole is attached by a join to a cart moving along a track. The pole starts upright, and the agent's goal is to prevent it from falling over. To control the system, apply a force of +1 or -1 to the cart. The agent receives a reward of +1 for every timestep the pole remains upright. The game ends when the pole reaches more than 15 degrees from vertical or the cart has traveled 2.4 units away from the center.[5]
- The primary environment for this project was ***LunarLander-v2,*** a continuous control task in the Box2D simulator. The simulation is based on a 1979 Atari Game. In this simulation, the goal is to land a rocket on the moon. We would expect that landing a real ship on the moon would require knowing lots of physics and control theory. In this game, we frame the problem as a RL problem and see if the agent can learn how to land on the moon without knowing anything about dynamics. We train the agent to land efficiently (and most importantly avoid crashing!)

  The problem can be formalized as an ***MDP.*** The environment has a landing pad which is always at coordinates (0,0). The ***state*** is composed of ***8 variables***: position x, position y, velocity x, velocity y, angle, angular velocity, and sensors for whether each of the two legs are touching the ground (the sensors have value 0.0 for not touching, or 1.0 for touching). Six of the eight state variables are ***continuous*** with values ranging between negative infinity and infinity. The values are float32.

  The ***4 discrete actions*** are: do nothing, fire left orientation engine, fire main engine, or fire the right orientation engine.

  Reward: The reward is an important part of an MDP. For this game, the agent receives a reward between 100 and 140 points for moving from the top of the screen to the landing pad at zero speed. The agent loses the reward if it subsequently moves away from the landing pad. The episode finishes with a large reward of +100 for landing successfully or -100 for crashing. A leg contacting the ground receives a reward of +10. Firing the main engine loses 0.3 points per frame. In this version, the ship

[4] FrozenLake-v0, OpenAI Gym Documentation, https://gym.openai.com/envs/FrozenLake-v0/

[5] Cartpole-v0, OpenAI Gym Documentation, https://gym.openai.com/envs/CartPole-v0/

has infinite fuel. The game is considered solved if an agent averages 200 points over the past 100 games.[6]

3. Model Choices

Problem Solving:

I structured the project as a progression of looking at Temporal Difference, a classic reinforcement learning algorithm, then trying out a Naive Implementation of Deep Q Learning, and then an implementation of Deep Q Learning so that I could really understand how algorithms could address the limitations of other algorithms to make them suited to solving problems in certain environments.

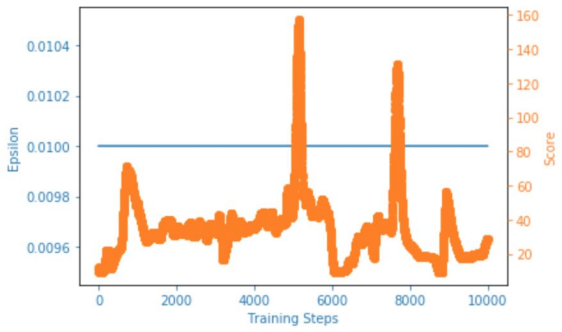
For the FrozenLake environment, I utilized **Temporal Difference** which keeps a running moving average, bootstrapping an estimate of the value function from the current estimate of the value function.[7] A **tabular learning method** uses a table to keep track of the states and actions an agent has visited. Such an approach was well suited to the FrozenLake environment because it has a small number of states and actions so it is possible to represent the state action paris as a table with the columns as the actions and the states as the rows.

I utilized an ***Epsilon Greedy action selection*** strategy which balances ***exploration-exploitation*** by looking up the Q-value for each action and most of the time taking the ***greedy max action*** but sometimes taking a random action with epsilon probability.

This classic algorithm was easy to implement in OpenAI Gym, but when I moved on to the LunarLander environment, the classic Temporal Difference algorithm wasn't enough for the problem. Unfortunately, the value function for the LunarLander environment can't be represented with a table because of the continuous state space variables.

Given the limitations of tabular methods, I utilized the following process to select an algorithm better suited to the LunarLander environment. The ship starts in a low orbit and descends until it lands or crashes. Then each landing attempt repeats independently of the previous one. This means that the algorithm should be well suited to ***episodic tasks.*** Q-Learning, SARSA, or Expected SARSA with a ***neural network function approximator*** are good choices for a **control problem** like this one. For this project, I explored the Q-Learning algorithm.

For researchers, it wasn't immediately clear how to pair Reinforcement Learning and Deep Learning. The deadly triad was challenging to overcome, and I wanted to make sure that I understood the challenges. Phil Tabor gives a very interesting tutorial where he does a Naive Implementation of Q-Learning for the Cartpole environment to show how it fails. This isn't the primary part of my project and I coded along with Tabor to complete it, but I included a copy of the notebook on GitHub in the NaiveDeepQLearning file because the results were interesting.



