

## Pytorch Workflow Fundamentals

1. Get data ready
2. Build or pick a pre-trained model
3. Fit the model to the data and make a prediction
4. Evaluate the model
5. Improve through experimentations
6. Save and reload the model

```
import torch
from torch import nn # nn contains all of Pytorch's building blocks of neural networks
import matplotlib.pyplot as plt
```

```
torch.__version__
```

```
↳ '2.5.1+cu124'
```

### 1. Data(preparing and loading)

Machine learning is a game of two parts:

1. Turn your data, whatever it is, into numbers(a representation).
2. Pick or build a model to learn the representations as best as possible.

```
weight=0.7
bias=0.3

# create data
start=0
end=1
step=0.02
X=torch.arange(start,end,step).unsqueeze(dim=1)
y=weight*X+bias
X[:10],y[:10]
```

```
↳ (tensor([[0.0000],
           [0.0200],
           [0.0400],
           [0.0600],
           [0.0800],
           [0.1000],
           [0.1200],
           [0.1400],
           [0.1600],
           [0.1800]]),
```

```
tensor([[0.3000],
        [0.3140],
        [0.3280],
        [0.3420],
        [0.3560],
        [0.3700],
        [0.3840],
        [0.3980],
        [0.4120],
        [0.4260]]))
```

```
# create train/test split
train_split=int(0.8*len(X))
X_train,y_train=X[:train_split],y[:train_split]
X_test,y_test=X[train_split:],y[train_split:]

len(X_train),len(y_train),len(X_test),len(y_test)
```

```
↔ (40, 40, 10, 10)
```

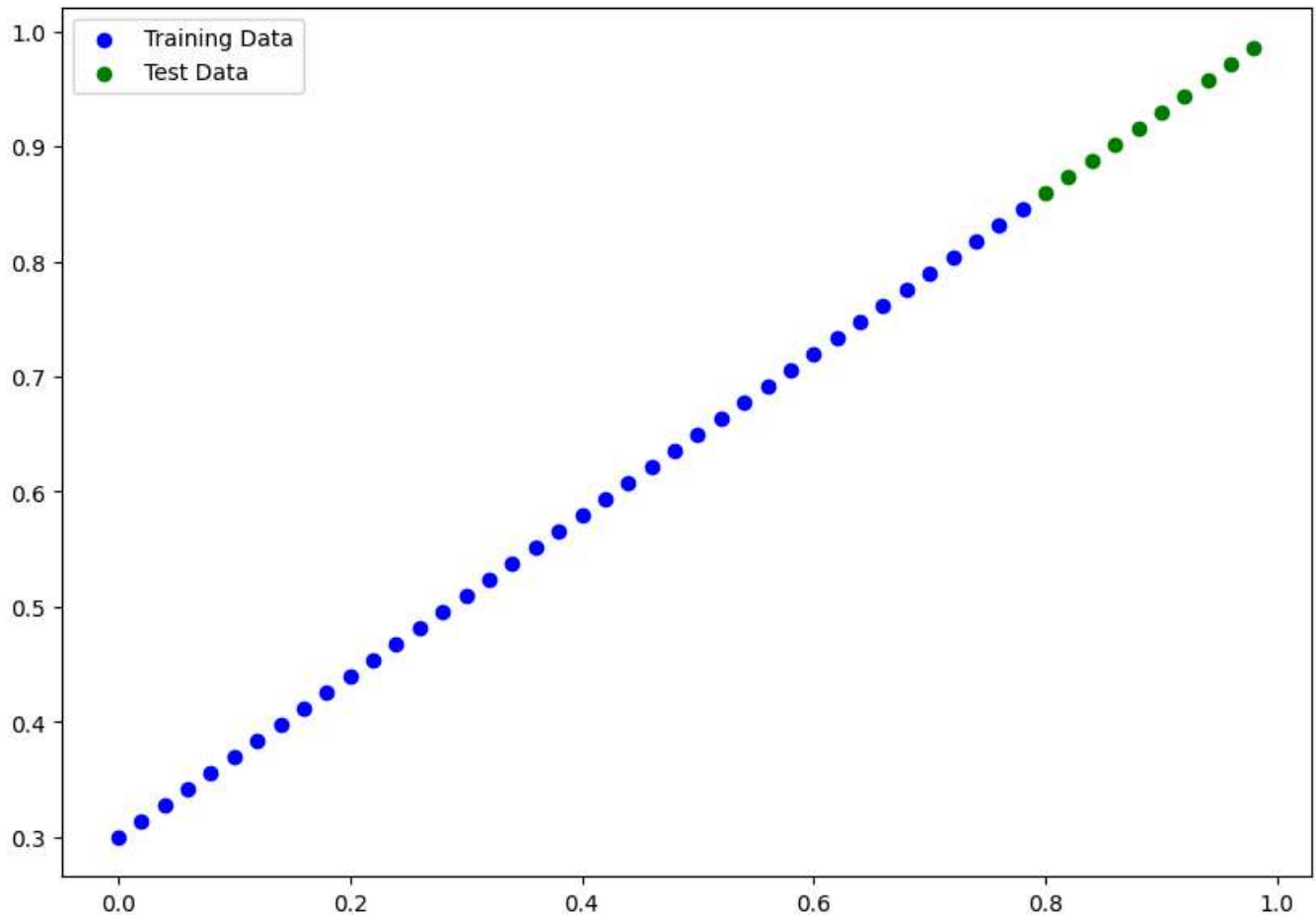
```
# let's plot the dataset
```

```
def plot_predictions(train_data=X_train,train_labels=y_train,test_data=X_test,test_labels=y_
    plt.figure(figsize=(10,7))
    plt.scatter(train_data,train_labels,c='b',label="Training Data")
    plt.scatter(test_data,test_labels,c='g',label="Test Data")
```

```
    if predictions is not None:
        plt.scatter(test_data,predictions,label="Predictions")
```

```
    plt.legend()
```

```
plot_predictions()
```



