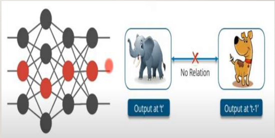
RECURRENT NEURAL NETWORK

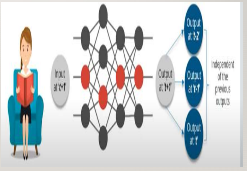
FEED FORWARD NETWORKS

- In Feed forward network flow of information takes place in the forward direction, as x is used to calculate some intermediate function in the hidden layer which in turn is used to calculate y.
- A trained feedforward network can be exposed to any random collection of photographs and the first photo it exposed will not affect how it classifies the second photo.



WHY NOT FEED FORWARD NETWORKS?

- When you read a book, you understand it based on your understanding of previous words.
- we cannot predict the next word in a sentence if we use feedforward nets.



WHERE WE USE RECURRENT NEURAL NETWORK?

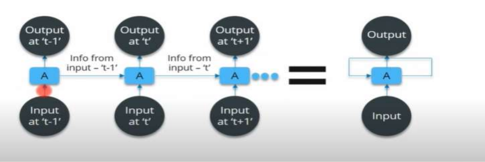
Recurrent Networks are a type of artificial neural network designed to recognize patterns in sequence in data such as:

- Text
- Handwriting
- Spoken words
- Numerical times series data emanating from sensors etc.

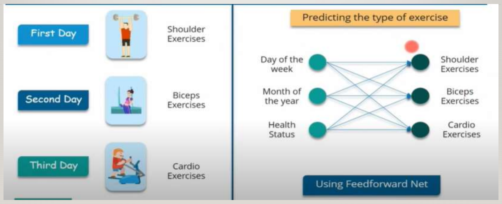
Suppose your gym trainer has made a schedule for you.
The exercises are repeated after every third day.

HOW TO OVER COME THIS CHALLENGE?