As epsilon decreases the agent's performance improves until it hits a peak as would be expected. Then after reaching a peak, the rewards come crashing back down. This occurs several times. The Naive Q Learning algorithm

[6] LunarLander-v2, OpenAI Gym Documentation, https://gym.openai.com/envs/LunarLander-v2/

[7] "Temporal Difference Learning," Wikipedia, https://en.wikipedia.org/wiki/Temporal_difference_learning

doesn't work. The agent discards its past experiences without learning from them. As epsilon decreases, the agent is more likely to get stuck in a local minimum. The same network is being used to evaluate the max action and choose the max action so the model is chasing a moving target. Maybe if we ran this long enough, the agent would actually learn, but Cartpole only has a state space of size 4. Most Atari games have much higher input dimensions, and even for the 8 input dimensions for the LunarLander simulation, we would not expect this Naive Q-Learning approach to work. Therefore, to implement the lessons from the Mnih et al. 2015 paper and avoid the ***deadly triad*** for a DQN, it is important to include a ***replay buffer*** and ***two networks:*** a target network and a prediction network.

Metaparameter Choices:

Design choices to implement the agent include selecting the ***function approximator,*** deciding how to ***update action values,*** and how to do ***exploration.***

Function Approximator: For a function approximator it is often best to start simple. One simple option would be a tile coder that represents features in a continuous state space. Unfortunately, for a tile coder, the number of features grows exponentially with the input dimensions, and in this case, the state space has 8 dimensions. This means that in this case, a ***neural network*** is a good choice for the function approximator.

Activation Function: If we use a sigmoidal function like tanh, we may run into ***vanishing gradient issues.*** **ReLUs** are a popular choice that make sense to use for this problem.

Training the Network: To use ***stochastic gradient descent,*** a number of hyperparameters would need to be tuned to be effective. Due to this, I explored other options besides vanilla SGD. One option would be ***adagrad.*** Adagrad decays values down to zero which is problematic for non-stationary learning. ***RMSProp*** utilizes information about the curvature of the plots to improve the descent step. ***ADAM*** is a good choice because it combines the curvature information from RMSProp with ***momentum*** which helps speed up learning.

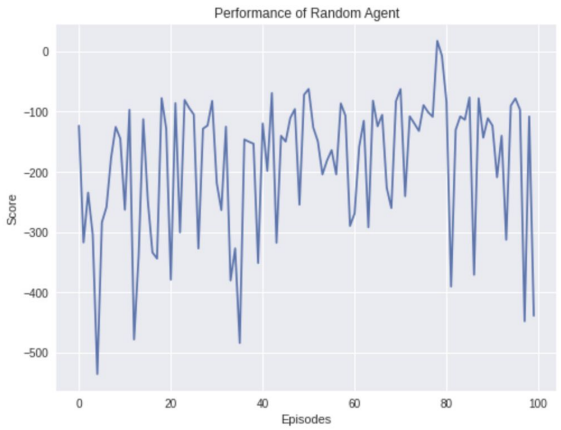
Exploration Strategy: One option would be to use ***optimistic initial values.*** This is a good option for a linear function approximator with non-negative values, but does not work well with neural networks. In this case, I chose to implement ***Epsilon Greedy.*** A downside of the epsilon greedy strategy is that it is that random action selection is completely random. One possible experiment is to compare the performance of an ***Epsilon Greedy*** strategy with a ***Softmax*** strategy where the probability of selecting an action is proportional to the value of the action. This way, we would expect that the agent would be less likely to select really bad actions.

4. Tests, Experiments and Results

Testing

My primary strategy to test whether the agent was learning was to print out the average rewards over the past 100 episodes while the agent was training and then plot the average score over the past 100 episodes vs. number of episodes. The environment is considered solved if an agent averages 200 points over the past 100 episodes so this testing strategy helps evaluate whether an agent is demonstrating learning.

I started every model run by plotting the performance of a completely random agent. As expected, the random agent performed poorly and achieved a large negative score.



On the other hand, a plot like with an upward trend in scores shows that the agent is learning:

Watching the demo video clips also can be useful for debugging because it is helpful to have a visual representation of what the agent is doing.

Experiments

To improve the agent's performance ***hyperparameter tuning*** is critical. Indeed, in the Mnih et al. 2015 article, hyperparameter tuning was critical to the agent's ability to perform well on Atari games. Given the timing constraints and my access to GPU resources, these experiments just scratch the surface of a thorough hyperparameter tuning approach.

Experiment with Neural Network Size

Purpose:

For the LunarLander problem, a neural network with a single hidden layer is sufficiently powerful to represent the value function. More neurons mean more representational power, but they also mean a larger number of parameters that must be learned.

Methods for Neural Network Size Experiment:

To experiment with neural network size, I ran three sets of 2000 episodes with networks that had hidden layers with input/output dimensions 32/16, 32/64, and 256/128. For each set of 2,000 episodes, I recorded the average reward for each hidden layer size.