## 2. Build a Model

# create a Linear Regression model in pytorch

```
class LinearRegressionModel(nn.Module):
    def __init__(self):
        super().__init__()
        #initialize model parameters
        self.weights=nn.Parameter(torch.randn(1,dtype=torch.float),requires_grad=True)
        self.bias=nn.Parameter(torch.randn(1,dtype=torch.float),requires_grad=True)
    # forward() defines the computation in the model
    def forward(self,x:torch.Tensor)->torch.Tensor:
        return self.weights*x+self.bias
```

PyTorch module	What does it do?
<code>torch.nn</code>	Contains all of the building blocks for computational graphs (essentially a series of computations executed in a particular way).
<code>torch.nn.Parameter</code>	Stores tensors that can be used with <code>nn.Module</code> . If <code>requires_grad=True</code> gradients (used for updating model parameters via <b>gradient descent</b> ) are calculated automatically, this is often referred to as "autograd".
<code>torch.nn.Module</code>	The base class for all neural network modules, all the building blocks for neural networks are subclasses. If you're building a neural network in PyTorch, your models should subclass <code>nn.Module</code> . Requires a <code>forward()</code> method be implemented.
<code>torch.optim</code>	Contains various optimization algorithms (these tell the model parameters stored in <code>nn.Parameter</code> how to best change to improve gradient descent and in turn reduce the loss).
<code>def forward()</code>	All <code>nn.Module</code> subclasses require a <code>forward()</code> method, this defines the computation that will take place on the data passed to the particular <code>nn.Module</code> (e.g. the linear regression formula above).

```

1 # Create a linear regression model in PyTorch
2 class LinearRegressionModel(nn.Module):
3     def __init__(self):
4         super().__init__()
5
6         # Initialize model parameters
7         self.weights = nn.Parameter(torch.randn(1,
8         requires_grad=True,
9         dtype=torch.float
10    ))
11
12        self.bias = nn.Parameter(torch.randn(1,
13        requires_grad=True,
14        dtype=torch.float
15    ))
16
17    # forward() defines the computation in the model
18    def forward(self, x: torch.Tensor) -> torch.Tensor:
19        return self.weights * x + self.bias
20

```

**Subclass `nn.Module`**  
(this contains all the building blocks for neural networks)

Initialise **model parameters** to be used in various computations (these could be different layers from `torch.nn`, single parameters, hard-coded values or functions)

`requires_grad=True` means PyTorch will track the gradients of this specific parameter for use with `torch.autograd` and gradient descent (for many `torch.nn` modules, `requires_grad=True` is set by default)

