

Application of deep learning and artificial intelligence in a congested signals environment

Jacob Brown

Defence Science Technology Laboratory

University of Wales Trinity Saint David, School of Applied Computing

Project Ambition

The purpose of the project was to demonstrate the feasibility of employing a Deep Learning technique to better optimise broadcast power across an available area of the radio frequency spectrum.

This was realized with the implementation of a Deep Q Network, this is a technique that uses Q values to build associations and predictions for future observations, performed by a Convolutional Neural Network.

The nature of the Rainbow Deep Q Network results in a series of techniques being employed as components inside of a larger net.

Model Attributes included

- Distributed Reinforcement Learning
- Duelling Deep Q Networks
- Double Deep Q Networks
- Prioritized experience relay

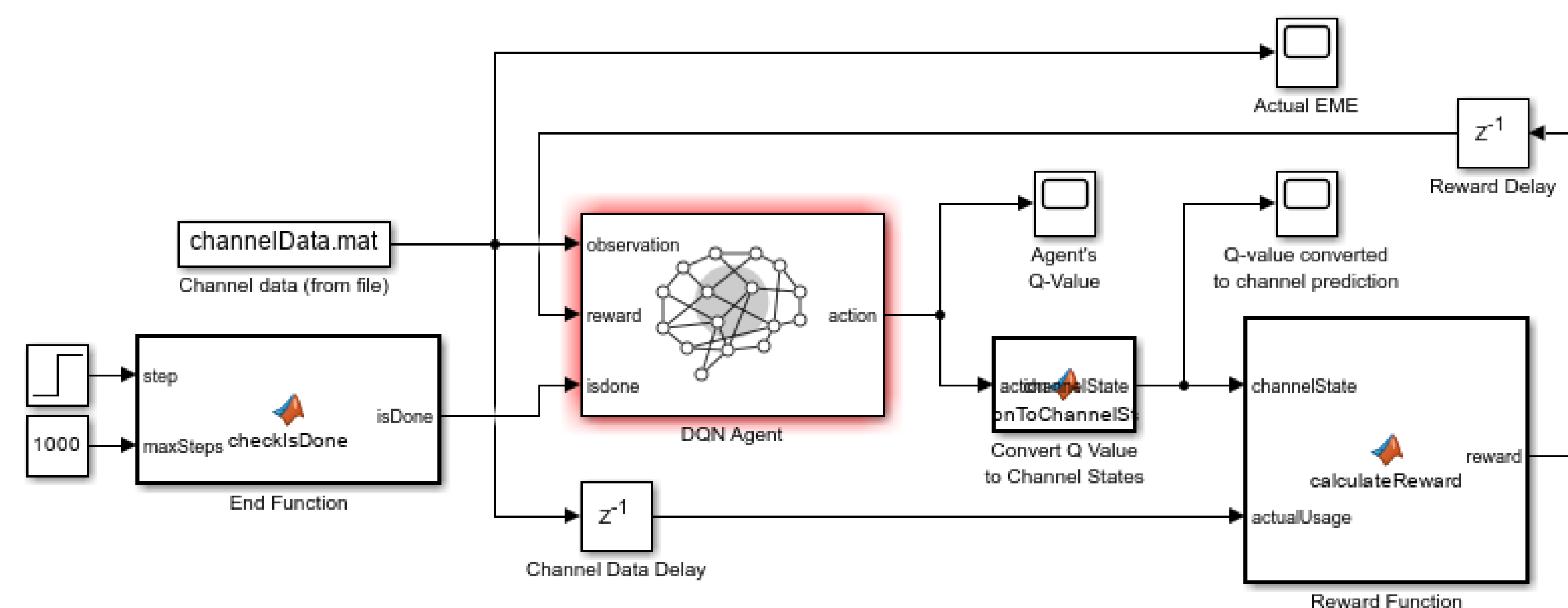
Overview

Following the project ambition being established and the network being selected, the need to establish a data baseline was formed. This took the form of a series of controlled data collects over the course of several months to best characterize the environment within several hundred metres of a telecommunications tower.

This then allowed for a controlled environment wherein individual devices were introduced to stimulate activity and demonstrate what various transmitters and received appeared as relative to the noise floor. This data was then recorded and detailed to create a robust training data set.

Data is then added to the Rainbow Deep Q Network as training and test data depending on the circumstance, until the accuracy trends to such a level that it can be assumed to perform to a relatively high degree of efficacy. Then being output as a series of images resembling a frequency/time capture and a binary activity graph for a given point in time, demonstrating when the network has identified a high level of activity and then deems that the activity is significant enough compared to the general noise floor that rerouting would be preferable to a continued transmission.

Ultimately, this results in a reduced energy consumption of a telecommunications tower by increasing usage of areas with limited activity without negatively – and sometimes even positively, impacting quality of service from that tower.



Early model of Simulink based Deep Q Network.

