


## Assignment Cover Sheet

### School of Computer, Data, and Mathematical Sciences

Student Name	Sandipbhai Pravinbhai Viradiya
Student Number	19921091
Unit Name and Number	301119: Advanced Machine Learning
Title of Assignment	Assignment 1
Due Date	25 Oct 2020
Date Submitted	25 Oct 2020
<b>DECLARATION</b> <i>I hold a copy of this assignment that I can produce if the original is lost or damaged.</i>  <i>I hereby certify that no part of this assignment/product has been copied from any other students work or from any other source except where due acknowledgement is made in the assignment. No part of this assignment/product has been written/produced for me by another person except where such collaboration has been authorised by the subject lecturer/tutor concerned.</i>  Signature:  .....  <i>(Note: An examiner or lecturer/tutor has the right not to mark this assignment if the above declaration has not been signed)</i>	

	Task 1	Task 2	Task 3	Task 4	Bonus	Total
Mark						
Possible	15	10	15	10	?	50

The maximum points possible for this assignment is 50 (including any bonus points).

## Table of Contents

<b><i>Task 1: Train a CNN to predict object positions .....</i></b>	<b><i>3</i></b>
<b><i>Task 2: Train a convolutional autoencoder .....</i></b>	<b><i>6</i></b>
<b><i>Task 3: Create a RL agent for Minipong (level 1).....</i></b>	<b><i>8</i></b>
<b><i>Task 4: Create a RL agent for Minipong (level 3).....</i></b>	<b><i>11</i></b>

## Task 1: Train a CNN to predict object positions

First, I created the dataset by running the sprites.py file. It generated four files named trainingpix.csv with 676 samples and testingpix.csv with 169 samples. Generated the label files as traininglabels.csv and testinglabels.csv with 3 labels (x/y/z).

### Data:

The dataset we have is of 255 in one dimension which is actually a 15 \* 15 image. It contains x between 1 and 13, y value between 1 and 13 and z value between 0 and 5.

### Preparing the data:

Our data is in a pixel which has been given in one row as 255 (15\*15). So, it needed to be reshaped 15\*15. I've used NumPy to reshape it and converted it into torch tensor. I tried by

### Model:

Please note that I've implemented both Keras (Assignment 1.ipynb) file and PyTorch (Assignment1.py). For both of this, the model implementation is almost the same as described below. The Keras is predicting x only, whereas, in PyTorch, it is predicting x,y and z from the same model.

The model contains 2 convolutional layers with two max-pooling layers, after that two linear layers where the second linear layer do the prediction job.

As our image is not RGB, this is going to be just a one input channel for the first layer. I tried giving a different number of neurons but got the best result by the below models.

### The first model to predict x only (Keras).

This model uses the first layer as convolutional with 1 input and output 32 neurons with 3x3 kernel. It starts with 15x15 kernel. After that, we are applying relu as an activation function and apply max pooling with 2x2 kernel, and it gives us the output as 6x6. After that, I've used another convolutional layer with 32 input and 64 output neurons with a 3x3 kernel. Again, applied relu and max pooling was set with 2x2 kernel. It gave 2x2 as output, after that made it linear, which gave us 256 channels and gave this to a fully connected neural network with 128 as output channel and applied relu. This 128 given as input to a linear neural network and predicted x range from 1 to 13. As Python start it from 0, we needed to provide 14 as the output. We could have subtracted one from every value of x and predict only 13.

In Keras for predicting x only, I've tried RMSprop and Adam as an optimizer in the prediction of x. As a **loss function**, as we have 1 to 13 index, which can be considered as a category, I used **SparseCategoricalCrossentropy**. Here I've not encoded the x value into one-hot encoding. So, categorical\_crossentropy can't be used as a loss function.

I've used 20 epochs here, but it could have work with just 10 epochs as we are getting 100% accuracy after 8 epochs. It took **4.14 seconds** in the prediction of x only.