Any subclass of `nn.Module` needs to override `forward()` (this defines the forward computation of the model)

```

# set manual seed since nn.Parameter are randomly initialized
torch.manual_seed(42)

```

```
model_0=LinearRegressionModel()  
list(model_0.parameters())
```

```
⇒ [Parameter containing:  
   tensor([0.3367], requires_grad=True),  
   Parameter containing:  
   tensor([0.1288], requires_grad=True)]
```

```
# we can also get the state of the model using .state_dict()  
model_0.state_dict()
```

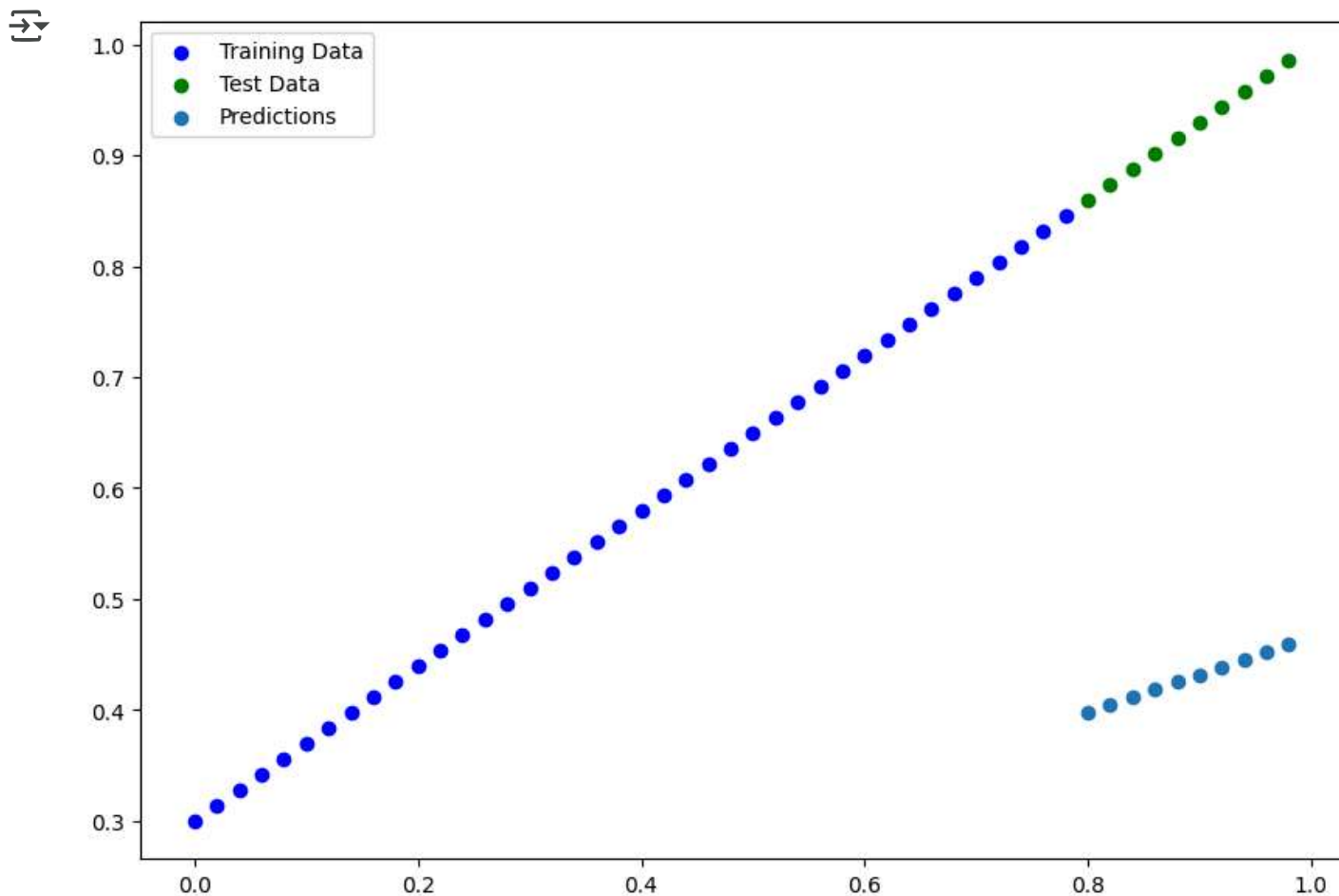
```
⇒ OrderedDict([('weights', tensor([0.3367])), ('bias', tensor([0.1288]))])
```

```
# Make predictions with model  
with torch.inference_mode():  
    y_preds=model_0(X_test)
```

```
# check the predictions  
print(f"Number of testing samples: {len(X_test)}")  
print(f"Number of predictions made: {len(y_preds)}")  
print(f"Predicted value:\n {y_preds}")
```

```
⇒ Number of testing samples: 10  
   Number of predictions made: 10  
   Predicted value:  
   tensor([[0.3982],  
           [0.4049],  
           [0.4116],  
           [0.4184],  
           [0.4251],  
           [0.4318],  
           [0.4386],  
           [0.4453],  
           [0.4520],  
           [0.4588]])
```

```
plot_predictions(predictions=y_preds)
```



