Homework #4 CPTS-570 Machine Learning

Ricardo Rivero

December 01, 2024

1 1. Q-Learning algorithm and experimentation.

To implement the Q-learning algorithm using a GridWorld environment, we defined a state space of 10x10, where the start state is the top left cell (cell 9 in the Y axis, 0 in the X axis), where the action could take any direction from left, right, up and down to move around. We then coded the "win state" with a reward of 1, and the "loss states" with a reward of -1, where every time the agent reached the win state, the simulation was reset. For the walls in the gridworld, we mapped the positions of the gray states and coded the simulation such that the model could not move into those positions.

Then, a deterministic, Markov Decision Process, with a discout factor $\beta = 0.9$ was used.

According to the Reinforcement Learning lecture, we implemented the Q-earning with a fixed learning rate $\alpha = 0.01$, and explored different explore/exploit policies as follows.

1a. ϵ-greedy.

For the epsilon-greedy policy, we tried three ϵ values, 0.1, 0.2, and 0.3, and allowed the simulation to be repeated for 10,000 episodes with a 10,000 maximum steps per episode. To determine convergence of the Q-values, we set up a threshold of 1e-3, which would stop the simulations when the change in Q-values was lower than this threshold.

This implementation resulted in convergence after 1795, 1922, and 1719 episodes for $\epsilon = 0.1, 0.2, 0.3$ respectively. Interestingly, the agent explored more of the world when $\epsilon = 0.1$, reaching a maximum Q-value of 0.95 at position (3,5) which is right below the winning state.

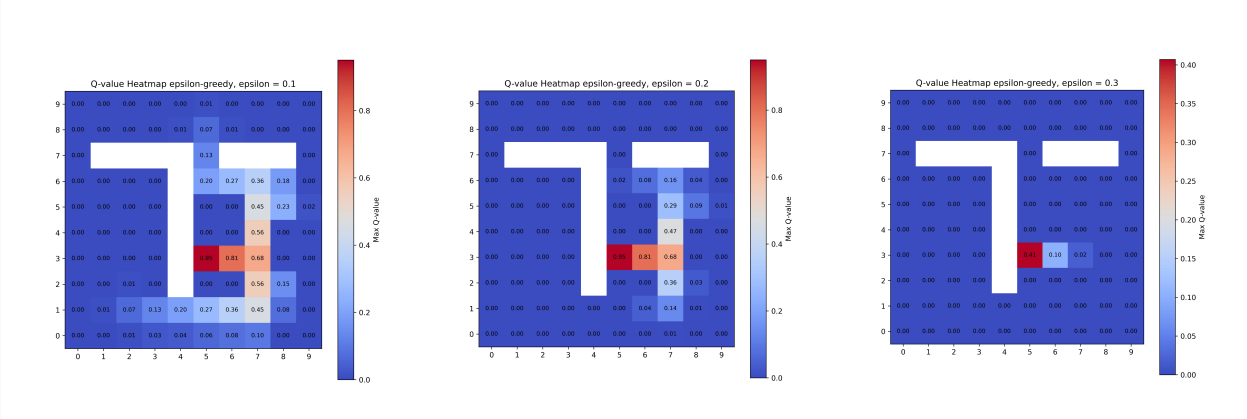

Figure 1: Implementation of Q-learning using a $\epsilon - greedy$ explore/exploit policy and different values of ϵ.

1b. Boltzman exploration.

For the implementation of the Boltzman exploration, the configuration of the grid and the simulation remained unchanged. However, we implemented a fixed decay schedule for the temperature of the exploration,

starting with a large valyue $\mathbf{T} = 20$ and a decay of 0.99 for each episode.

This exploration/exploitation policy reached the same maximum Q-value as the ϵ-greedy when $\epsilon = 0.1$, with a Q-value of 0.95 at (3,5). However, these exploration policy explored a bigger portion of the map and often found value in exploring different routes to reaching the winning state.