**Accuracy** on the test data was also 100% for prediction of x only.

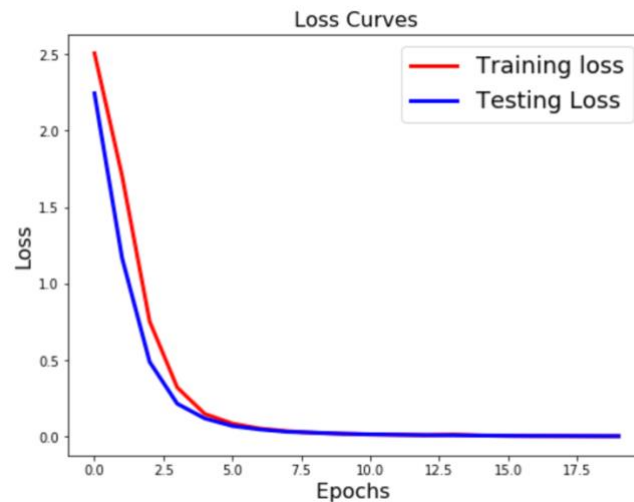


Figure 1: Training and Testing Loss (Predicting x only)

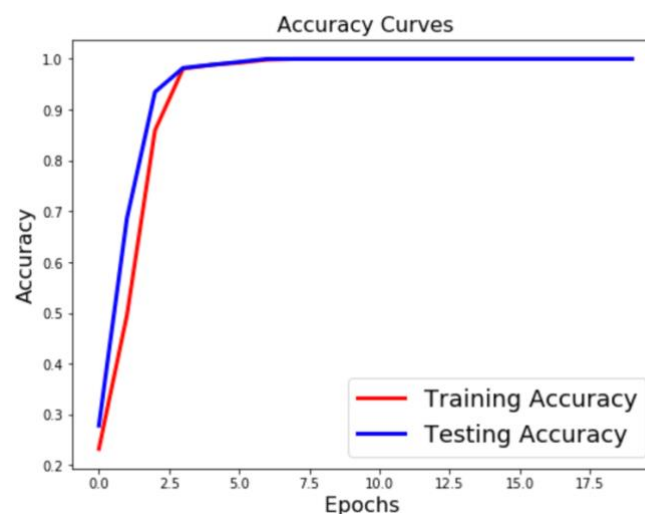


Figure 2: Training and Testing Accuracy (Predicting x only)

## The second model to predict x/y/z (PyTorch)

I've used almost the same model as I used in the Keras. The first layer as convolutional with 1 input and output 32 neurons with 3x3 kernel. It starts with 15x15 kernel. After that, we are applying relu as an activation function and apply max pooling with 2x2 kernel, and it gives us the output as 6x6. After that, I've used another convolutional layer with 32 input and 64 output neurons with a 3x3 kernel. Again, applied relu. The difference between Keras and Pytorch is here. I have not used the max-pooling after the second convolution. So,

instead of 2x2 we got in Keras, here, we have 4x4 as an output, after that made it linear, which gave us 1024 channels and gave this to a fully connected neural network with 128 as output channel and applied relu.

Here, we have one dropout layer with 0.25 to prevent a model from overfitting.

Now, from 128, we have created three separate linear layers. One predicts x (1 to 13), one predicts y (1 to 13) and one predicts z (0 to 4). Here, the model is not returning x only, but it is returning dictionary having x, y and z. The difference is here in training. Before we were trying to decrease the loss of x only, here, the model gets train once, and then a loss will be counted individually of x, y and z and then the sum of the losses will be given in backpropagation to the optimizer to optimize it. Below is the screenshot of that part of code to understand it better.

```
# Fully connected network
self.fc1 = nn.Linear(64 * 4 * 4, 128)
self.fc2 = nn.Linear(128, 14) # Predict x
self.fc3 = nn.Linear(128, 14) # Predict y
self.fc4 = nn.Linear(128, 5) # Predict z
```

Figure 3: Fully connected network part of CNN

```
x = self.fc2(x_final) # x prediction
y = self.fc3(x_final) # y prediction
z = self.fc4(x_final) # z prediction

return {"x":x,"y":y,"z":z}
```

Figure 4: Implemented forward method where x, y and z are returned

```
# forward + backward + optimize
outputs = net(x_train_tensor.data)
loss_x = criterion(outputs['x'], y_train_tensor[:,0])
loss_y = criterion(outputs['y'], y_train_tensor[:,1])
loss_z = criterion(outputs['z'], y_train_tensor[:,2])

# Calculating total loss
total_loss = loss_x + loss_y + loss_z
total_loss.backward()
optimizer.step()
```

Figure 5: PyTorch where x, y and z are being predicted

I've used **cross entropy loss** as a loss function in this. It predicts the probability of each value of x from 1 to 13. After that, the maximum probability we get is the predicted x or y or z value.