y\_test-y\_preds

```
tensor([[0.4618],  
        [0.4691],  
        [0.4764],  
        [0.4836],  
        [0.4909],  
        [0.4982],  
        [0.5054],  
        [0.5127],  
        [0.5200],  
        [0.5272]])
```

### 3. Train the model

#create the loss function

```
loss_fun=nn.L1Loss()
```

```
optimizer=torch.optim.SGD(params=model_0.parameters(),lr=0.01)
```

# PyTorch training loop

```

1 # Pass the data through the model for a number of epochs (e.g. 100)
2 for epoch in range(epochs):
3
4     # Put model in training mode (this is the default state of a model)
5     model.train()
6
7     # 1. Forward pass on train data using the forward() method inside
8     y_pred = model(X_train)
9
10    # 2. Calculate the loss (how different are the model's predictions to the true values)
11    loss = loss_fn(y_pred, y_true)
12
13    # 3. Zero the gradients of the optimizer (they accumulate by default)
14    optimizer.zero_grad()
15
16    # 4. Perform backpropagation on the loss
17    loss.backward()
18
19    # 5. Progress/step the optimizer (gradient descent)
20    optimizer.step()

```

Note: all of this can be turned into a function

Pass the data through the model for a number of **epochs** (e.g. 100 for 100 passes of the data)

Pass the data through the model, this will perform the **forward()** method located within the model object

Calculate the **loss value** (how wrong the model's predictions are)

Zero the **optimizer gradients** (they accumulate every epoch, zero them to start fresh each forward pass)

Perform **backpropagation** on the loss function (compute the gradient of every parameter with `requires_grad=True`)

Step the **optimizer** to update the model's parameters with respect to the gradients calculated by `loss.backward()`

# PyTorch testing loop

```

1 # Setup empty lists to keep track of model progress
2 epoch_count = []
3 train_loss_values = []
4 test_loss_values = []
5
6 # Pass the data through the model for a number of epochs (e.g. 100) epochs:
7 for epoch in range(epochs):
8
9     ### Training loop code here ###
10
11     ### Testing starts ###
12
13     # Put the model in evaluation mode
14     model.eval()
15
16     # Turn on inference mode context manager
17     with torch.inference_mode():
18         # 1. Forward pass on test data
19         test_pred = model(X_test)
20
21         # 2. Calculate loss on test data
22         test_loss = loss_fn(test_pred, y_test)
23
24     # Print out what's happening every 10 epochs
25     if epoch % 10 == 0:
26         epoch_count.append(epoch)
27         train_loss_values.append(loss)
28         test_loss_values.append(test_loss)
29         print(f"Epoch: {epoch} | MAE Train Loss: {loss} | MAE Test Loss: {test_loss}")

```

Note: all of this can be turned into a function

Create empty lists for storing useful values (helpful for tracking model progress)

Tell the model we want to **evaluate** rather than train (this turns off functionality used for training but not evaluation)

Turn on **`torch.inference_mode()`** context manager to disable functionality such as gradient tracking for inference (gradient tracking not needed for inference)

Pass the test data through the model (this will call the model's implemented **`forward()`** method)

Calculate the **test loss value** (how wrong the model's predictions are on the test dataset, lower is better)