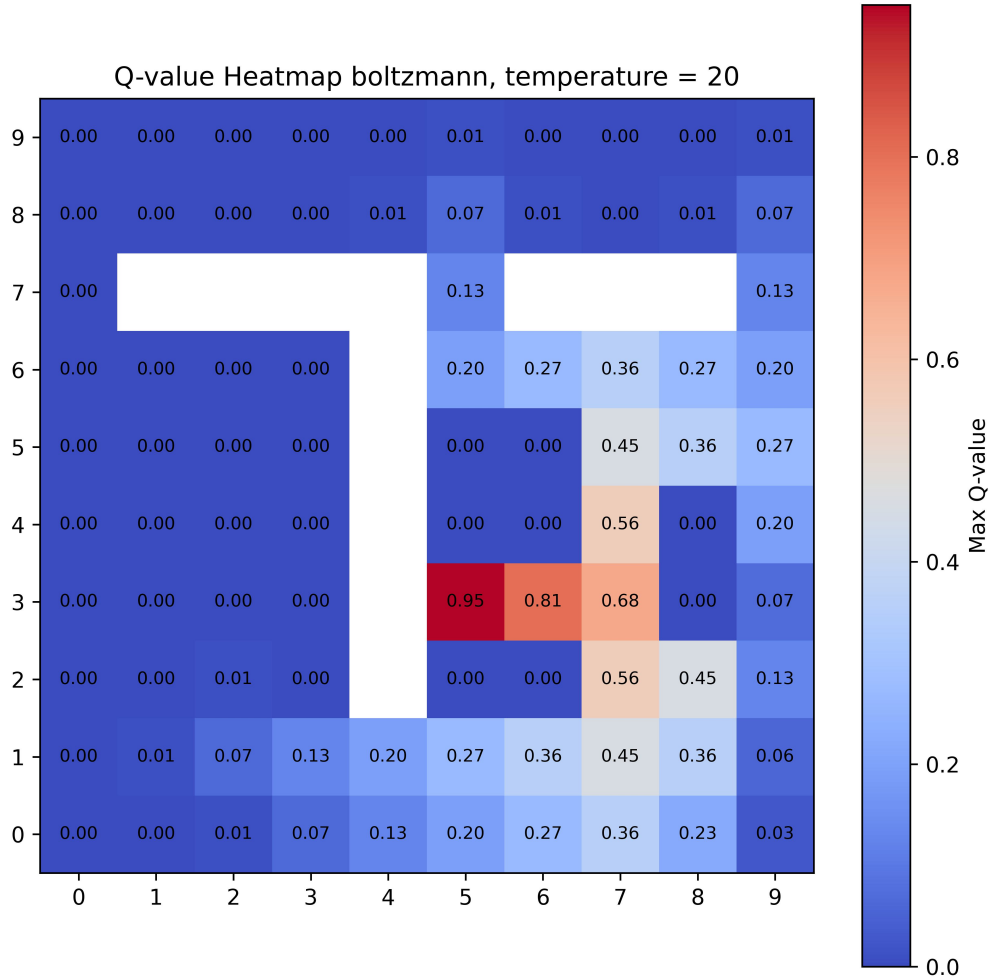

Figure 2: Implementation of Q-learning with a Boltzman exploration policy and a temperature of 20, with a decay of 0.99.

The Boltzman policy converged after 1,490 episodes, but its training was slower than the ϵ-greedy policy.

2. Convolutional Neural Network for image classification.

A convolutional neural network was implement for solving an image classification task based on the Fashion MNIST dataset. Briefly, the architecture of the neural network is as follows:

1. CNN Layer #1: 1 input channel, 8 output channels, a kernel size of 5, padding = 2, stride = 2, followed by ReLU operation.
2. CNN Layer #2: 8 input channel, 16 output channels, a kernel size of 3, padding = 1, stride = 2, followed by ReLU operation.
3. CNN Layer #3: 16 input channel, 32 output channels, a kernel size of 3, padding = 1, stride = 2, followed by ReLU operation.
4. CNN Layer #4: 32 input channel, 32 output channels, a kernel size of 3, padding = 1, stride = 2, followed by ReLU operation.
5. Average Pooling layer: nn.AdaptiveAvgPool2d(1).
6. Fully connected layer: 32 input channels, 10 output channels (equal to the number of classes in the dataset)

As suggested in the demonstration, a batch size of 64 was used for the initial implementation of the CNN, then the model was trained using an Adam optimizer with a learning rate of 0.001 and Cross Entropy was used as the loss function. After 20 training epochs, the model resulted in a training average loss and accuracy of 0.2461 and 90.8%, respectively. Then, we evaluated the model using 10,000 examples, resulting in a testing accuracy of 89.37%.

