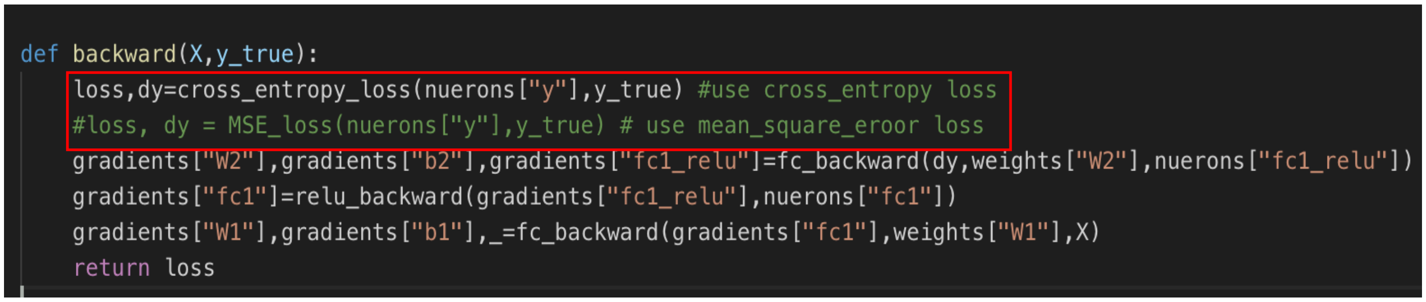
Hyper-paramters:

Num of epochs = 10
Batch size =128
Learning rate = 0.01
Model size = [784. 100, 10]

Usage:

- Numpy implementation:

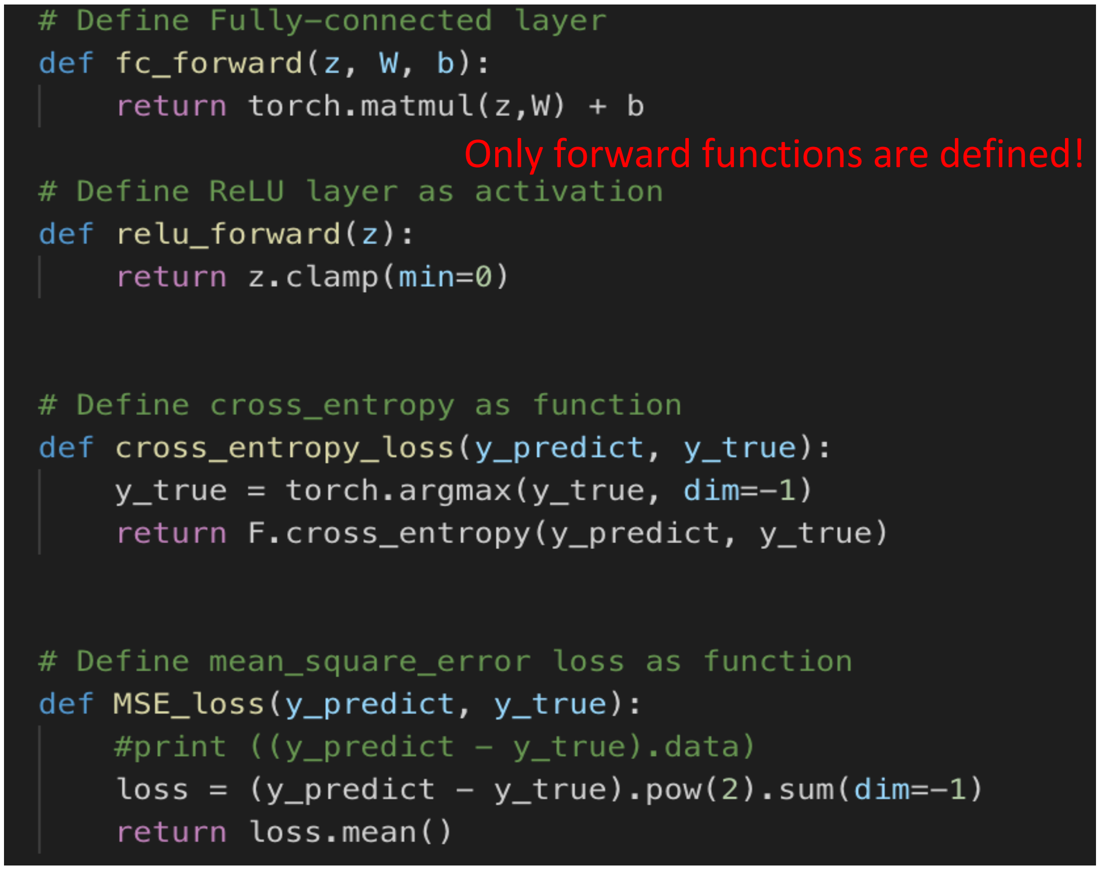
For downloading and saving mnist data set.
First run `$ python download_mnist.py`
Then go to numpy_dnn.py, select your loss function under *backward* function(line 77).
Run `$ python numpy_dnn.py`
Final test accuracy for crossentropy: 92.97%
Final test accuracy for mse: 92.66%



```
def backward(X,y_true):
    loss,dy=cross_entropy_loss(nuerons["y"],y_true) #use cross_entropy loss
    #loss, dy = MSE_loss(nuerons["y"],y_true) # use mean_square_eroor loss
    gradients["W2"],gradients["b2"],gradients["fc1_relu"]=fc_backward(dy,weights["W2"],nuerons["fc1_relu"])
    gradients["fc1"]=relu_backward(gradients["fc1_relu"],nuerons["fc1"])
    gradients["W1"],gradients["b1"],_=fc_backward(gradients["fc1"],weights["W1"],X)
    return loss
```