Display **information outputs** for how the model is doing during training/testing every ~10 epochs (note: what gets printed out here can be adjusted for specific problems)

```
torch.manual_seed(42)
```

```
# set the number of epochs(how many times the model will pass over the training data)
epochs=100
```



```
# create empty loss lists to track the values

train_loss_values=[]
test_loss_values=[]
epoch_count=[]

for epoch in range(epochs):
    ### Training

    # put the model in training mode(this is the default state of the model)
    model_0.train()
    # 1. Forward pass on the train data using forward() method inside
    y_pred=model_0(X_train)
    # 2. calculate the loss(how different are our models predictions to the ground truth)
    loss=loss_fun(y_pred,y_train)
    # 3. Zero grad of the optimizer
    optimizer.zero_grad()
    # 4. loss backwards
    loss.backward()
    # 5. Progress the optimizer
    optimizer.step()
```

### ### Testing

```
# Put the model in evaluation mode
model_0.eval()
with torch.inference_mode():
    # 1. Forward pass on the test data
    test_pred=model_0(X_test)
    # 2. Calculate loss on the test data
    test_loss=loss_fun(test_pred,y_test.type(torch.float))

# Print out what is happening
```

```
if epoch%10==0:
    epoch_count.append(epoch)
    train_loss_values.append(loss.detach().numpy())
    test_loss_values.append(test_loss.detach().numpy())
    print(f"Epoch:{epoch} | MAE Train loss: {loss} | MAE Test loss: {test_loss}")
```

```
➡ Epoch:0 | MAE Train loss: 0.31288138031959534 | MAE Test loss: 0.48106518387794495
Epoch:10 | MAE Train loss: 0.1976713240146637 | MAE Test loss: 0.3463551998138428
Epoch:20 | MAE Train loss: 0.08908725529909134 | MAE Test loss: 0.21729660034179688
Epoch:30 | MAE Train loss: 0.053148526698350906 | MAE Test loss: 0.14464017748832703
Epoch:40 | MAE Train loss: 0.04543796554207802 | MAE Test loss: 0.11360953003168106
Epoch:50 | MAE Train loss: 0.04167863354086876 | MAE Test loss: 0.09919948130846024
Epoch:60 | MAE Train loss: 0.03818932920694351 | MAE Test loss: 0.08886633068323135
```