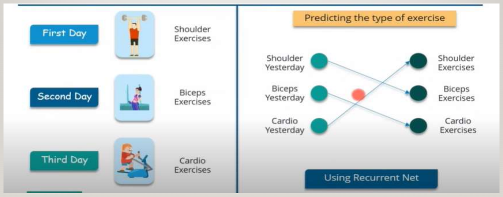
- Recurrent Neural Network



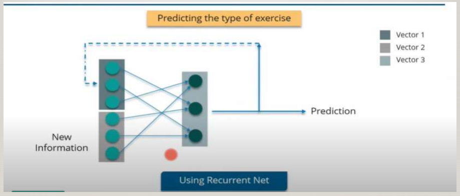
RECURRENT NEURAL NETWORK

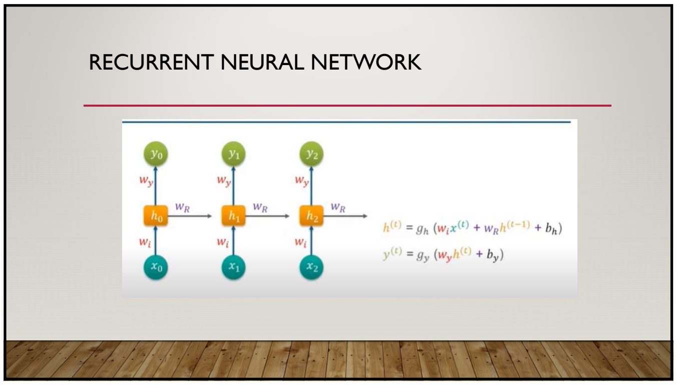


RECURRENT NEURAL NETWORK



RECURRENT NEURAL NETWORK

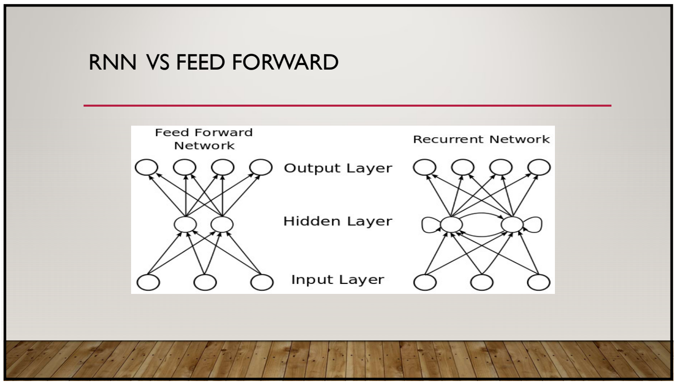




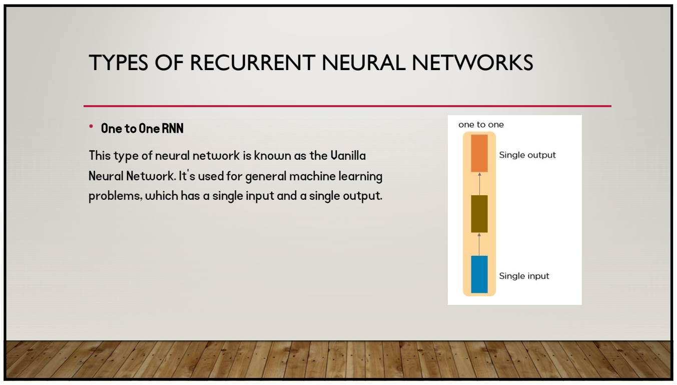
RECURRENT NEURAL NETWORK

$$h^{(t)} = g_h\left(w_i x^{(t)} + w_R h^{(t-1)} + b_h\right)$$

$$y^{(t)} = g_y\left(w_y h^{(t)} + b_y\right)$$



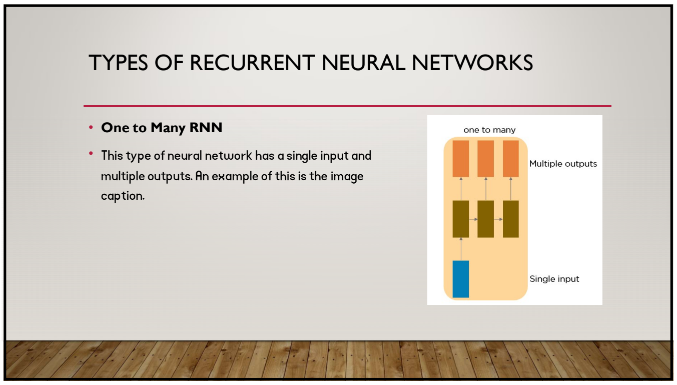
RNN VS FEED FORWARD



TYPES OF RECURRENT NEURAL NETWORKS

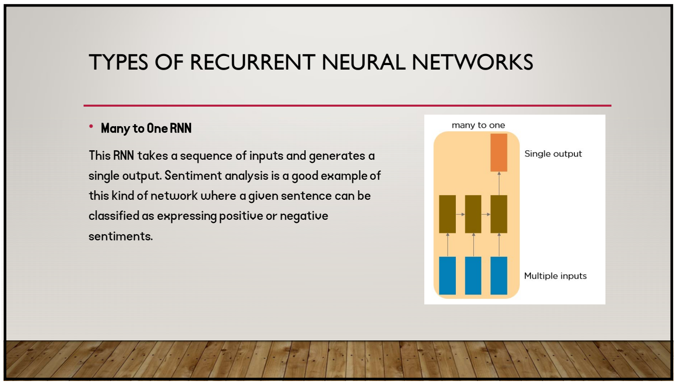
- **One to One RNN**

This type of neural network is known as the Vanilla Neural Network. It's used for general machine learning problems, which has a single input and a single output.



TYPES OF RECURRENT NEURAL NETWORKS

- **One to Many RNN**
- This type of neural network has a single input and multiple outputs. An example of this is the image caption.