Results of Neural Network Size Experiment:

The ***network architecture*** with a fully-connected input layer with 8 units and an output of 256 units followed by a fully connected hidden layer with input size 256 units and output size of 128 units followed by an output layer with an input unit size of 128 units and then output size of 4 units (corresponding to the number of actions), performed best with an average reward of 247.43 across the three 2000 episode trials.

Here is a plot of the agent training with the larger neural network:

Compared to the smaller neural network:

Problem Solving: Given that solving the LunarLander environment requires an average score of 200 across the past 100 episodes, I realized that the smallest neural network with a hidden layer with input/output dimensions of 32/16 provided insufficient representation. With this architecture, the agent had an average reward score of 200.76 across the three trials. The lowest average score was 197.7 which doesn't successfully solve the environment. To ensure that the agent will learn how to solve the environment, it is important to use more neurons in the hidden layer. Indeed, utilizing the largest number of neurons tested allowed the agent to achieve average reward scores of 200 over the past 100 episodes after between 500-700 episodes instead of over a thousand episodes like the middle and smallest number of neurons.

Experiment with Replay Buffer Size

Purpose: As the Naive Deep-Q Learning implementation showed, without a ***replay buffer, correlated elements*** made the neural network unstable. Mnih et al. 2015 demonstrated how having a replay buffer was an important factor for pairing RL and Deep Learning. The larger the replay buffer size, the less likely correlated elements will be sampled so the neural network will likely be more stable. However, a large experience buffer also requires a large amount of memory and slows down the training time.

A replay buffer fills up with an agent's memories and then overwrites old experiences with new experiences. With a small buffer, the agent will care most about the things it has experienced recently. This could be limiting with regards to the agent's performance. However, having a really large buffer could also have downsides if the agent has recent promising trajectories. On the Stack Exchange thread, "How large should the replay buffer be?" one commenter gives a good analogy to how a large replay buffer is like a college student having all of the books for their entire academic career on the bookshelf. If the student recently decided to pursue a programming career but needs to sample from all of their books including their first-grade books that is a disadvantage. [8]

Methods for Replay Buffer Experiment:

To experiment with the replay buffer size, my plan was to run three sets of 2000 episodes with a 1000, 50k, and 100k replay buffer size. It would be helpful to run the experiment with a larger range.

Results of Replay Buffer Size Experiment

With a 100k replay buffer, the average score over three trials of 2000 episodes was 247.43.

Without any replay buffer, the Naive Deep Q implementation was unstable. In this case, using a small buffer of size 1,000, the ship didn't learn how to land within 2000 episodes and achieved a negative average reward.

[8] "How large should the replay buffer be?", Stack Exchange Post, https://ai.stackexchange.com/questions/11640/how-large-should-the-replay-buffer-be

Problem Solving: This experiment affirmed the importance of having an adequately sized replay buffer.

Traditionally, human pathologists diagnose cancer by observing stained specimens on a slide glass with microscopes. With advances in slide scanning technology and reduced costs for storing digital images, many such slides have been scanned and saved as digital images called

WSIs (Whole Slide Images)[9]. Promising current research focuses on histopathological image analysis using machine learning, in particular deep learning techniques.

It is essential for breast cancer staging to detect whether breast cancer has spread to the lymph nodes. Identifying metastatic tissue in lymph node scans is a time-consuming task, and challenging to perform accurately.[10] A 2017 *JAMA* article described a retrospective study where a review by pathology experts changed assessments of whether cancer had spread to the lymph nodes for 24% of patients. There are hopes that digital histopathologic analysis could improve the efficiency and accuracy of identifying metastatic tissue in lymph nodes.[11]

This project features a 2018 Kaggle competition to "create an algorithm to identify metastatic cancer in small image patches taken from larger digital pathology scans" of lymph node sections. This is a binary image classification task of presence of tumor tissue or no tumor tissue.[12]

Dataset: The dataset for this project comes from the ***PatchCamelyon(PCam)*** dataset , a benchmark medical imaging dataset of H&E stained WSIs of sentinel lymph node sections from breast cancer patients in the Netherlands in 2015.[13] If cancer has metastasized, it is most likely to have spread to the sentinel lymph node. We are given a training set with 220,025 images, and a test set with 57.5k images.[14] Both the training and testing set have the structure of a directory with .tif image files inside. The .tif file format is used for high resolution files that are

[9] Komura and Ishikawa, "Machine Learning Models for Histopathological Image Analysis," *Computational and Structural Biotechnology Journal,* 2018, https://www.sciencedirect.com/science/article/pii/S2001037017300867.