I tried **SGD** (stochastic gradient descent) optimizer with a learning rate of 0.01 and momentum of 0.9, but the accuracy was around 98% for x, y and z with **170 epochs**.

So, I changed it to **Adam** with the same learning rate 0.01, and I got 99-100% training accuracy for x, y and z with only **50 epochs**. The testing accuracy was also around 98-100% for all of them.

For testing, I have manually passed the testing data with x, y and z, and from the response, I used `accuracy_score` function of sklearn library to calculate the accuracy.

In PyTorch, we need to loop through the epochs and train the model, count the loss, backpropagate it and again optimize it. In Keras, all this is built in the library itself.

It was taking 3.25 to 3.75 seconds to train x, y and z. But, as predicting x only gives us 100% training and testing accuracy, we could have predicted x, y and z individually and combine them.

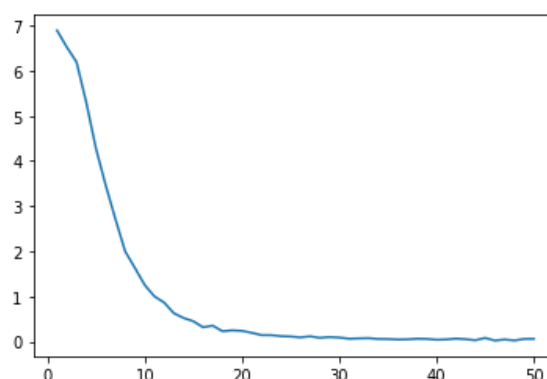


Figure 6: PyTorch training loss

As we can see that training loss was started high at 6.894, but over the time it got to 1 after 10 epochs. After 30, it was almost 0.06.

## Task 2: Train a convolutional autoencoder

In this task, we are just going to use `trainingpix` to train, compress and decompress the image.

In this, I've used a simple network where and encoded the image from 15x15 to 7x7 and decoded back to 15x15. Below is the model. I used a little big size here as we have only 4 parts in the image and it will get spread it will compress the image too much.

The first layer is a convolutional layer with input as 1 channel and output as 16 channels with 3x3 kernel, a stride of 2 and padding of 1. After this relu will be applied and it will get us 8x8 image. After that, I've used another convolutional layer with input as 16 channel and output as 8 channels with 3x3 kernel, a stride of 1 and padding of 1. Again, the relu function was applied after that max-pooling with 2 X 2 and stride as 1 gave us 7x7 encoded image. So, encoder gave us 8 channels with 7x7 images.

In the decoder, transposed convolution takes 8 channels to 16 channels using a 3x3 kernel. This takes each pixel to a 15x15 image Stride 2 provides a 15x15 image. The next convolution layer takes 16 channels as input and gave back 8 channels with 3x3 kernel, stride and padding as 1. It gave the image with the same 15x15. Next layer took 8 channels and gave 1 channel as required with 3x3 kernel, stride and padding as 1 again as we did not want to lose the dimension.

As we will get 225 pixels as the prediction and original data, we can use **MSELoss**, which is mean squared error loss between our prediction and original data. MSELoss is actually a squared L2 norm. Below is the loss graph.

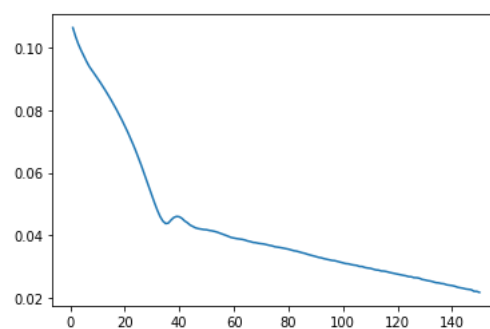
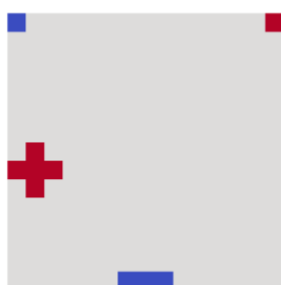


Figure 7: Loss over the epochs

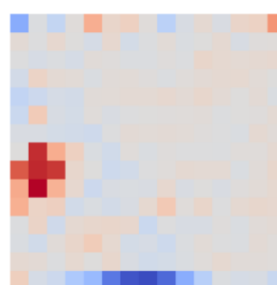
So, our encoded image is 7x7 with **150 epochs**.