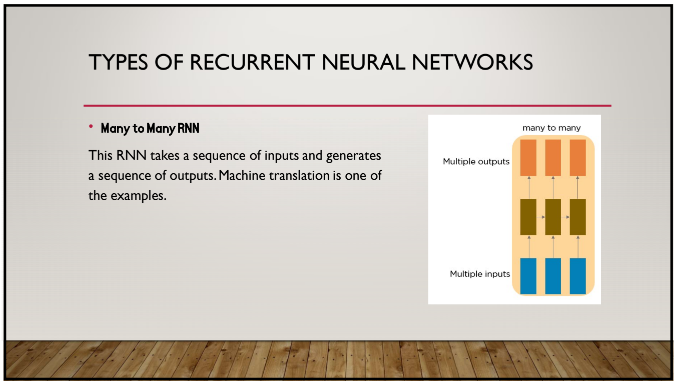


TYPES OF RECURRENT NEURAL NETWORKS

- **Many to One RNN**

This RNN takes a sequence of inputs and generates a single output. Sentiment analysis is a good example of this kind of network where a given sentence can be classified as expressing positive or negative sentiments.

many to one
Single output
Multiple inputs

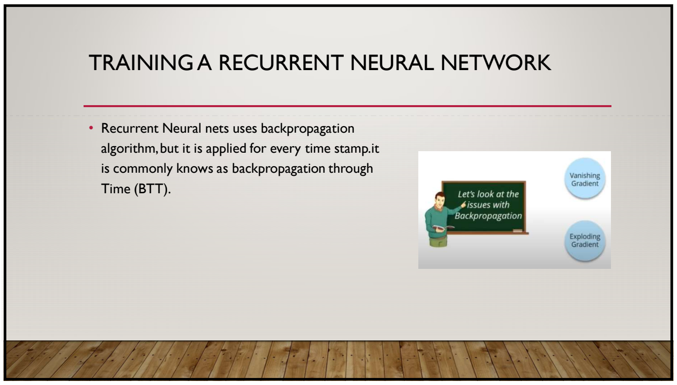


TYPES OF RECURRENT NEURAL NETWORKS

- **Many to Many RNN**

This RNN takes a sequence of inputs and generates a sequence of outputs. Machine translation is one of the examples.

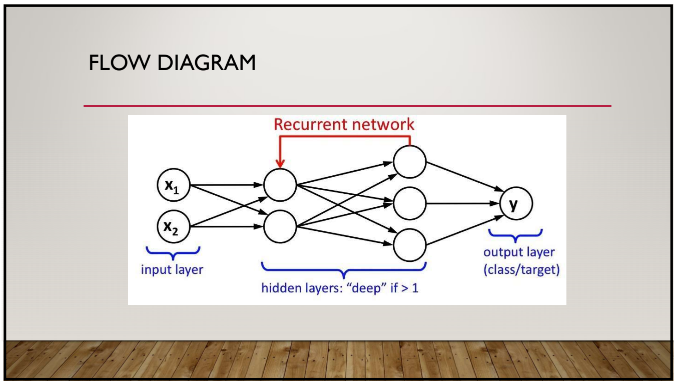
many to many
Multiple outputs
Multiple inputs



TRAINING A RECURRENT NEURAL NETWORK

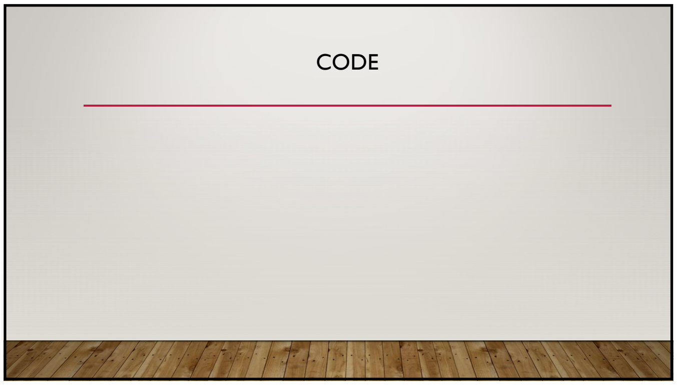
- Recurrent Neural nets uses backpropagation algorithm, but it is applied for every time stamp.it is commonly knows as backpropagation through Time (BTT).