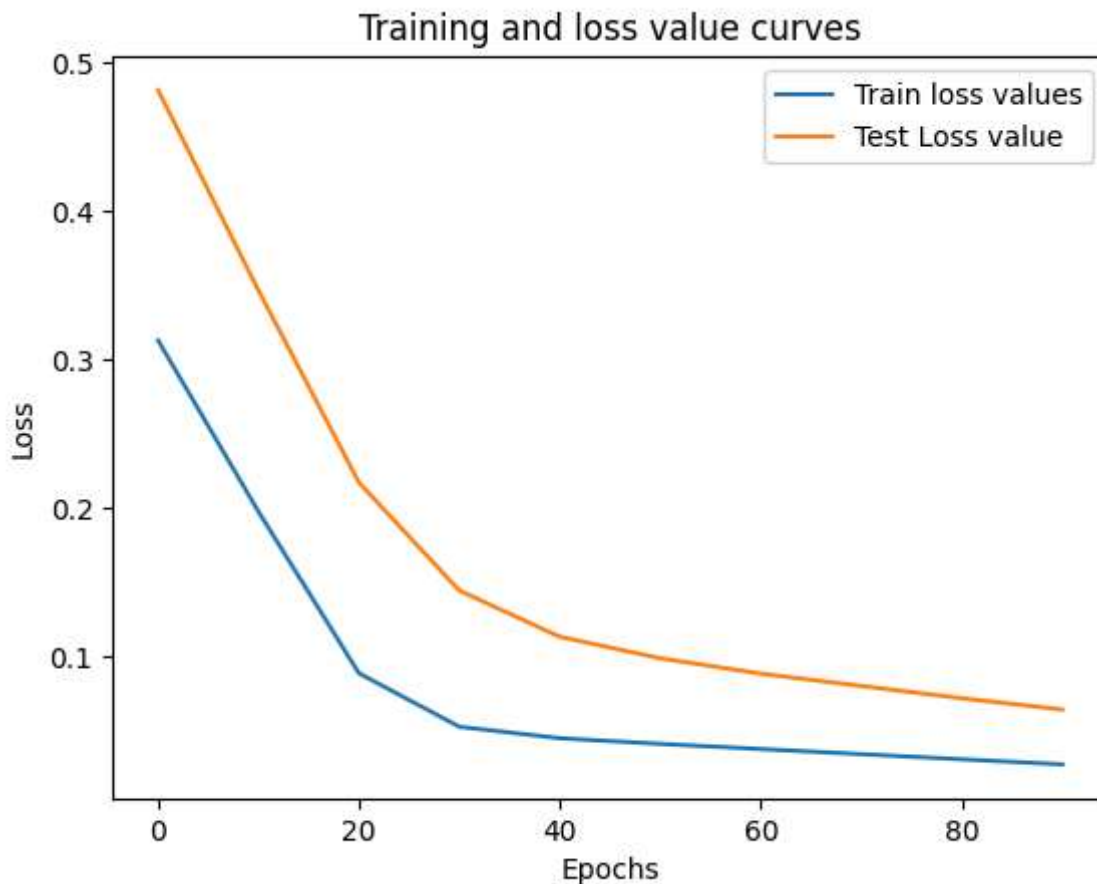


```
Epoch:70 | MAE Train loss: 0.03476089984178543 | MAE Test loss: 0.0805937647819519
Epoch:80 | MAE Train loss: 0.03132382780313492 | MAE Test loss: 0.07232122868299484
Epoch:90 | MAE Train loss: 0.02788739837706089 | MAE Test loss: 0.06473556160926819
```

```
# plot the loss curves
```

```
plt.plot(epoch_count,train_loss_values,label="Train loss values")
plt.plot(epoch_count,test_loss_values,label="Test Loss value")
plt.title("Training and loss value curves")
plt.ylabel("Loss")
plt.xlabel("Epochs")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7906b8dd5ad0>
```



```
# find the model parameters
```


```
print(f"The model learned the following values for weights and bias: {model_0.state_dict()}")
print(f"Original values for weights and bias: {weight} and {bias}")
```

```
<matplotlib.figure.Figure at 0x7906b8dd5ad0> The model learned the following values for weights and bias: OrderedDict([('weights', te
Original values for weights and bias: 0.7 and 0.3
```