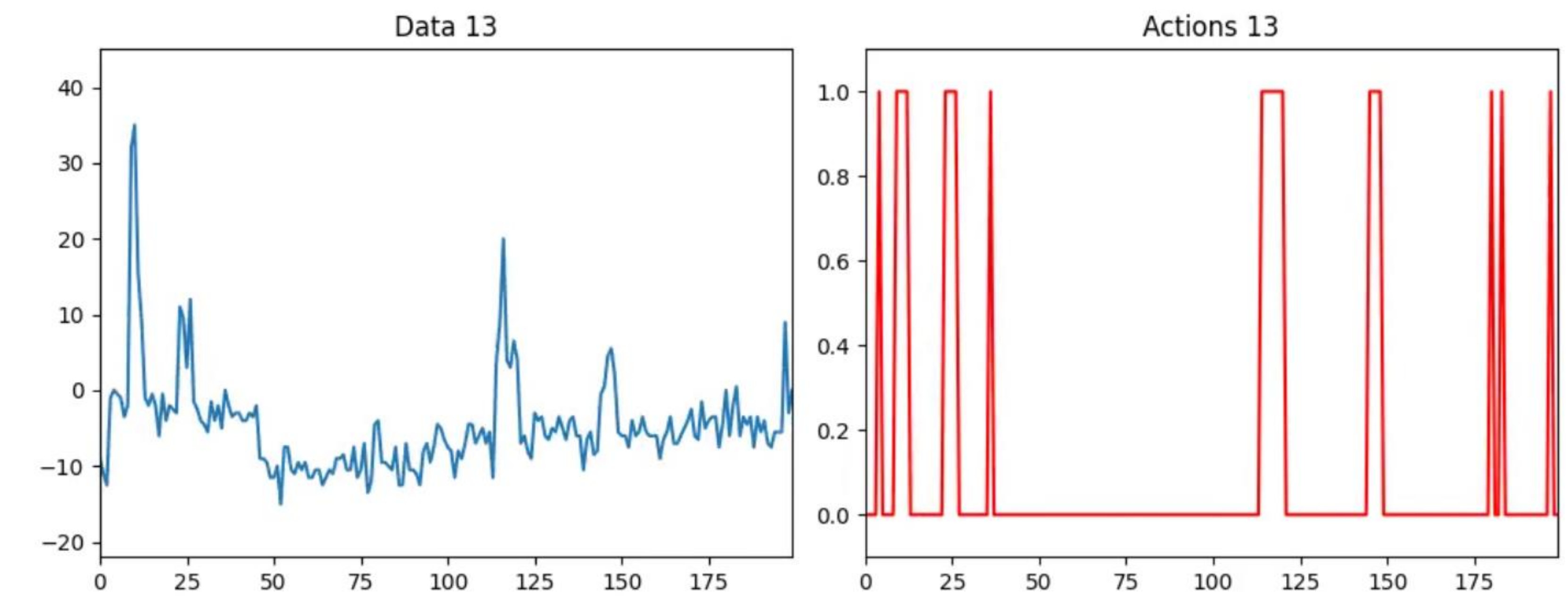


Figure of incoming data (Left) against model response (Right)

Code Structure and Basis

The code structure, was initially split between two separate forks, a Python and MATLAB version, the MATLAB being used for conceptual and system wide emulation while the Python fork was used for technical development.

This ultimately proved too strenuous and hard to manage, beginning to devolve into a plethora of poorly cross compatible versions that ultimately resulted in the Python fork being selected as the primarily development vehicle.