Let's look at the issues with Backpropagation
Vanishing Gradient
Exploding Gradient

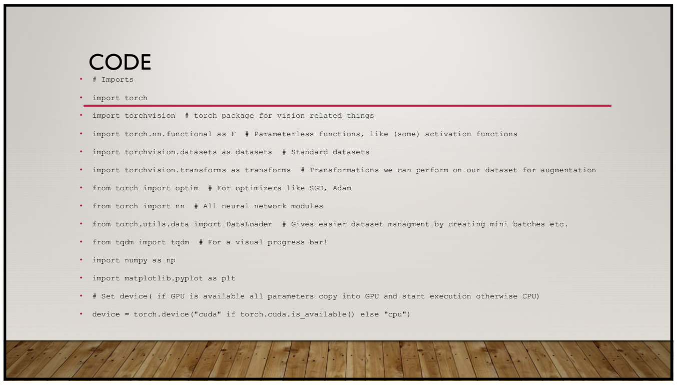


FLOW DIAGRAM

Recurrent network
x_1
x_2
y
input layer
hidden layers: "deep" if > 1
output layer (class/target)

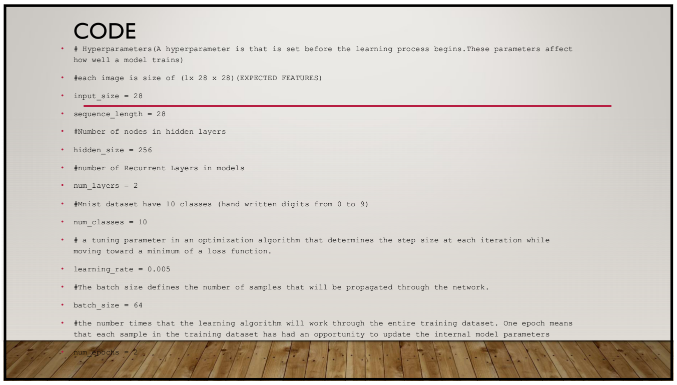


CODE



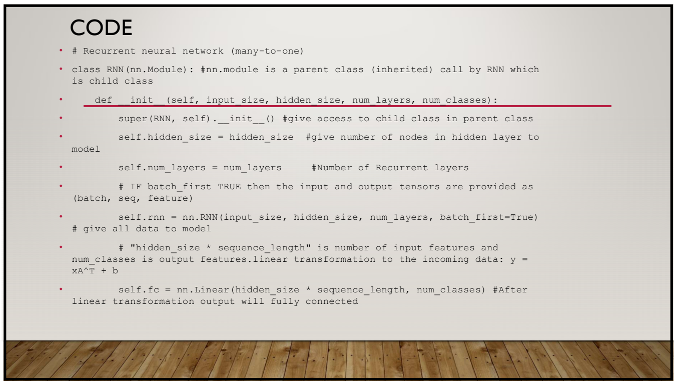
CODE

```
# Imports
import torch
import torchvision  # torch package for vision related things
import torch.nn.functional as F  # Parameterless functions, like (some) activation functions
import torchvision.datasets as datasets  # Standard datasets
import torchvision.transforms as transforms  # Transformations we can perform on our dataset for augmentation
from torch import optim  # For optimizers like SGD, Adam
from torch import nn  # All neural network modules
from torch.utils.data import DataLoader  # Gives easier dataset managment by creating mini batches etc.
from tqdm import tqdm  # For a visual progress bar!
import numpy as np
import matplotlib.pyplot as plt
# Set device( if GPU is available all parameters copy into GPU and start execution otherwise CPU)
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



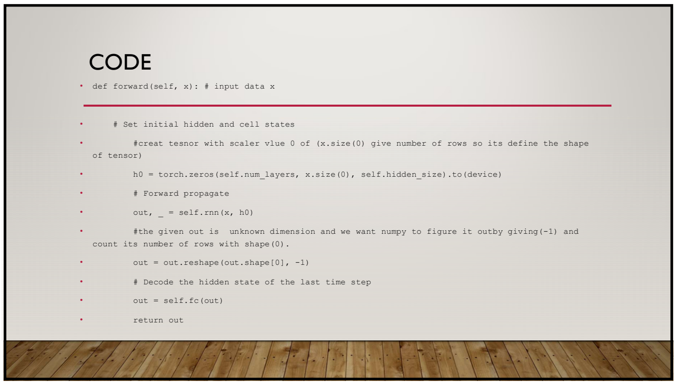
CODE