[10] Litjens et al. "1399 H&E-stained sentinel lymph node sections of breast cancer patients: the CAMELYON dataset." *Gigascience,* 2018. https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6007545/

[11] Bejnordi, Vieta, and Von Diest. "Diagnostic Assessment of Deep Learning Algorithms for Detection of Lymph Node Metastases in Women with Breast Cancer." *JAMA.* 2017. https://jamanetwork-com.colorado.idm.oclc.org/article.aspx?doi=10.1001/jama.2017.14585

[12] "Histopathological Cancer Detection." Kaggle Competition. https://www.kaggle.com/c/histopathologic-cancer-detection/overview

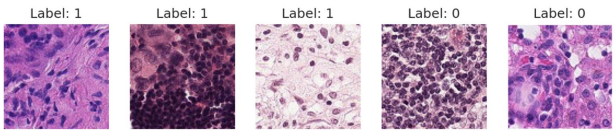
[13] Vieta and Von Diest 2017

[14] "Histopathological Cancer Detection." Kaggle

not compressed.[15] The images are 96 pixels by 96 pixels with three color channels [R,G,B]. Each image has a label corresponding to 1, metastatic tissue present, or 0, no metastatic tissue. A positive label means that the center 32 pixel by 32 pixel patch of the image contains at least one pixel with tumor tissue.[16] There is about a 60-40 split between 0 labels and 1 labels. The Kaggle dataset for this project is a subset of the PCam dataset that doesn't contain duplicates.[17]

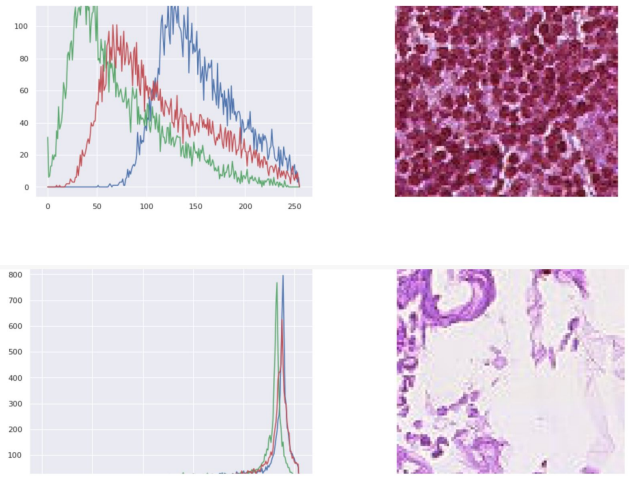
5. Exploratory Data Analysis:

The dataset images are stored in a structured .tif format, but the image data doesn't tell us what is in the image so image data is considered unstructured data.[18] EDA for unstructured data is somewhat open-ended. I plotted sample images with labels, but I couldn't visually identify distinctions between the classes.

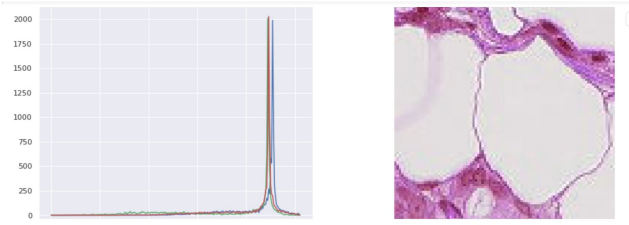


Color images are 3-D stacks of arrays with three color channels: R, G, and B. I plotted histograms of samples of images with tumor tissue and samples of images with no tumor tissue. These flattened histograms separated each image into its color channels and binned each pixel by its intensity value between 0 and 255. The histograms for the samples from each class didn't look all that similar to one another.

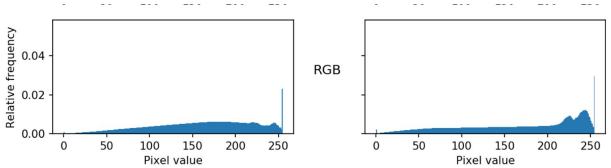
For example, here are two histograms for images without tumor tissue:



This histogram for a sample with tumor tissue looks somewhat similar to the previous histogram for a sample without tumor tissue.



The Kaggle Competition Notebook for the Health Hackers team compares distributions of pixel values for samples of thousands of images for each class. In that team's visualizations, the frequency of pixel values across all three channels showed differences between samples from the two classes:



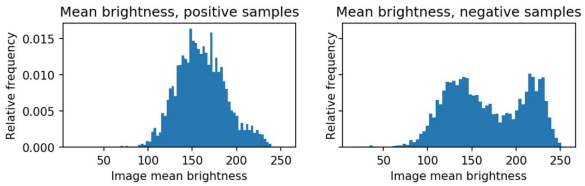
Histograms of the mean brightness for samples from each class showed differences as well:

[15] Vince Tabora. JPEG, TIFF, PNG, SVG File Formats And When To Use Them. *Medium.* July 6, 2020. https://medium.com/hd-pro/jpeg-tiff-png-svg-file-formats-and-when-to-use-them-1b2cde4074d3

[16] PCam Dataset. https://github.com/basveeling/pcam

[17] "Histopathological Cancer Detection." Kaggle

[18] Quora Response: https://www.quora.com/Why-are-images-considered-unstructured-data-when-they-can-be-stored-in-databases



[19] It is promising that basic EDA suggests differences between the two classes of images.