This used syntax from Pytorch, the industry standard for research and development of AI tools and one of the most well documented resources for AI development in the open-source community.

```

stateTensor = torch.FloatTensor(stateSeq.unsqueeze(0).to(device)
actTensor = torch.FloatTensor(action).to(device)
rewardTensor = torch.FloatTensor([reward]).to(device)
nextTensor = torch.FloatTensor(nextState).to(device)

qValues = dqn(stateTensor)
seqNext = numpy.concatenate((stateSeq[1:], [nextState]))
nextSeqTensor = torch.FloatTensor(seqNext).unsqueeze(0).to(device)
nextQs = targetDQN(nextSeqTensor).detach()

qVal = torch.sum(qValues * actTensor)
nextQVal = rewardTensor + gamma * torch.max(nextQs)

qVal = qVal.view(1, 1)
nextQVal = nextQVal.view(1, 1)

loss = criterion(qVal, nextQVal)

optimizer.zero_grad()
loss.backward()
optimizer.step()

episodeLoss += loss.item()
qValSum += qVal.item()

stepCount += 1
sumReward += reward
stateSeq = numpy.concatenate((stateSeq[1:], [nextState]))

episodeMask.append(action)

rewardPerEpisode.append(sumReward)
trainingMask.append(episodeMask)
banePerEpisode.append(episodeLoss / stepCount)
meanQ.append(qValSum / stepCount)
epsilons.append(epsilon)

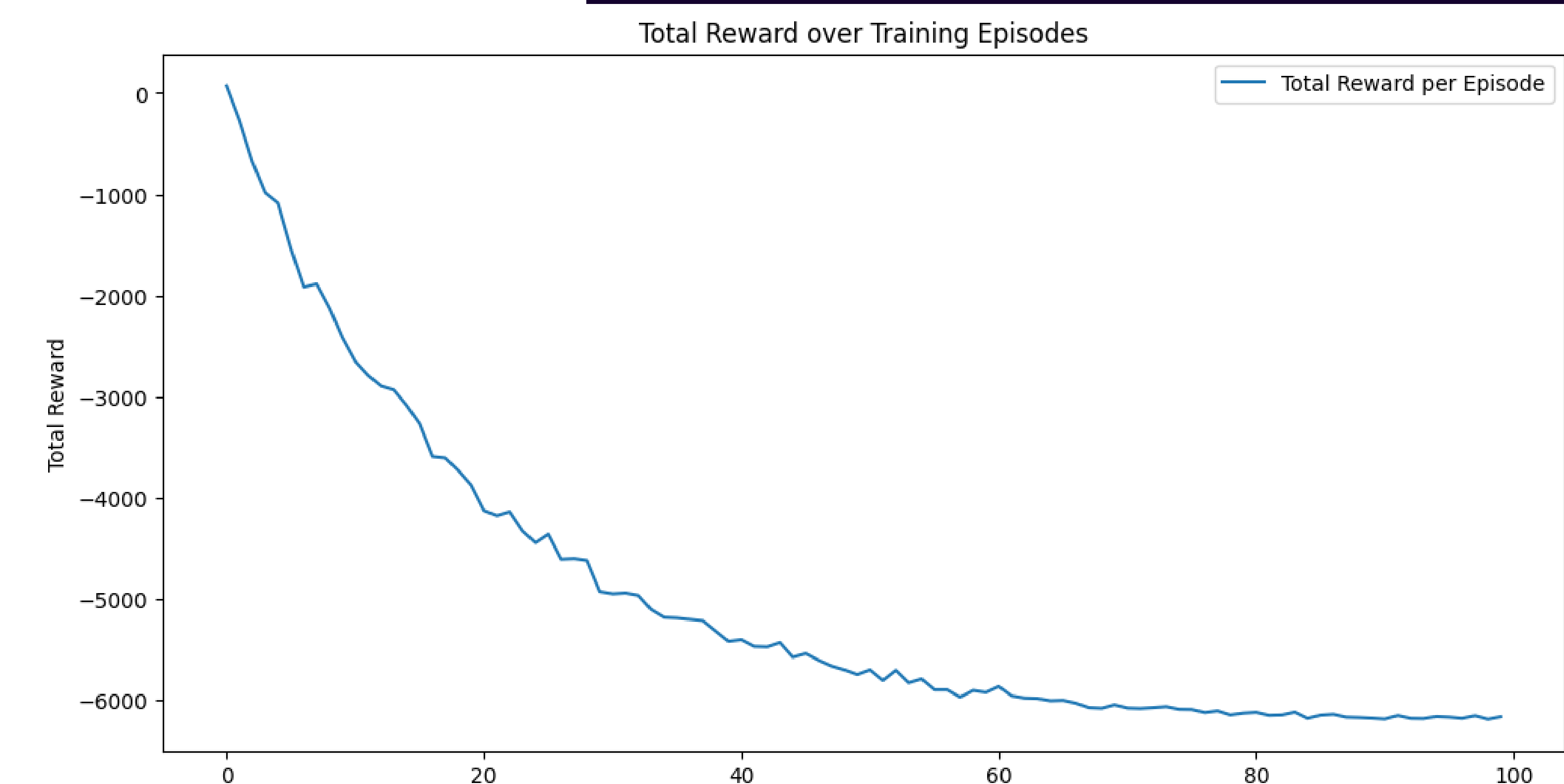
if epsilon > epsilonMin:
    epsilon *= epsilonDecay

if episode % targetUpdate == 0:
    targetDQN.load_state_dict(dqn.state_dict())

print("Episode: {}".format(episode))
print("Total Reward: {}, Average Loss: {}".format(banePerEpisode[-1].data[0], Epsilon: epsilon, Steps: stepCount),

```

Code snippet from the model training script



Total reward graph of the Deep Q Network over the course of 100 training episodes

Training Validation

To develop a robust validation process, a series of data analytics were collected throughout the training process. This took the form of analysing the development of the Q values, Loss factor, Epsilon and Reward values to validate that training progressed effectively.

This validation allowed the operation of the model to go forward and be applied against real world data with a high level of effectiveness, containing a reasonable and robust framework that enabled easy debugging and maintenance of the system even after its version completion, ensuring a long lifespan of the model.