```
# Hyperparameters(A hyperparameter is that is set before the learning process begins.These parameters affect
how well a model trains)
#each image is size of (1x 28 x 28)(EXPECTED FEATURES)
input_size = 28
sequence_length = 28
#Number of nodes in hidden layers
hidden_size = 256
#number of Recurrent Layers in models
num_layers = 2
#Mnist dataset have 10 classes (hand written digits from 0 to 9)
num_classes = 10
# a tuning parameter in an optimization algorithm that determines the step size at each iteration while
moving toward a minimum of a loss function.
learning_rate = 0.005
#The batch size defines the number of samples that will be propagated through the network.
batch_size = 64
#the number times that the learning algorithm will work through the entire training dataset. One epoch means
that each sample in the training dataset has had an opportunity to update the internal model parameters
num_epochs = 2
```



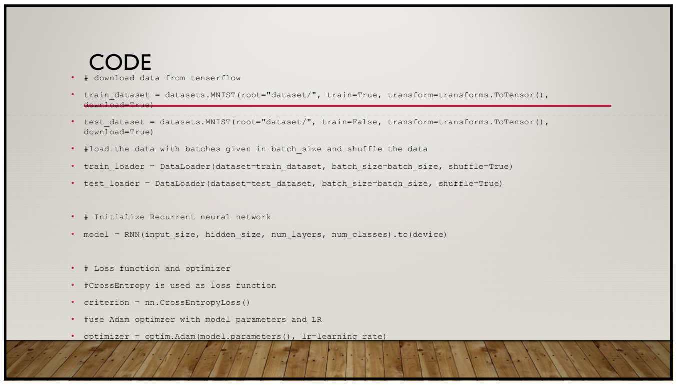
CODE

```
# Recurrent neural network (many-to-one)
class RNN(nn.Module): #nn.module is a parent class (inherited) call by RNN which
is child class
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(RNN, self).__init__() #give access to child class in parent class
        self.hidden_size = hidden_size  #give number of nodes in hidden layer to
model
        self.num_layers = num_layers     #Number of Recurrent layers
        # IF batch_first TRUE then the input and output tensors are provided as
(batch, seq, feature)
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
# give all data to model
        # "hidden_size * sequence_length" is number of input features and
num_classes is output features.linear transformation to the incoming data: y =
xA^T + b
        self.fc = nn.Linear(hidden_size * sequence_length, num_classes) #After
linear transformation output will fully connected
```



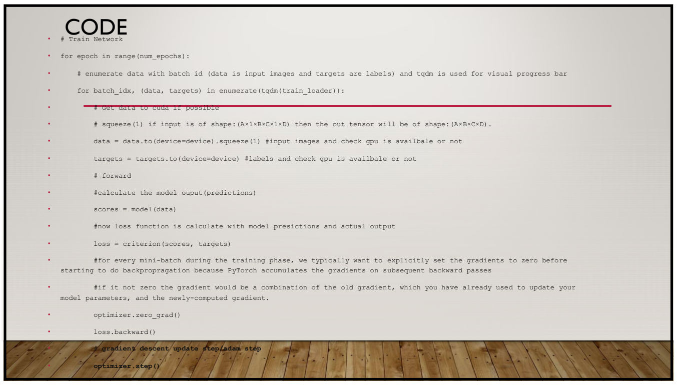
CODE