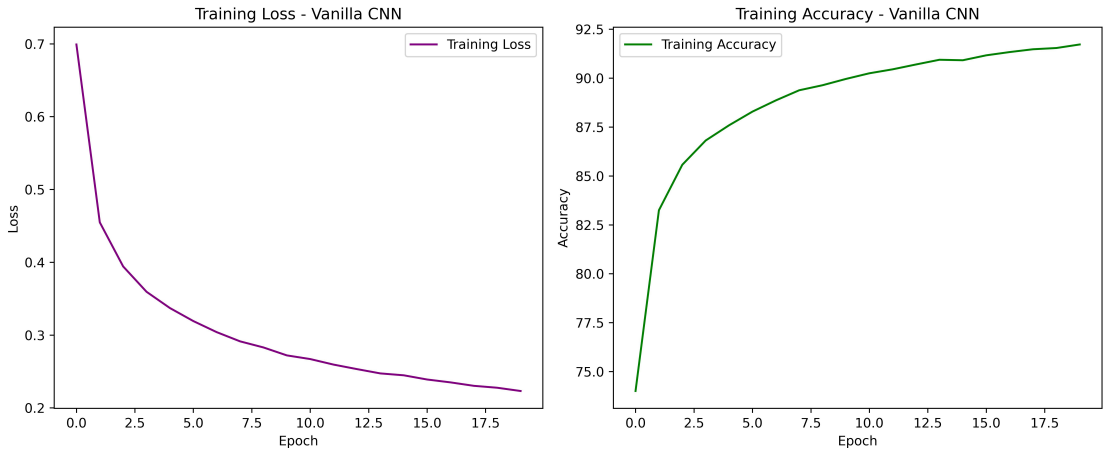

Figure 3: Vanilla implementation of the Convolutional Neural Network to classify images from the Fashion MNIST dataset.

Next, we modified the implementation to enhance the classification performance of the neural network. Namely, we added a batch normalization step after every convolutional layer using the number of output channels of the respective convolutional layer. Additionally, we tried by increasing the batch size to 128 and 256, and increasing the learning rate to 0.005 and 0.01. All of these modifications resulted in an increase in the average loss on the classifier. Finally, we tested the performance of the classifier using a batch size of 32 and a learning rate of 0.001, which resulted in an increase in the performance of the classifier, with an average loss and training accuracy of 0.2029 and 92.44% at the end of 20 training epochs, and a testing accuracy of 90.72%.

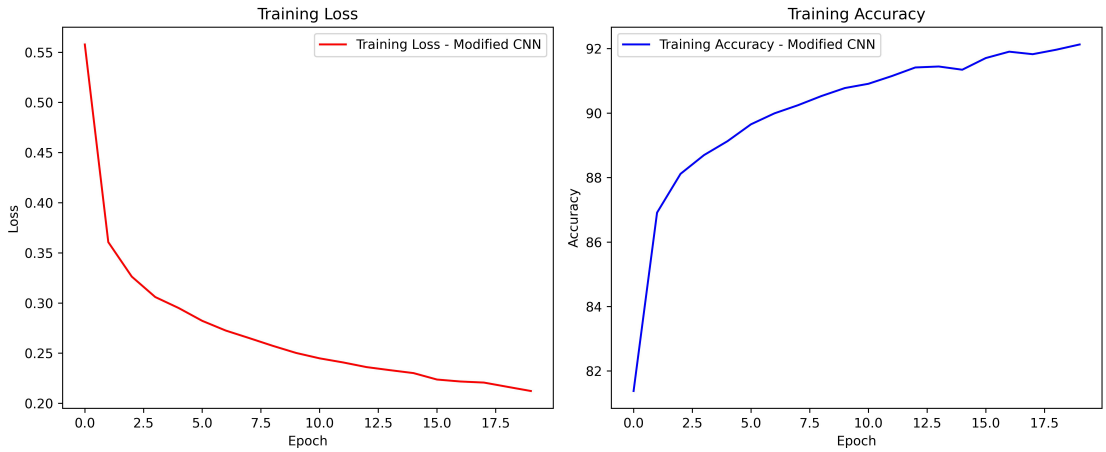

Figure 4: Modified implementation of the Convolutional Neural Network to classify images from the Fashion MNIST dataset.