Data Cleaning and Preprocessing:

First, a note about cleaning. This project used a subset of the PCam dataset from Kaggle so I didn't need to worry about wrongly labeled images or bad images. All of the images were the same 96 x 96 pixel size. Given the quality of the dataset, I didn't perform any data clearing procedures.

For data preprocessing, I ***normalized*** the image pixel values. The documentation for the PCam dataset provided an example of ***data augmentation,*** perturbing data with random transformations.[20] A CNN has ***invariance*** if i can, "robustly classify objects even if placed in different orientations."[21] Veeling et al. 2018 note that data augmentation with random transformations may improve generalization but still have downsides including not capturing local symmetries or guaranteeing equivariance. The researchers propose exploiting the fact that histopathology images are inherently symmetric under rotation and reflection. Further steps for this project can include testing the G-NN proposed by Veeling et al.[22]

Plan of Analysis: The approach in computer vision used to be utilizing machine learning to build up a feature table and then perform classification. Improvements in computer hardware, the backpropagation algorithm, and autodiff have led to DeepLearning achieving great success on image classification tasks. A **convolution** is a matrix product that performs feature extraction. **Filter values** are weights that are learned while training the network. Convolutions train under **gradient descent** so they automatically learn to extract features that minimize the loss for correctly classifying images according to their labels.[23] The large size of the PCam dataset makes it possible to train a deep neural network.

6. Model Architecture

The architecture that I used involves n sets of two convolutional layers followed by max pooling (subsampling) and then finally a classifier. Below is a visual of the architecture from Professor Kim's slides:

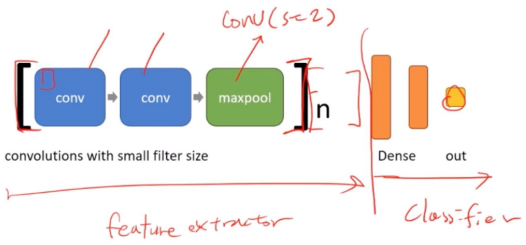
[19] "**Complete beginner's guide [EDA, Keras, LB 0.93]",** Health Hackers Team, Kaggle Histopathological Cancer Detection. https://www.kaggle.com/gomezp/complete-beginner-s-guide-eda-keras-lb-0-93

[20] PCam Dataset, https://github.com/basveeling/pcam

[21] Gandhi, Arun. "Data Augmentation: How to Use Deep Learning When You Have Limited Data, Part 2," Nanonets Blog, 2018, https://nanonets.com/blog/data-augmentation-how-to-use-deep-learning-when-you-have-limited-data-part-2/

[22] Veeling et al. "Rotation Equivariant CNNs for Digital Pathology." MICCAI. 2018. https://link-springer-com.colorado.idm.oclc.org/chapter/10.1007%2F978-3-030-00934-2_24

[23] Brownlee, Jason. "How Do Convolutional Layers Work in Deep Learning Networks?" https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks/



[24]

I chose the architecture because it is similar to the **VGGNet (2014) architecture** that performed well for classifying images in the ImageNet dataset.

I used all **filters** of 3 x 3 based on lessons learned from Yann LeCun's LeNet 5 architecture which was designed to recognize characters from the Mnist Dataset. LeNet 5 also used convolutions followed by subsampling and then a classifier. LeCun used filers of varying sizes so both smaller filters and larger filters were used. For computational efficiency, it is actually better to use smaller filters multiple times

Max Pooling is an important component of this architecture. One of the biggest challenges with fitting a neural network is the large number of parameters. Subsampling with max pooling shrinks down the feature map size.

When choosing an architecture, I looked at the Bejnordi et al. 2017 paper in *Jama.* The paper includes a table of "Test Data Set Results of the 32 Submitted Algorithms vs Pathologists" for a very similar challenge to the one in this Kaggle competition.