```
def forward(self, x): # input data x

    # Set initial hidden and cell states
        #creat tesnor with scaler vlue 0 of (x.size(0) give number of rows so its define the shape of tensor)
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(device)
        # Forward propagate
        out, _ = self.rnn(x, h0)
        #the given out is  unknown dimension and we want numpy to figure it outby giving(-1) and count its number of rows with shape(0).
        out = out.reshape(out.shape[0], -1)
        # Decode the hidden state of the last time step
        out = self.fc(out)
        return out
```



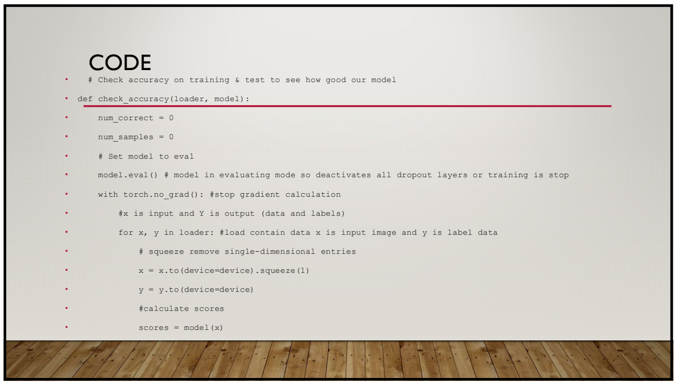
CODE

```
# download data from tenserflow
train_dataset = datasets.MNIST(root="dataset/", train=True, transform=transforms.ToTensor(), download=True)
test_dataset = datasets.MNIST(root="dataset/", train=False, transform=transforms.ToTensor(), download=True)
#load the data with batches given in batch_size and shuffle the data
train_loader = DataLoader(dataset=train_dataset, batch_size=batch_size, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=batch_size, shuffle=True)

# Initialize Recurrent neural network
model = RNN(input_size, hidden_size, num_layers, num_classes).to(device)

# Loss function and optimizer
#CrossEntropy is used as loss function
criterion = nn.CrossEntropyLoss()
#use Adam optimzer with model parameters and LR
optimizer = optim.Adam(model.parameters(), lr=learning_rate)
```



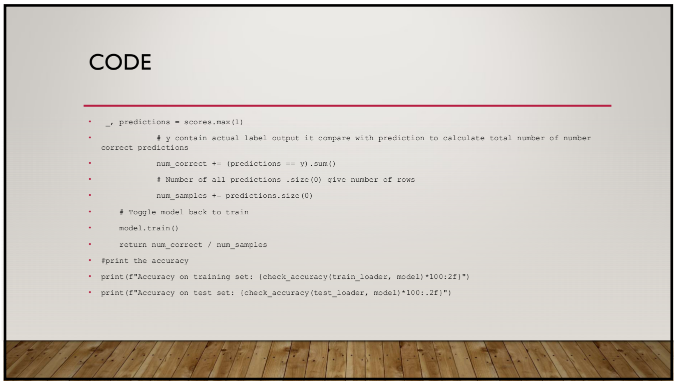
CODE