3. Sculley et al, Hidden Technical Debt in Machine Learning Systems. NIPS 2015:2503-2511.

The paper by Sculley et al addresses the problems that arises with machine learning implementations and how it is dangerous to think that there are free wins in the field. To do this, they explore ML-specific risk factors that are related to the software engineering concept of *technical debt*. Namely, they classify these risks into the following categories: boundary erosion, entanglement, hidden feedback loops, undeclared consumers, data dependencies, configuration issues, changes in the external world, and system-level antipatterns. Next, I will outline the key points for each of these categories:

1. **Complex models erode boundaries:** The authors claim that without establishing strict abstraction boundaries for machine learning systems that prescribe the specific intended behavior, technical debt may increase significantly in the following ways.

    - Entanglement: Since machine learning systems mix signals together, isolating the improvements to the systems might be impossible, for example: if a feature or its distribution is changed, the rest of the features might be affected too.
    - Correction cascades: Learning a new model that's takes a previous model as input, to learn small corrections to problems that slightly differ from the original problem might lead to an increase in the cost of evaluating the improvement of the new model.
    - Undeclared Consumers: This debt arises when the output of a machine learning system is made widely available, which might lead to undeclared and most importantly uncontrolled users to use the output of the model as an input to another system. This is particularly important, because in a worst case scenario, this might lead to malicious uses of the model or to creating hidden feedback loops.

2. **Data dependencies cost more than code dependencies:** The authors present this as an analog to the dependency debt that arises from code complexity and technical debt in software engineering. However, for machine learning, the problem is related to identifying data dependencies that can be inappropriately converted into large dependency chains. A concept that particularly stood out for me was the underutilized data dependencies, which in this case are input signals that provide little incremental modeling benefit but might increase the vulnerability of the system.

3. **Feedback loops:** Here the authors pose the case for live models, which cannot be fully tested before being released as they rely on feedback loops that influence the model's self behavior. Here, they

propose that the frequency of model updates also increases the analysis debt of the model by making it difficult to detect and address feedback loops.

4. **ML-System anti-patterns:** Here the most important and common issue for me is the "Glue Code", which has become very common with public implementation of machine learning libraries or other preprocessing libraries. The problem with this type of implementations, or building a machine-learning system around a standard implementation is that it often requires freezing or reformatting the complete code to fit the peculiarities of a specific package.

5. **Configuration Debt:** It is known that machine-learning systems have many "tweakable parts" that range from hyperparameters to ways of representing the data. According to the authors, some developers treat these configurations as an "afterthought", where testing and verifying the configuration of the system is seen as a secondary task. To address this, they propose a list of principles for good configuration, that include simplicity of specifying configurations in term of changes from a previous version, robustness to "manual" errors, easy visualization of changes, easy verification of configuration facts, detectability of redundancy, code reviews.

6. **Dealing with changes in the external world:** Since most of the models that interact with the real world are not being challenged with a static setting, the authors propose that it is important to implement mitigation strategies for updating decision thresholds using simple evaluations on heldout validation data.

7. **Other areas:**

   - Data testing debt: A subset of the input data must be tested as a sanity check to keep a well-functioning system.
   - Reproducibility debt: It is important for us to be able to re-run experiments and get similar results.
   - Process management debt: In mature systems that include more than one model, it is important to address the distribution of resources to models that might have distinct business priorities.
   - Cultural debt: The authors propose that research teams in ML should not only value when the researchers achieve an increase in performance but also favor a culture where feature deletion, complexity reduction, reproducibility, stability, and monitoring are also rewarded.

4. Breck et al, The ML test score: A rubric for ML production readiness and technical debt reduction. BigData 2017: 1123-1132.

Related to the previous paper, Breck et al propose in this paper a set of 28 specific tests to help in the quantification of issues related to production readiness and reduction of technical debt in ML systems.

1. **Tests for Features and Data**

   - **Feature Schema Compliance:** Define and enforce schemas that capture feature expectations to automatically verify incoming data.
   - **All Features are Beneficial:** Ensure each feature contributes meaningful predictive power by evaluating their individual impact.
   - **No Feature's Cost is Too Much:** Assess the computational and maintenance costs of features relative to their benefits.
   - **Features Adhere to Meta-Level Requirements:** Enforce project-specific data constraints, such as privacy standards or feature availability.
   - **Data Pipeline Privacy Controls:** Implement safeguards to protect sensitive information throughout data processing stages.

