Calculating gradients of variables in Pytorch.

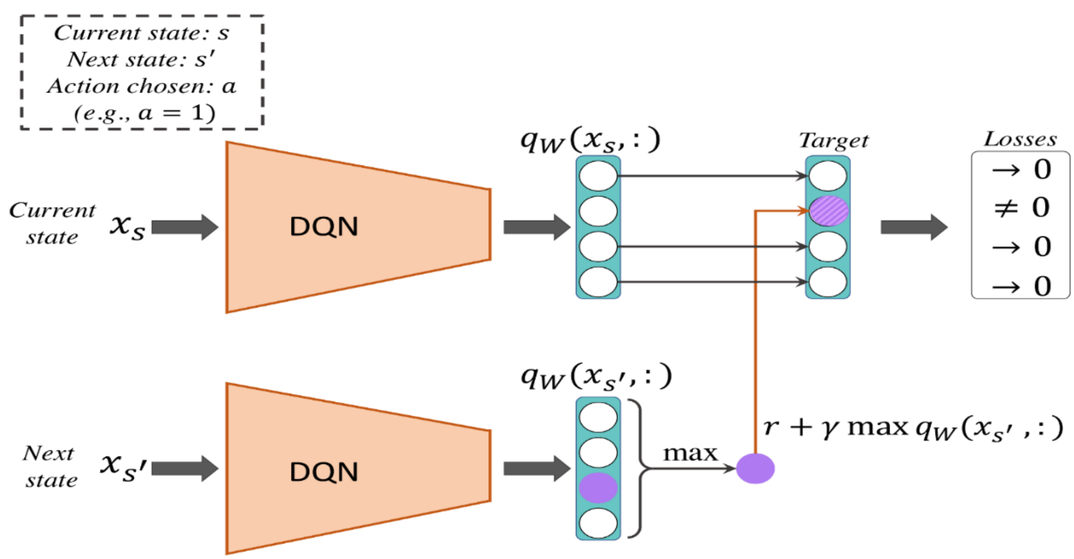
The output of the NN will be a matrix of size (batch_size, action_space)

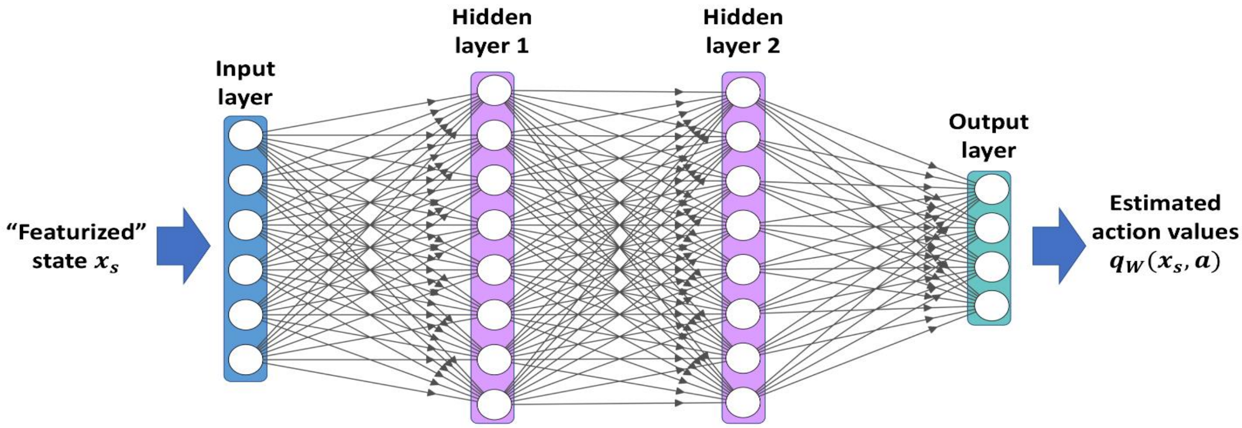
Action space could be "up","down","left",etc..

The Task is framed as a regression problem using the MSE loss function. This means that the loss takes in single values, instead of tensors. The loss takes in 2 scaler values

a. The output value given from the NN for the action that was taken
b. The target value, which is constructed by the Bellman equation.

On back-propagation, the only node that should be used to back-propagate the error is the one representing the action that was taken, this is the only node with a non-zero loss, and thus a non-zero gradient. The only weights to be updated, between the hidden and output layer, will be the ones projecting onto that node.





Tensor_states (Batch_size,action_space)

4.0	2.5	2.9
1.2	1.5	1.9
5.0	5.5	5.9
4.8	2.5	2.9
4.0	2.8	3.9

Actions (Indices)

2
1
1
0
1

Result of gather() Predicted_state_action_values

4.0	2.5	2.9
1.2	1.5	1.9
5.0	5.5	5.9
4.8	2.5	2.9
4.0	2.8	3.9

2.9
1.5
5.5
4.8
2.8

Loss = (predicted_state_action_values,target_state_action_values)

Gradients for each sample. Just an example, nothing was calculated.

0	0	1,3
0	0.63	0
0	2.01	0
3.03	0	0
0	2.02	0

Gather() operation on predicted_state_values()

```
predicted_state_action_values = main_net(tensor_states).gather(1, tens
or_actions
```

The loss/gradients for the non-yellow elements will be 0. (This is done in the 2nd code snippet)

Predicted action values for 1 state

Predictions =

5.0	5.5	5.9	5.0

Index = 1

Prediction = $\boxed{5.5}$

Target = $\boxed{2.9}$

Output = $\boxed{\text{MSE(prediction, target)}}$

Output.backward() # Calculate the gradients for the elements that were fed into the loss function.