```
# Train Network
for epoch in range(num_epochs):
    # enumerate data with batch id (data is input images and targets are labels) and tqdm is used for visual progress bar
    for batch_idx, (data, targets) in enumerate(tqdm(train_loader)):
        # Get data to cuda if possible
        # squeeze(1) if input is of shape:(A×1×B×C×1×D) then the out tensor will be of shape:(A×B×C×D).
        data = data.to(device=device).squeeze(1) #input images and check gpu is availbale or not
        targets = targets.to(device=device) #labels and check gpu is availbale or not
        # forward
        #calculate the model ouput(predictions)
        scores = model(data)
        #now loss function is calculate with model presictions and actual output
        loss = criterion(scores, targets)
        #for every mini-batch during the training phase, we typically want to explicitly set the gradients to zero before starting to do backpropragation because PyTorch accumulates the gradients on subsequent backward passes
        #if it not zero the gradient would be a combination of the old gradient, which you have already used to update your model parameters, and the newly-computed gradient.
        optimizer.zero_grad()
        loss.backward()
        # gradient descent update step/adam step
        optimizer.step()
```



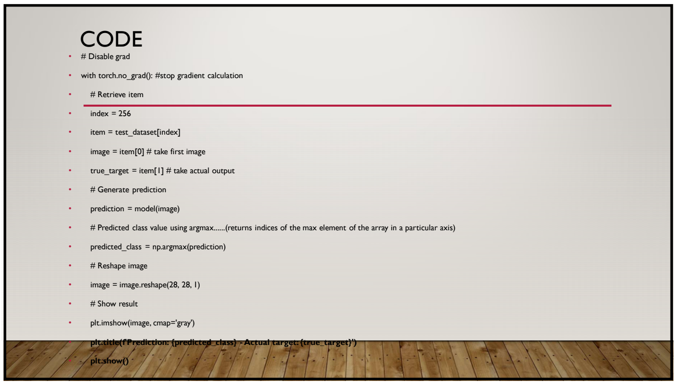
CODE

```
  # Check accuracy on training & test to see how good our model
def check_accuracy(loader, model):
    num_correct = 0
    num_samples = 0
    # Set model to eval
    model.eval() # model in evaluating mode so deactivates all dropout layers or training is stop
    with torch.no_grad(): #stop gradient calculation
        #x is input and Y is output (data and labels)
        for x, y in loader: #load contain data x is input image and y is label data
            # squeeze remove single-dimensional entries
            x = x.to(device=device).squeeze(1)
            y = y.to(device=device)
            #calculate scores
            scores = model(x)
```



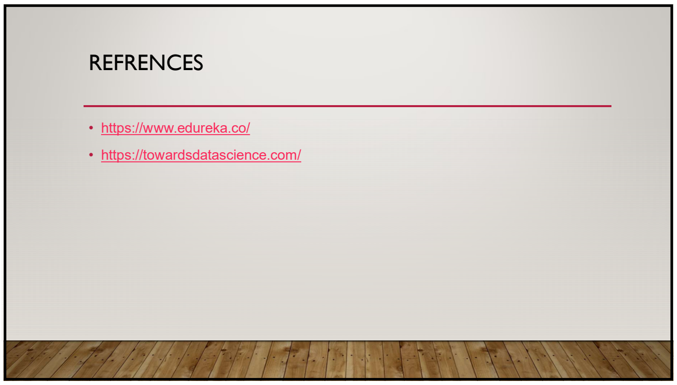
CODE

```
•  _, predictions = scores.max(1)
•            # y contain actual label output it compare with prediction to calculate total number of number
   correct predictions
•            num_correct += (predictions == y).sum()
•            # Number of all predictions .size(0) give number of rows
•            num_samples += predictions.size(0)
•     # Toggle model back to train
•     model.train()
•     return num_correct / num_samples
• #print the accuracy
• print(f"Accuracy on training set: {check_accuracy(train_loader, model)*100:2f}")
• print(f"Accuracy on test set: {check_accuracy(test_loader, model)*100:.2f}")
```



CODE

```
• # Disable grad
• with torch.no_grad(): #stop gradient calculation
•   # Retrieve item
•   index = 256
•   item = test_dataset[index]
•   image = item[0] # take first image
•   true_target = item[1] # take actual output
•   # Generate prediction
•   prediction = model(image)
•   # Predicted class value using argmax......(returns indices of the max element of the array in a particular axis)
•   predicted_class = np.argmax(prediction)
•   # Reshape image
•   image = image.reshape(28, 28, 1)
•   # Show result
•   plt.imshow(image, cmap='gray')
•   plt.title(f'Prediction: {predicted_class} - Actual target: {true_target}')
•   plt.show()
```



REFRENCES

- https://www.edureka.co/
- https://towardsdatascience.com/

THANKS