Table 2. Test Data Set Results of the 32 Submitted Algorithms vs Pathologists for Tasks 1 and 2 in the CAMELYON16 Challenge[a]

Table 2. Test Data Set Results of the 32 Submitted Algorithms vs Pathologists for Tasks 1 and 2 in the CAMELYON16 Challenge[a]

Codename[b]	Task 1: Metastasis Identification FROC Score (95% CI)[c]	Task 2: Metastases Classification AUC (95% CI)[c]	*P* Value for Comparison of the Algorithm vs Pathologists WTC[d]	Algorithm Model Deep Learning	Architecture	Comments
HMS and MIT II	0.807 (0.732-0.889)	0.994 (0.983-0.999)	<.001	✓	GoogLeNet[24]	Ensemble of 2 networks; stain standardization; extensive data augmentation; hard negative mining
HMS and MGH III	0.760 (0.692-0.857)	0.976 (0.941-0.999	<.001	✓	ResNet[25]	Fine-tuned pretrained network; fully convolutional network
HMS and MGH I	0.596 (0.578-0.734)	0.964 (0.928-0.989)	<.001	✓	GoogLeNet[24]	Fine-tuned pretrained network
CULab III	0.703 (0.605-0.799)	0.940 (0.888-0.980)	<.001	✓	VGG-16[26]	Fine-tuned pretrained network; fully convolutional network
HMS and MIT I	0.693 (0.600-0.819)	0.923 (0.855-0.977)	.11	✓	GoogLeNet[24]	Ensemble of 2 networks; hard negative mining
ExB I	0.511 (0.363-0.620)	0.916 (0.858-0.962)	.02	✓	ResNet[25]	Varied class balance during training
CULab I	0.544 (0.467-0.629)	0.909 (0.851-0.954)	.04	✓	VGG-Net[26]	Fine-tuned pretrained network

The top performing algorithms included GoogLeNet, ResNet, and VGGNet architectures.[25]

I chose an architecture based on VGGNet since in addition to working well for image classification tasks, it is simple to understand.

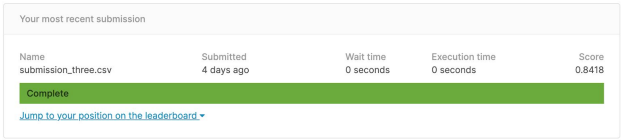
7. Results and Analysis

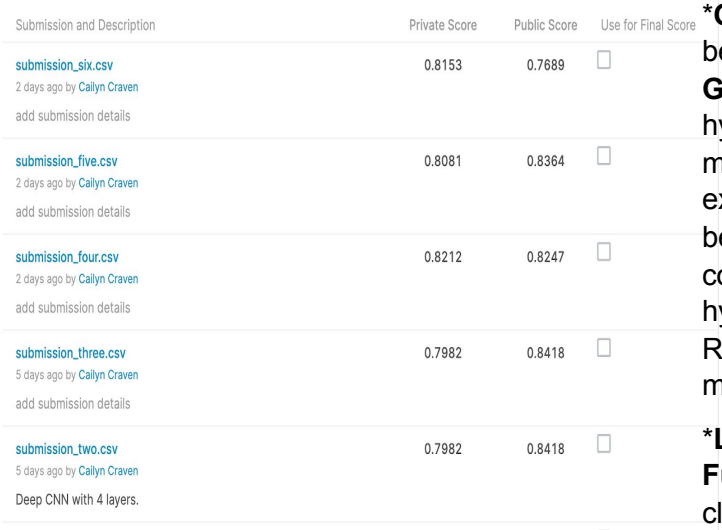
The majority class has no tumor tissue so classifying all images as no tumor tissue would have an accuracy of 60%, but have no functionality for identifying whether cancer has spread. With regards to **Evaluation,** the Kaggle page says, "Submissions are evaluated on ***area under the ROC curve between the predicted probability and observed target." AUC*** indicates how well a model does at separating classes. Since AUC is a probability, its values range between 0 and 1 with higher being better and 1 indicating a perfect job classifying. I focused on maximizing the AUC metric during the project.

My top five submissions had a range of public scores between 0.7689 to 0.8418 and private scores between 0.7982 and 0.8212.

[24] Kim, Geena. Convolutional Neural Network Lecture. University of Colorado Boulder Fall 2020.

[25] Vieta and Von Diest 2017





I split the training set so 25% was a validation set. I trained on 75% of the dataset and then evaluated the performance on the 25% of the dataset held out before testing the model on the Kaggle test set. Keras gave accuracy metric readouts like the following as the models trained.

```
5156/5156 [==============================] - 606s 117ms/step - loss: 0.4160 - accuracy: 0.8147
- auc: 0.8786 - val_loss: 0.3713 - val_accuracy: 0.8408 - val_auc: 0.9062
```

Hyperparameter Optimization Procedure and Summary:

The following is a summary. Please see my notebooks for more detailed analysis.

Choosing an Activation Function:** The activation function transforms the summed weighted input into the output for that input. Originally perceptrons used **Binary Threshold Step** functions. Options for activation functions include sigmoid, tanh, and ReLU. Unfortunately nonlinear functions like sigmoid and tanh can't be used with networks with many layers because they suffer from the ***vanishing gradient problem. For the hidden layers, I used the ***ReLU*** activation function. For the output layer, I used the **sigmoid function** since it is a good option for binary classification. If this was a multiclass problem, I would have used the softmax function.