1st Code Snippet

Python 3.7.9 (default, Aug 31 2020, 17:10:11) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32

```
>>> import torch
>>> import torch.nn as nn
>>> predictions = torch.tensor([0.235,1.200,1.321], requires_grad = True)
>>> predictions
tensor([0.2350, 1.2000, 1.3210], requires_grad=True)
>>> targets = torch.tensor([2.335,0.890,1.154])
>>> targets
tensor([2.3350, 0.8900, 1.1540])
>>> prediction = predictions[1]
>>> target = targets[1]
>>> prediction
tensor(1.2000, grad_fn=<SelectBackward>)
>>> target
```

```
tensor(0.8900)
>>> loss = nn.MSELoss()
>>> output = loss(prediction, target)
>>> output.backward()
>>> prediction.grad
>>> target.grad
>>> predictions.grad
tensor([0.0000, 0.6200, 0.0000])
>>> targets.grad
>>> 2 * (prediction - target)   [The Derivative of the MSE w.r.t.
```
prediction works out to be 2(prediction-target)]

```
tensor(0.6200, grad_fn=<MulBackward0>)
```

2nd Code snippet

Computing the gradients with the gather() function.

Same approach, but using the gather() function. The gather function is used instead of simply indexing.

```
(base) C:\Users\billy>python
Python 3.7.9 (default, Aug 31 2020, 17:10:11) [MSC v.1916 64 bit (AMD64)] :: Anaconda, Inc. on win32
>>> import torch.nn as nn
>>> import torch
```

>>> predictions = torch.randn(3,requires_grad = True)

>>> predictions

tensor([0.9243, -0.5061, -0.0116], requires_grad=True)

>>> **prediction = torch.gather(predictions,0,torch.tensor([0]))**

>>> prediction

tensor([0.9243], grad_fn=<GatherBackward>)

>>> target = torch.tensor([2.565])

>>> target

tensor([2.565])

>>> prediction.size()

torch.Size([1])

>>> target.size()

torch.Size([1])

>>> loss = nn.MSELoss()

>>> output = loss(prediction,target)

>>> output.backward() # Calculate gradients

>>> predictions.grad

tensor([-3.2815, 0.0000, 0.0000])   # The gradient is calculated and stored in a specific index. The optimizer then uses these gradients to update the values e.g. value += learning_rate * value.grad

>>> target.grad (This is empty as the 'target' is never updated)

>>> 2*(prediction-target)   [This again is the partial derivative w.r.t. the MSELoss function.]

```
tensor([-3.2815], grad_fn=<MulBackward0>)
```

---

requires_grad: This member, if true starts tracking all the operation history and forms a backward graph for gradient calculation.

grad: grad holds the value of gradient. If `requires_grad` is False it will hold a None value. Even if `requires_grad` is True, it will hold a None value unless `.backward()` function is called from some other node. For example, if you call `out.backward()` for some variable ***out*** that involved ***x*** in its calculations then `x.grad` will hold **∂out/∂x**.

`Backward()` simply calculates the gradients

The .backward() function is called on the error!

FULL EXAMPLE OF THE Q-LOSS FUNCTION CODE

This is the ‘Naïve’ version which loops every sample in a batch.

```python
criterion = nn.MSELoss()

def naive_dqn_loss_V2(batch, main_net, target_net, gamma, batch_size = 32, device = "cuda", optimizer = 'opt'):

    tensor_states,tensor_next_non_final_states,tensor_actions,tensor_rewards,non_final_mask = batch

    # create predicted (NN output)
    predicted_state_action_values = main_net(tensor_states) # dims = batch_size,number_of_actions

    next_state_values = torch.zeros((batch_size,predicted_state_action_values.size()[1]),  device=device)

    #print(next_state_values)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(tensor_next_non_final_states) #
        next_state_values = next_state_values.detach()

    next_state_values = next_state_values * gamma

    batch_loss = 0.0

    for prediction, reward, action,  tensor in zip(predicted_state_action_values, tensor_rewards,tensor_actions, next_state_values):

        target = tensor.max() + reward


        target = target.detach() # detach, no gradient flow to the ground Truth!!!

        prediction = prediction[action]

        # scalar, NOT vector inputs to the loss function!
        sample_loss = criterion(prediction,target)

        batch_loss += sample_loss

    return batch_loss / batch_size
```

The batch version of the Loss, using the gather() function on the tensor_states() matrix.

```
def simple_dqn_loss(batch, main_net, target_net, gamma, batch_size = 32, device="
cuda"):

    tensor_states,tensor_next_non_final_states,tensor_actions,tensor_rewards,non_
final_mask = batch

    predicted_state_action_values = main_net(tensor_states).gather(1, tensor_acti
ons.unsqueeze(-1)).squeeze(-1)

    next_state_values = torch.zeros(batch_size, device=device)

    with torch.no_grad():
        #compute next state values using Target network
        next_state_values[non_final_mask] = target_net(tensor_next_non_final_stat
es).max(1)[0] #only take the values and not the indices
        #detach from pytorch computation graph
        next_state_values = next_state_values.detach()


    #target values using Bellman aproximation
    target_state_action_values = tensor_rewards + (gamma * next_state_values )


    return criterion(predicted_state_action_values,target_state_action_values)
```