Here is the good and bad example of the encoded images:

**Good Examples:**



**Original**



**Encoded**

Figure 8: Good example

### Bad Examples:

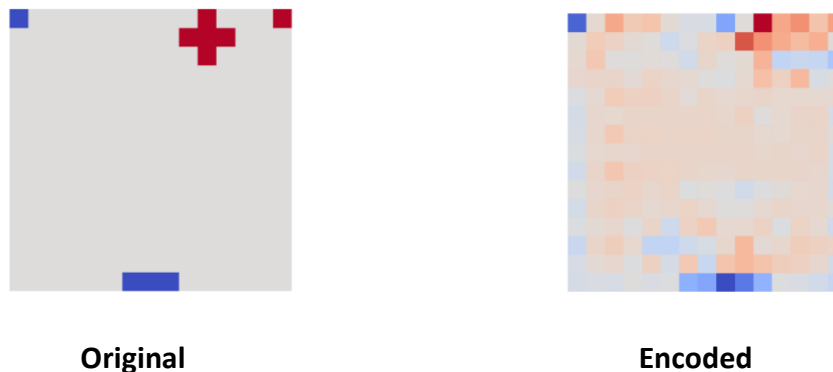


Figure 9: Bad example

**What would be the absolute minimal size of the hidden layer representation that allows perfect reconstruction of the original image?**

The minimal size of the hidden layer should be more than the input size. There is not any perfect measurement, but taking it to more than double size of the channel would be helpful to let the model learn and understand the image.

The number of hidden layers also depends on the problem. For example, in our scenario, we got it using total 5 layers. Ideally, it should be at more than the output we want. For example, if we want 4 predictions to get out, the hidden layers should get us more than 4 channels so that in the last layer we get 4 out from it.

So, a number of channel and number of hidden layers depends on the problem, but it should be minimum more than required input (In first hidden layer) and output (last hidden layer or fully connected network).

### Task 3: Create a RL agent for Minipong (level 1)

The manual strategy is straightforward. We are getting 0 if + is relative to the centre of the paddle, 1 if + is on the left side, 2 if + is on the right side.

Our actions are,

- 0: Do Nothing
- 1: Move Left
- 2: Move Right



So, if the state is positive, that means the + is at the right side of the paddle. We need to move the paddle to the right (action = 2) side in order to get to the +. Same way, if the state is negative, action will be 1, and if the state is not negative or positive, we do not need to do anything which leads us to action as 0.

```
def mypolicy(state):  
    """  
    The state is 0: + is in the middle, do nothing  
    The state is negative: + is on the left, move left  
    The state is positive: + is on the right, move right  
  
    0: do nothing  
    1: move left  
    2: move right  
    """  
    action = 0 # Do nothing  
  
    if state > 0:  
        action = 2  
    elif state < 0:  
        action = 1  
  
    return action
```

I've used **DQN** for this question. The DQN contains 3 linear layers. I've used 150 hidden layer size. MSELoss as the loss function and Adam as an optimizer with a learning rate of 0.01.

Our network looks like  $x \rightarrow \text{fc1} \rightarrow \text{ReLU} \rightarrow \text{fc2} \rightarrow \text{ReLU} \rightarrow \text{fc3} \rightarrow y$

It is taking 1 as input channel and 150 as output channel. After relu, it goes to another linear layer with 150 input channels and 300 output channels with relu again. From 300 input to it gives 3 actions now. Herewith network and forward function, we will have two more functions named update and predict. An update will **Update the weights of the network given a training sample** and **predict will compute Q values for all actions using the DQL**.

Here, for the first level, we are getting only one state which is dz, so, our number of states will be 1. We've three action 0 (do nothing), 1 (move left) and 2 (move right) which makes a number of actions as 3 for further coding.

We've also been asked to give  $\epsilon = 1$  and reduce it to 0.1. **Epsilon prevents the model from getting overfitted**. Use of the  $\epsilon$  is to select the action or select from Q. If the random value is less than epsilon, we will select random action. In contrast, if it is more than  $\epsilon$ , we will select the maximum value from the prediction for the specific state as an action. This will be used in the below description.