***Optimizer Function:** With more time, it would be a good idea to experiment with **Stochastic Gradient Descent** and tune the hyperparameters like: base learning rate, momentum, and decay. In this case, I experimented with both Adam and RMSProp because they are both advanced options that could be used out of the box without hyperparameter tuning. Both Adam and RMSProp showed similar performance with the models I built.

***Loss Function:** The **Cross Entropy Loss Function** is an appropriate choice for a classification problem.

***Zero Padding:** when building a network with many layers, if zero padding isn't used, each subsequent layer loses a pixel off the edge of the image. Adding zeros around the edge of the input image makes it so the convolution kernel overlaps with the edges of the image. I added zero padding, but this was less important for this particular dataset. The label for each image is only based on the 32 by 32 pixels in the center of the image. The images in the dataset are 96 by 96 pixels to allow for pixels around the edges getting lost in subsequent layers even if the implementation doesn't utilize zero padding.

***Number of layers:** the human visual system has multiple processing layers. Creating a deep neural network more closely imitates the human visual system. Early layers respond to lines and simple textures. In higher layers, feature maps extract specific objects, in this case tumor tissue. VGGNet is typically around 20 layers. GoogLeNet, another high performing architecture, is 22 layers deep. More layers come with more computational cost and VGGNet

is already extremely slow to train so for this project I used a smaller number between two and eight layers.

The models that I built with more layers had more of a tendency to overfit. They would achieve really strong AUC scores in the mid 90's on the training data and then on the test set for the Kaggle competition only achieve AUC scores in the low 80's. One of the issues that I encountered was downsampling too aggressively so I received an error about the input having negative dimensions.

***Reducing the Number of Parameters:** A fully connected neural network would have too many parameters. With **max pooling,** we can summarize the outputs of convolutional layers in a concise manner. I used the model.summary() feature to show the number of parameters.

Below is a CNN without pooling. The convolutional layers have a small number of parameters, but the dense layer has 184k parameters so the network has 187,332 total parameters.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 96, 96, 10)        280
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 96, 96, 10)        910
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 96, 96, 10)        910
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 96, 96, 10)        910
_________________________________________________________________
flatten (Flatten)            (None, 92160)             0
_________________________________________________________________
dense (Dense)                (None, 2)                 184322
=================================================================
Total params: 187,332
Trainable params: 187,332
Non-trainable params: 0
_________________________________________________________________
```

After adding (2,2) MaxPooling after the convolutional layers, the dense layer has a much smaller number of parameters and the network has 3,732 parameters total.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_4 (Conv2D)            (None, 96, 96, 10)        280
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 48, 48, 10)        0
_________________________________________________________________
conv2d_5 (Conv2D)            (None, 48, 48, 10)        910
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 24, 24, 10)        0
_________________________________________________________________
conv2d_6 (Conv2D)            (None, 24, 24, 10)        910
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 12, 12, 10)        0
_________________________________________________________________
conv2d_7 (Conv2D)            (None, 12, 12, 10)        910
_________________________________________________________________
max_pooling2d_3 (MaxPooling2 (None, 6, 6, 10)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 360)               0
_________________________________________________________________
dense_1 (Dense)              (None, 2)                 722
=================================================================
Total params: 3,732
Trainable params: 3,732
Non-trainable params: 0
_________________________________________________________________
```

Keeping a model with four layers but then just adding max pooling and dropout sped up the training time and gave a substantial boost to achieving AUC scores in the mid 80's.

Stride is another way to reduce the number of parameters.

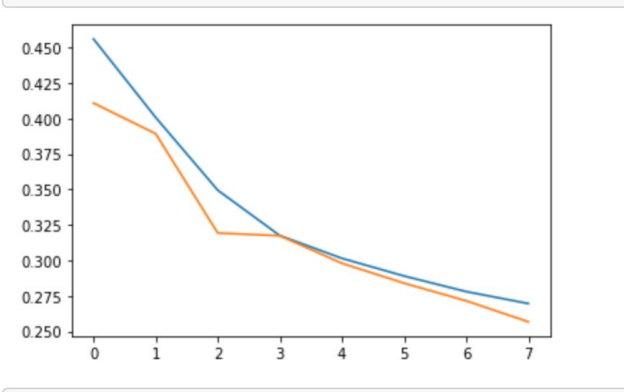
***Batch Size:** CNNs are trained with gradient descent. An estimate of the error is used to update the weights based on a subset of the training data. The more training examples used in the estimate, the more accurate the estimate will be. This makes it more likely that the weights will be adjusted in a way that improves the performance of the model. Unfortunately, it also means greater computational cost. A batch size of one would be completely stochastic. I experimented with small batch sizes of 32, 64, and 128. (Note: I tried to train a network with different batch sizes over 3 epochs and then choose which one performed the best. This experiment seemed to suggest that a smaller batch size of 16 was best. I cannot find this experiment so I need to rerun it. I need to consider whether running different batch sizes on 3 epochs is effective, or if I can design a better experimental approach. When making predictions, I realized that it was important to use a batch size that evenly divided the dataset or

make sure that the last batch could be smaller so images wouldn't be cut off.)