- **Rapid Feature Addition:** Facilitate quick integration of new features to enhance system responsiveness and adaptability.
- **Comprehensive Testing of Feature Code:** Rigorously test all feature generation code to prevent undetected bugs and ensure data integrity.

2. **Tests for Model Development**

- **Model Specification Reviews:** Conduct code reviews and maintain version control for all model specifications to ensure reproducibility and auditability.
- **Correlation of Offline and Online Metrics:** Validate that offline performance indicators align with real-world impact metrics through controlled experiments like A/B testing.
- **Hyperparameter Tuning:** Systematically optimize hyperparameters using methods such as grid search or advanced tuning strategies to improve model performance and uncover reliability issues.
- **Awareness of Model Staleness:** Measure the effects of outdated models on prediction quality to determine appropriate update frequencies and prevent degradation.
- **Baseline Model Comparison:** Regularly benchmark against simpler models to validate the necessity and effectiveness of complex approaches.
- **Sufficient Quality Across Data Slices:** Ensure consistent model performance across various subsets of data to identify and address localized degradation.
- **Inclusion Considerations:** Test models for biases and ensure equitable performance across different user groups and data segments.

3. **Tests for ML Infrastructure**

- **Reproducible Training:** Strive for deterministic training processes to facilitate debugging and ensure consistent model outcomes by removing sources of nondeterminism.
- **Unit Testing of Model Specifications:** Implement lightweight tests for model code to detect integration and implementation errors early, enhancing development speed and reliability.
- **Comprehensive Pipeline Integration Testing:** Automate tests that cover the entire ML pipeline—from data assembly and feature generation to model training and deployment—to catch issues before they reach production.
- **Pre-Serving Model Quality Validation:** Automatically verify model performance metrics before deployment to prevent introducing underperforming models into production environments.
- **Debugging Support for Models:** Provide tools that allow detailed inspection of model computations, enabling efficient diagnosis of unexpected behaviors and facilitating robust debugging processes.
- **Canary Deployment of Models:** Gradually roll out new models to a subset of traffic to monitor performance and catch issues without widespread impact, ensuring safer deployments.
- **Rollback Mechanisms:** Establish swift procedures to revert to previous model versions in case of deployment failures or performance drops, minimizing downtime and maintaining system reliability.

4. **Monitoring Tests for ML**

- **Dependency Change Notifications:** Set up alerts for alterations in data sources or dependencies that could affect model behavior, ensuring timely awareness of potential issues.
- **Data Invariant Checks:** Continuously validate that incoming data adheres to predefined schemas and expectations to detect anomalies and maintain data integrity.
- **Training and Serving Consistency:** Monitor for discrepancies between training and inference pipelines to prevent skewed predictions and ensure consistent model behavior.

- **Model Staleness Alerts:** Track the age of deployed models and associated data dependencies to ensure timely updates and prevent performance degradation due to outdated models.
- **Numerical Stability Monitoring:** Detect and alert on irregular numerical values within models, such as NaNs or infinities, to prevent computational issues and maintain model reliability.
- **Performance Regression Tracking:** Keep an eye on computational metrics like training speed, inference latency, and memory usage to maintain system efficiency and promptly address regressions.
- **Prediction Quality Surveillance:** Continuously assess prediction accuracy on live data to identify and address degradation promptly, ensuring sustained model effectiveness.

Finally, when applying the ML Test Score rubric to 36 diverse ML teams, the authors found that these rubrics provided valuable insights into enhancing production readiness and reducing technical debt. First, the use of structured checklists enabled teams to identify untested code and address gaps in monitoring systems, ensuring more reliable model performance. Second, addressing dependency issues underscored the necessity for clear communication and robust data management to prevent hidden couplings and inconsistencies. Third, the adoption of integration testing frameworks like TFX demonstrated significant improvements in pipeline reliability and maintainability. Overall, these tests offer a comprehensive roadmap for engineering more dependable and scalable machine learning systems, effectively mitigating technical debt and fostering sustainable development practices.