After selecting the action, we pass it to our MiniPong environment (step function) and get the reward, next state and if the terminal state has been reached. We update the reward, and if the terminal state has not been reached, we update the network weights using  $R + \gamma \max(S')$ . For  $\max(S')$ , we need to again predict from the neural network using the next state we got in the previous step. As explained above, epsilon will get updated, and it will

decrease with decay to 0.1. After training it for 750 episodes (76.41 seconds), I got below the graph where we got reward till 156. We can see that it is fluctuating too much. We can also see the average is going up slowly with the orange line in the left side graph, which indicates that our agent is learning.

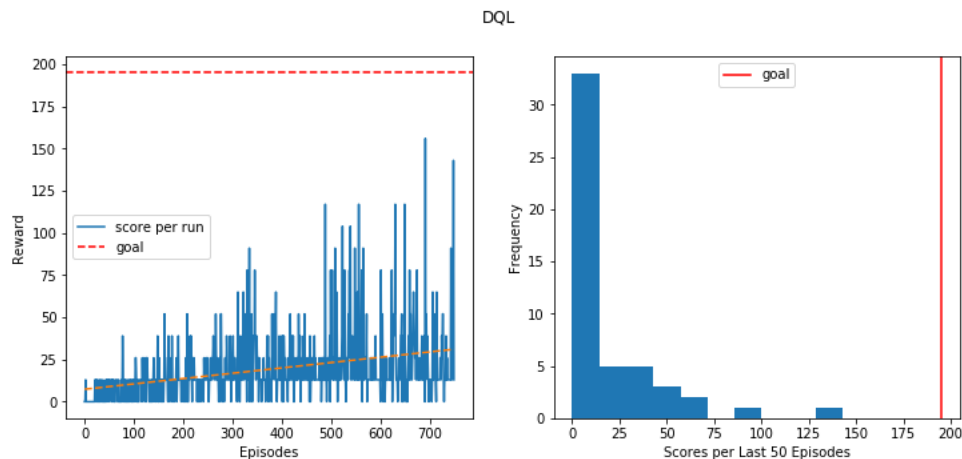


Figure 10: Training reward (750 episodes, 76.41 seconds)

I trained it for 750 episodes, and it took 76.41 seconds to get trained.

After this, as required, I ran it for 50 episodes and used the trained policy to test how it behaves and what reward it gives. Surprisingly it behaved amazing and gave me high rewards. It gave me 325 highest rewards. Below is the graph which represents the testing reward. We can see that most of the time we got a reward less than 50, but we can see the spike at 100 too, which is good. We've got more than 100 rewards many a time.

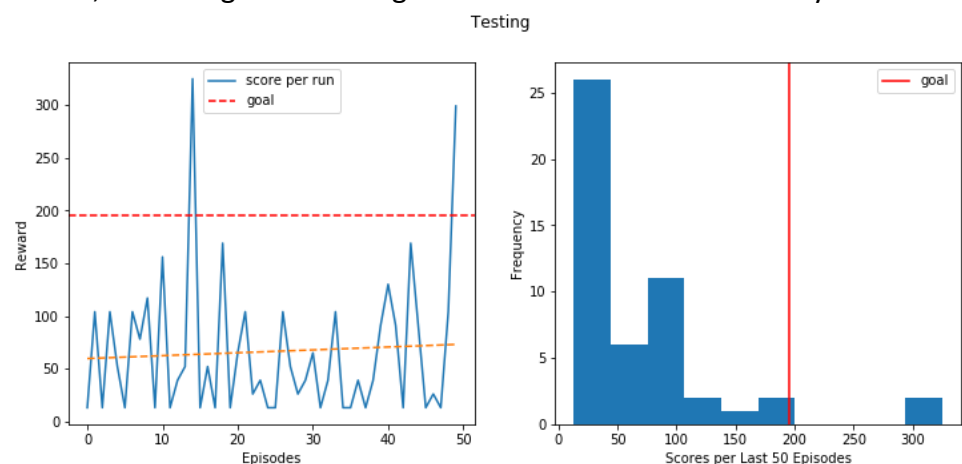


Figure 11: Testing reward

Finally, as required, I've calculated Test-average and Test-StandardDeviation of the testing reward.

**Test-average: 66.3**

**Test-StandardDeviation: 67.41**

If I would have initialised pong with `pong = Minipong(level=2, size=5)`, this information will definitely **help us**. It would have helped us because currently, we are just having `dz`, which is a relative distance in x-coordinate between paddle and ball. If we pass the level as 2, we will get y-coordinate of the ball and `dz`. This help the model understand the position better and get us a more accurate result.