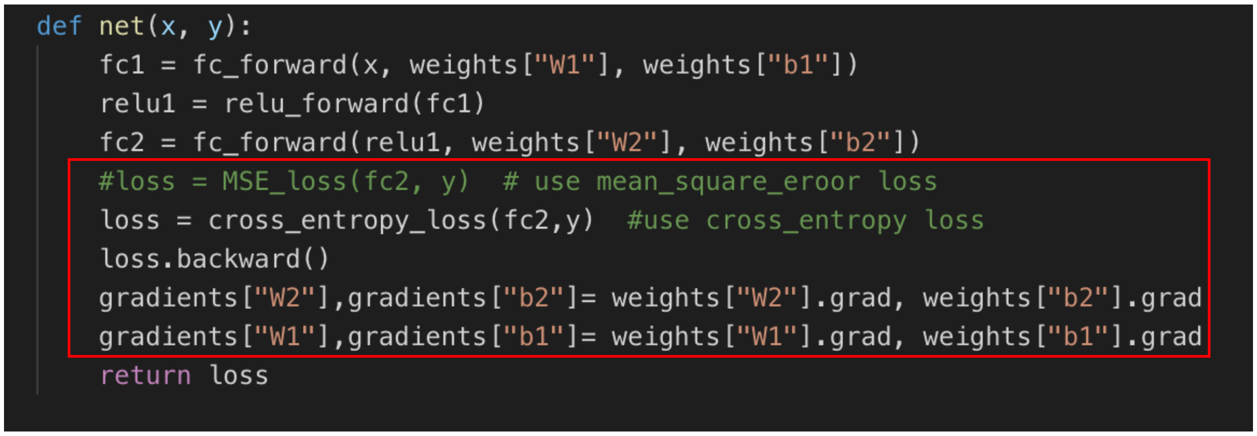
- Tensor implementation using autograd

We replace numpy-array as torch-tensor so that pytorch can help us compute gradient automatically. The forwarding functions compare to numpy implementation are basically same here, but we don't define any backward functions in tensor implementation. And let pytorch to compute the gradient by calling simple torch function (see line72-74). Select your own loss function in line 70 and 71.
Run `$ python autograd_dnn.py`
Final test accuracy for crossentropy: 91.83%
Final test accuracy for mse: 94.44%



```
# Define Fully-connected layer
def fc_forward(z, W, b):
    return torch.matmul(z,W) + b
```