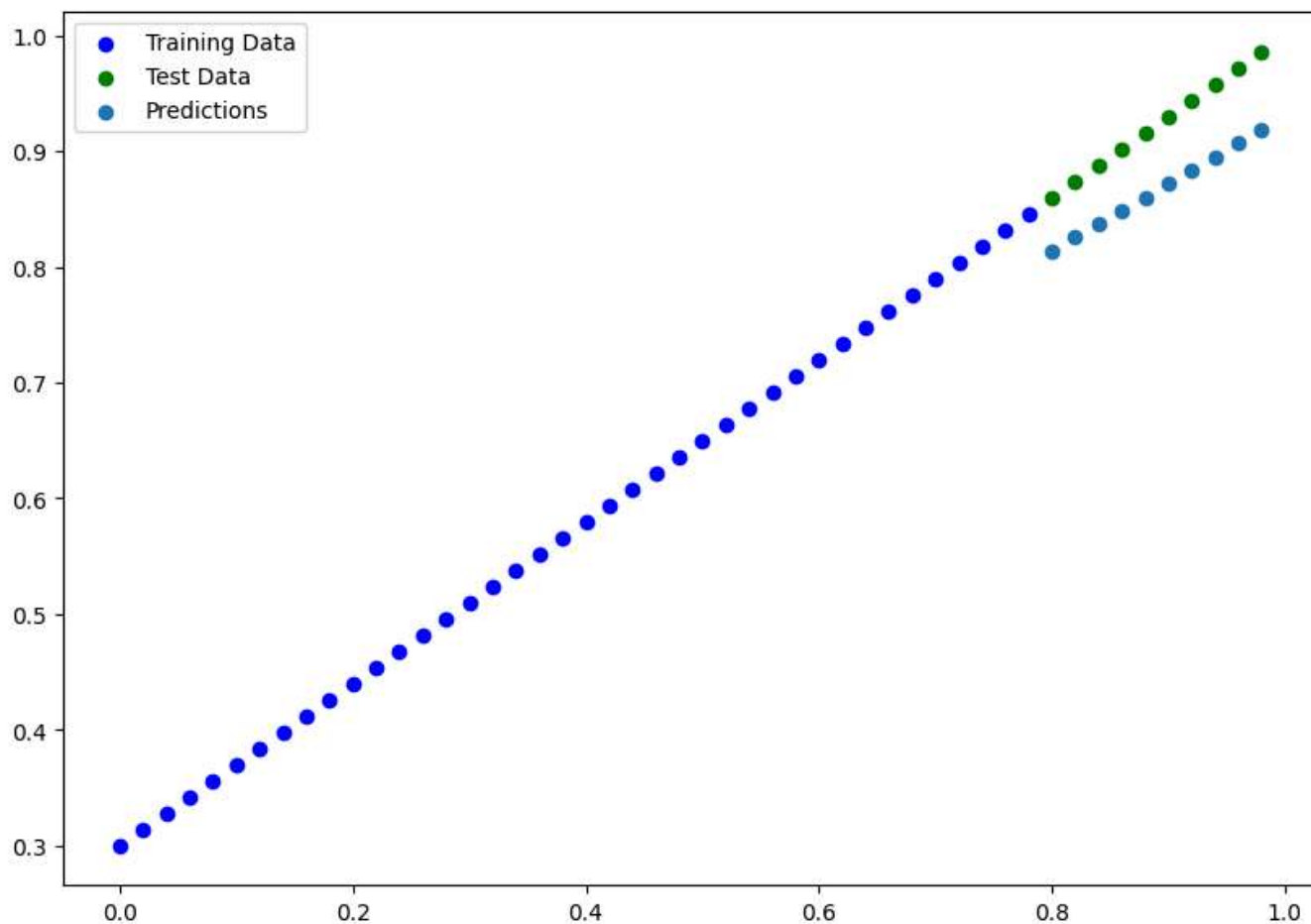
#### 4. Making predictions with a pre-trained Pytorch model(inference)

```
# 1. Set the model in evaluation mode
model_0.eval()

# 2. Setup the inference mode context manager
with torch.inference_mode():
    # 3. Make sure the calculations are done with the model and data on the same device
    # in our case, we haven't setup device-agnostic code yet so our data and model are
    # on the CPU by default.
    # model_0.to(device)
    # X_test = X_test.to(device)
    y_preds = model_0(X_test)
y_preds
```

 tensor([[0.8141],  
[0.8256],  
[0.8372],  
[0.8488],  
[0.8603],  
[0.8719],  
[0.8835],  
[0.8950],  
[0.9066],  
[0.9182]])

```
plot_predictions(predictions=y_preds)
```



y\_test-y\_preds



```
tensor([[0.0459],
        [0.0484],
        [0.0508],
        [0.0532],
        [0.0557],
        [0.0581],
        [0.0605],
        [0.0630],
        [0.0654],
        [0.0678]])
```

## 5. Loading a model and saving a Pytorch model

```
from pathlib import Path
```

```
#1. Create models directory
```

```
Model_Path=Path("models")
Model_Path.mkdir(parents=True,exist_ok=True)
```

```
# 2. Create model save path
```

```
model_name="01_pytorch_workflow_model_0.pth"
model_save_path=Model_Path/model_name
```

```
# 3. Save the model state dict
```

```
print(f"Saving model to : {model_save_path}")
torch.save(obj=model_0.state_dict(),f=model_save_path)
```

```
➡ Saving model to : models/01_pytorch_workflow_model_0.pth
```

```
# check the saved model file path
```

```
!ls -l models/01_pytorch_workflow_model_0.pth
```

```
➡ -rw-r--r-- 1 root root 1680 Feb 28 05:01 models/01_pytorch_workflow_model_0.pth
```

```
# loading a saved pytorch model's state_dict()
```

```
loaded_model_0=LinearRegressionModel()
```

```
loaded_model_0.load_state_dict(torch.load(f=model_save_path))
```

```
➡ <ipython-input-58-0ccbbce14452>:5: FutureWarning: You are using `torch.load` with `weights_only=False` (the default) which is unsafe. To silence this warning and avoid this warning in the future, please use `torch.load` with `weights_only=True` if you are sure you are loading only weights. To suppress this warning, you can also pass `weights_only=False` to the `torch.load` call.
loaded_model_0.load_state_dict(torch.load(f=model_save_path))
```