## Task 4: Create a RL agent for Minipong (level 3)

I tried both of the approaches for this question **Policy Gradient** and **TD**. I've submitted the code file of both of them.

As we are using level 3, we are getting 3 states now, `y`, `dx`, `dz`. These are `y`, the ball y-coordinate; `dx`, the change in ball x-coordinate from the last step to now; and `dz` (same as previous levels).

For TD, I've used the same approach as Q3, but now the `q_learning` function has been removed and added the code to train it and count the running reward as given in `minipong.py` file.

For policy gradient **REINFORCE**, I've used two given 3 states as input and 128 as output and given 128 as input again to get 3 actions out using the softmax. In policy gradient now the action selection has been changed. The three of the state we pass to the neural network, and it gives us three probability of respect to each action. We take out the action using the sample method and pass it to the step function of our environment. Till we get the terminal state, this process is being repeated. As we get a terminal state, the policy will get updated. We used the rewards and applied the discount factor in the update. After that, we've normalized it and to avoid division by zero, we have used one value at the time of normalisation. After that, log probability is applied with reward and saved. The loop gets repeated. Somehow, my cumulative reward is going to 70 only. Below is the training reward graph from policy gradient.

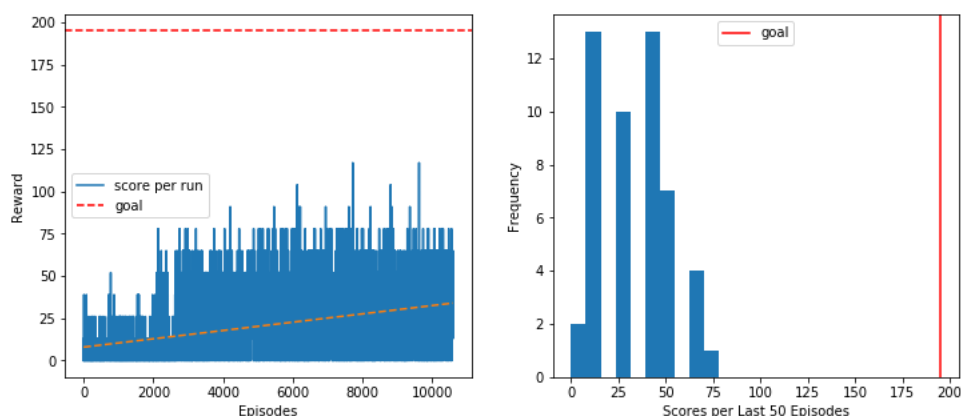


Figure 12: Training reward using level 3

I simulated 50 episodes for the testing after this in policy gradient, but the reward is almost stable or slightly increasing some time, as shown in the below image. (go to next page)

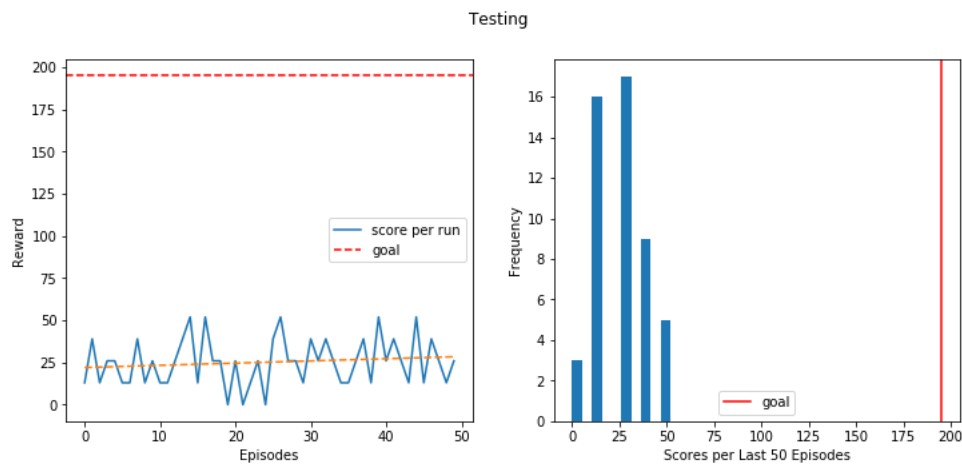


Figure 13: Testing reward using level 3

**Test-average: 25.22**

**Test-StandardDeviation: 13.98**

I tried by changing the optimizer, the learning rate, changing the linear network in and out channels, but the accuracy was not differing too much with a change.