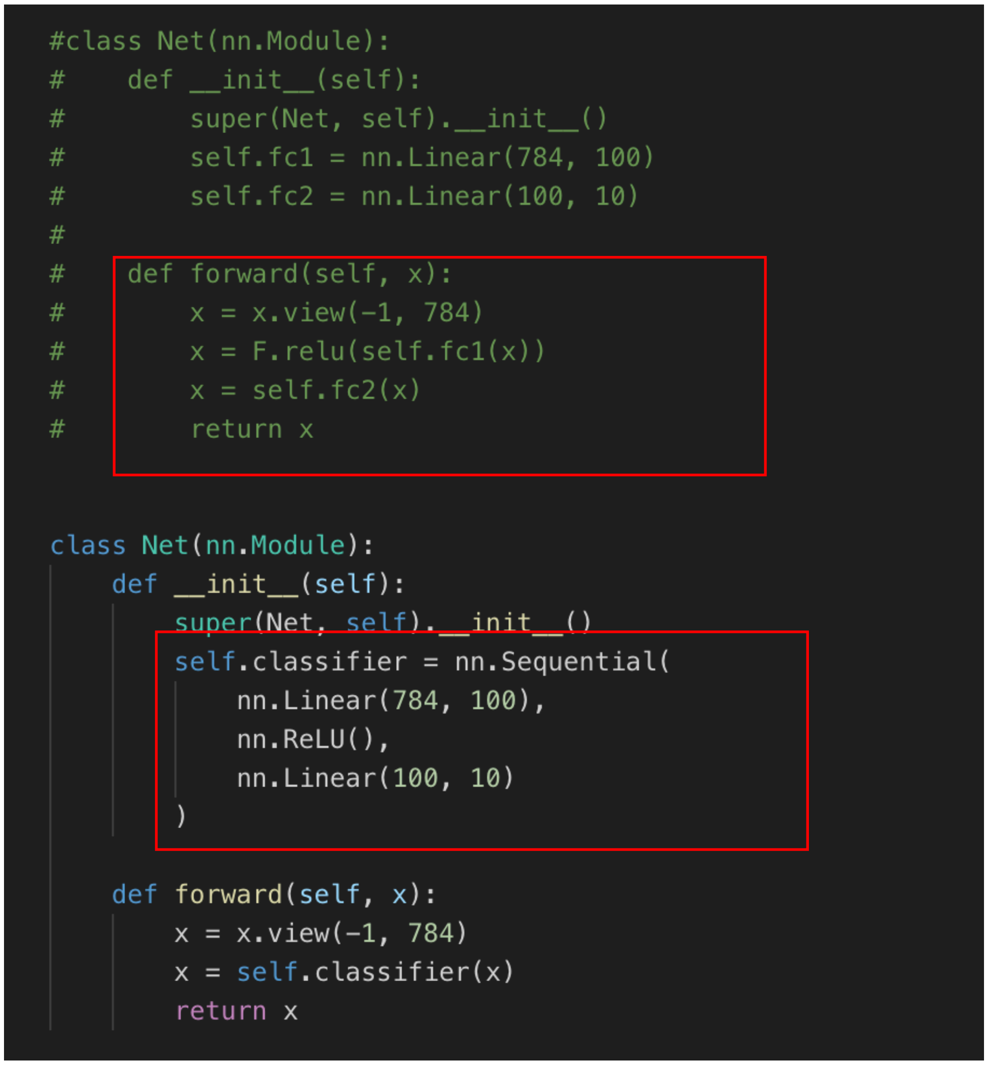
```
# Define ReLU layer as activation
def relu_forward(z):
    return z.clamp(min=0)


# Define cross_entropy as function
def cross_entropy_loss(y_predict, y_true):
    y_true = torch.argmax(y_true, dim=-1)
    return F.cross_entropy(y_predict, y_true)


# Define mean_square_error loss as function
def MSE_loss(y_predict, y_true):
    #print ((y_predict - y_true).data)
    loss = (y_predict - y_true).pow(2).sum(dim=-1)
    return loss.mean()
```