***Avoiding Overfitting:**

Number of Epochs: An epoch refers to the number of times that the network will do a forward and backward pass of the training dataset. The number of epochs does not increase the model complexity, but running more epochs increases the chances of overfitting.

As the network trains, we would expect the loss to decrease as the network "learns" what differentiates one class from another. One way to avoid overfitting is to track the loss over a number of epochs for the training and validation set. If the loss begins to increase for the validation set, it suggests that overfitting is occurring. For example, in the graph below, the blue line is the training loss and the gold line is the training loss over 8 epochs. In this case, loss for both continued to decrease indicating that learning was occurring as expected.



In this project, I used the **Keras callback module** to store the best parameters before the network started overfitting and then used the stored weights in a .hdf5 file to predict classes.

Two possible strategies for avoiding overfitting and making the most from our training data include: **dropout** and **batch normalization. Dropout** involves selecting a subset of units and ignoring that subset in the forward pass as well as the backpropagation. It allows us to train the network on different parts of the data. If a part of the network starts to overfit, other parts of the network will compensate because they haven't been trained on those same samples.

Batch normalization takes the output of a particular layer and scales it so that it always has a mean of zero and standard deviation of one.

I looked at resources like Stack Overflow posts and then tried out some variations of placements for Dropout, Batch Normalization, and combinations of the two.[26] One of my next steps would be to work more closely from an implementation of VGGNet so I could get a better idea of where to place Dropout and Batch Normalization as well as if it is best to utilize them together.

Analysis: Throughout this project, I struggled with overfitting, particularly as I attempted to add more convolutional layers. In future work, it would be interesting to exploit properties of the images such as histopathology images being inherently symmetric under rotation and reflection in the data augmentation stage. It looked like other researchers also increased performance when they addressed the class imbalance of having more images without cancer than with cancer. Perhaps these changes in the data preprocessing stage could help the model generalize better.

Dropout and batch normalization are two regularization strategies that help avoid overfitting. When attempting to address the overfitting, I read that a disharmony can occur when using batch normalization and dropout

[26] Stack Overflow, "Where Dropout Should Be Inserted?" https://stackoverflow.com/questions/46841362/where-dropout-should-be-inserted-fully-connected-layer-convolutional-layer#:~:text=3%20Answers&text=Usually%2C%20dropout%20is%20placed%20on,can%20really%20place%20it%20everywhere.

together wherein dropout slows down learning and batch normalization tends to make learning go faster. Sometimes the effects counter each other in such a way that the network performs worse when both are used than if only one was used on its own.[27] The network where I only used dropout had a higher private score than the models that used both so this is something I could explore in greater depth.

For future work, I would switch the optimizer function to SGD and optimize the hyperparameters including learning rate. I would like to continue working on this project and try to achieve an AUC score over 0.90. I think that I could achieve such scores with a 4 layer network with hyperparameter optimization.

8. Conclusion

Overall, I was able to achieve a decent AUC score utilizing a relatively simple Deep Convolutional Neural Network incorporating components of the VGGNet architecture. The model was a considerable improvement over classifying based on the majority class. This was my first Computer Vision project, and my first time working with the Keras library. This project very much just scratches the surface as an introduction to CNNs and there are numerous avenues to explore further, particularly with regards to hyperparameter optimization.

One criticism of CNNs is that even if they perform well, they are a black box. For applications like cancer diagnosis that are a matter of life and death, humans would want to understand why a model returns the results that it does. In recent years, there has been a great deal of focus on improving the interpretability of Deep Neural Networks. For this project, I wanted to convolve images with the kernel and visualize the result to see if I could see what properties of the image were emphasized by the kernel in that layer. With more time, I would do this to see if it improved my interpretation of the model.

The last lecture for the course material mentioned **transfer learning** where using standard large models and pre-trained weights from natural image sets like ImageNet have become a common technique for deep learning applications. Given the differences between natural images and medical images, an interesting and open direction of study explores utilizing transfer learning for medical images. Raghu et al. 2019 suggest that lightweight models can perform comparatively better than ImageNet architectures. It would be interesting to keep investigating this area.[28]

The code for this project is available at: https://github.com/cailyn-craven/Cancer_Classification

[27] "Introduction to Deep Learning with PyTorch." DataCamp. https://campus.datacamp.com/courses/introduction-to-deep-learning-with-pytorch/using-convolutional-neural-networks?ex=10

[28] Raghu et al. "Transfusion: Understanding Transfer Learning for Medical Imaging." arXiv 2019. https://arxiv.org/abs/1902.07208

CSPB 3202 Artificial Intelligence, Spring, 2020, Boulder, CO, USA	Craven 2020