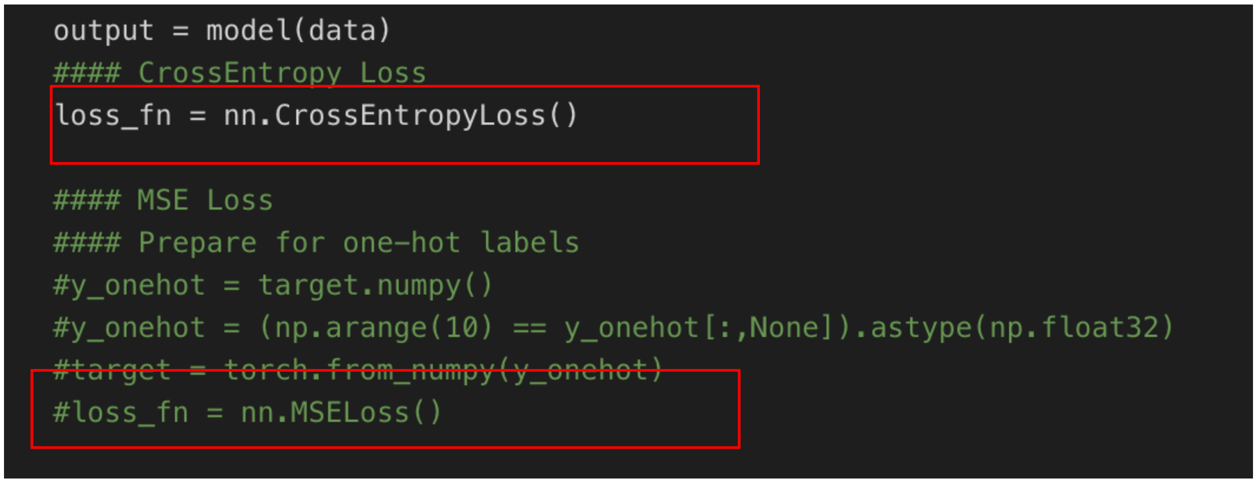
```
def net(x, y):
    fc1 = fc_forward(x, weights["W1"], weights["b1"])
    relu1 = relu_forward(fc1)
    fc2 = fc_forward(relu1, weights["W2"], weights["b2"])
    #loss = MSE_loss(fc2, y)  # use mean_square_eroor loss
    loss = cross_entropy_loss(fc2,y)  #use cross_entropy loss
    loss.backward()
    gradients["W2"],gradients["b2"]= weights["W2"].grad, weights["b2"].grad
    gradients["W1"],gradients["b1"]= weights["W1"].grad, weights["b1"].grad
    return loss
```

- For pytorch implementation:

You can select to use nn Sequential module or customized module from line 11 to line 37. And select loss function starts from line 47.
Run $ python pytorch_dnn.py



```
#class Net(nn.Module):
#    def __init__(self):
#        super(Net, self).__init__()
#        self.fc1 = nn.Linear(784, 100)
#        self.fc2 = nn.Linear(100, 10)
#
#    def forward(self, x):
#        x = x.view(-1, 784)
#        x = F.relu(self.fc1(x))
#        x = self.fc2(x)
#        return x


class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.classifier = nn.Sequential(
            nn.Linear(784, 100),
            nn.ReLU(),
            nn.Linear(100, 10)
        )

    def forward(self, x):
        x = x.view(-1, 784)
        x = self.classifier(x)
        return x
```



```
output = model(data)
#### CrossEntropy Loss
loss_fn = nn.CrossEntropyLoss()

#### MSE Loss
#### Prepare for one-hot labels
#y_onehot = target.numpy()
#y_onehot = (np.arange(10) == y_onehot[:,None]).astype(np.float32)
#target = torch.from_numpy(y_onehot)
#loss_fn = nn.MSELoss()
```

Final test accuracy for crossentropy: 94.59%
Final test accuracy for mse: 91.80%

Note:

1. The difference between execution time is due to the data loader in pytorch is slow. The computation time for forward and backward are similar in these two implementations.
2. In the file of numpy_dnn.py, we initialize the weights as following which is same as in pytorch. Otherwise, different initialization methods would lead to different training performance.

```python
# Initialize Weights and bias same as in pytorch
# model size 784-100-10
weights = {}
input_units = 784
std1 = 1. / math.sqrt(100)
std2 = 1. / math.sqrt(10)
weights["W1"] = np.random.uniform(-std1, std1,(input_units, 100)).astype(np.float64)
weights["b1"] = np.random.uniform(-std1, std1, 100).astype(np.float64)
weights["W2"] = np.random.uniform(-std2, std2,(100, 10)).astype(np.float64)
weights["b2"] = np.random.uniform(-std2, std2, 10).astype(np.float64)
```