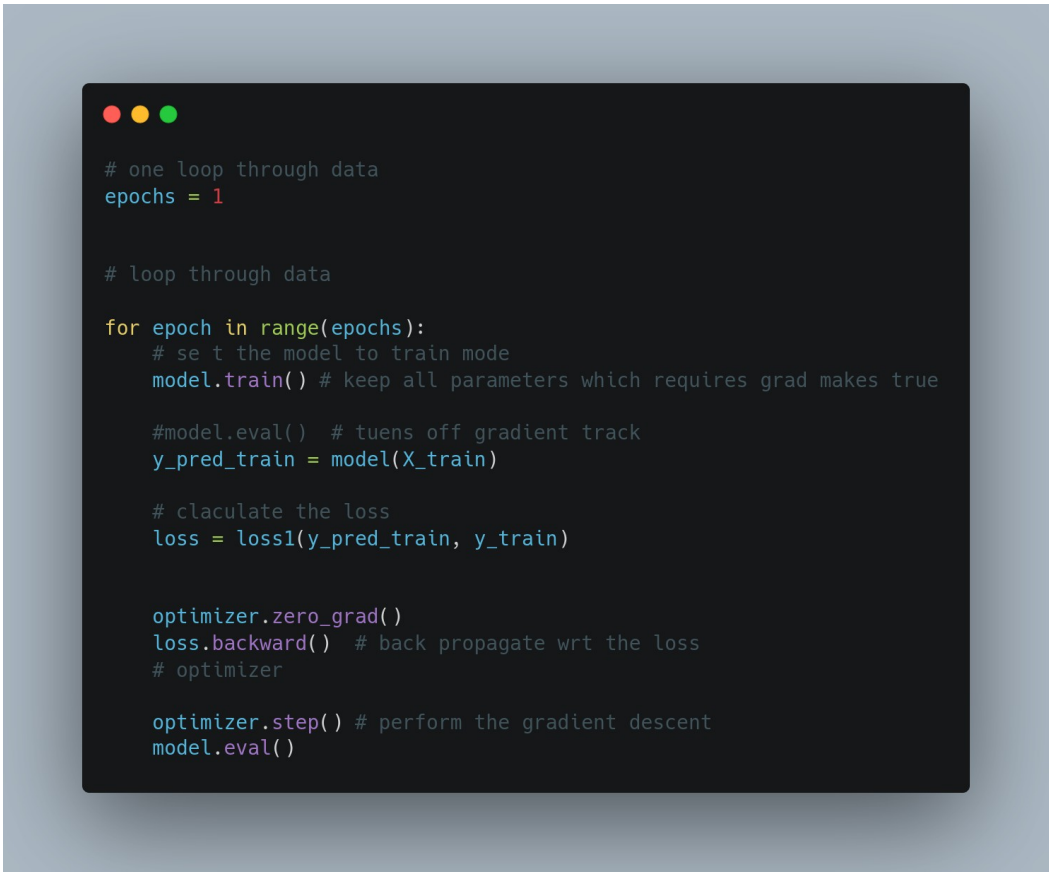
PYTORCH UNCODE

UNDERSTANDING CODE DEEPER --



```
# one loop through data
epochs = 1


# loop through data

for epoch in range(epochs):
    # se t the model to train mode
    model.train() # keep all parameters which requires grad makes true

    #model.eval()  # tuens off gradient track
    y_pred_train = model(X_train)

    # claculate the loss
    loss = loss1(y_pred_train, y_train)


    optimizer.zero_grad()
    loss.backward()  # back propagate wrt the loss
    # optimizer

    optimizer.step() # perform the gradient descent
    model.eval()
```

	Epoch	Batch
Definition	One complete pass through the entire training dataset.	A subset of the training data used to update the model once.
Purpose	Allows the model to learn from all training samples.	Efficiently computes gradients using only a small portion of the data.
Training Efficiency	Larger number of epochs may require more time to train the model.	Smaller batch sizes may lead to faster iterations.
Parameter Updates	Model parameters are updated after processing the entire dataset.	Model parameters are updated after processing a batch of data.
Generalization	Too few epochs may result in underfitting; too many may result in overfitting.	Smaller batch sizes can lead to more noisy updates, which may affect generalization.
Memory Usage	Memory usage may be high if the dataset is large, as it processes all samples at once.	Memory usage is lower compared to processing the entire dataset at once.
Practical	Choosing an appropriate number of epochs is	Batch size should be chosen based on

	Epoch	Batch
Consideration	important for finding a balance between underfitting and overfitting.	available memory and computational resources.

During training, the dataset is divided into smaller subsets called batches. The model processes each batch in succession, updating its parameters (weights and biases) based on the error it makes while predicting the targets for that batch. After processing all the batches in the dataset, one epoch is completed.

For example, if you have 10,000 training samples and use a batch size of 100, each epoch would involve 100 iterations (batches) since 100 samples are processed at a time. The model's parameters are updated after each batch, and after 100 batches, the model has seen all 10,000 samples once, completing one epoch.

Training a model for multiple epochs allows it to iteratively improve its performance by refining its parameters based on the accumulated errors over multiple passes through the data. The number of epochs is typically a hyperparameter that needs to be chosen before training, and it depends on factors such as the complexity of the model, the size of the dataset, and the learning rate.

It's important to find the right balance when choosing the number of epochs. Too few epochs may result in underfitting, where the model hasn't learned enough from the data, while too many epochs may lead to overfitting, where the model memorizes the training data but fails to generalize well on unseen data. Properly tuning the number of epochs is essential to achieving good model performance on new, unseen data.

Model.train() --

In PyTorch, `model.train()` is a method used to set the model in training mode. It is a common practice to call `model.train()` before starting the training loop, and it is used in conjunction with `model.eval()` for switching between training and evaluation (testing) modes.

The internal workings of `model.train()` involve enabling certain layers and operations in the model that behave differently during training and evaluation (inference). The key components affected by `model.train()` are:

1. Dropout Layers: Dropout is a regularization technique used to prevent overfitting. During training, dropout randomly "drops out" some neurons by setting their activations to zero with a given probability. This helps the model become more robust and avoid relying too much on specific neurons. When the model is in training mode (`model.train()` is called), dropout layers are active.

2. Batch Normalization Layers: Batch normalization is used to normalize the activations of the previous layer by adjusting the mean and variance of the inputs. This helps stabilize and speed up training. During training, batch normalization uses the mini-batch statistics to compute normalization parameters. When the model is in training mode (`model.train()` is called), batch normalization layers are active and update their parameters using the mini-batch statistics.
3. Other Layers with Different Behaviors: Certain layers or operations might have different behaviors during training and testing. For example, some custom layers might implement special dropout-like behavior or specific data augmentation techniques. `model.train()` ensures that these layers behave as intended during training.

	`model.train()`	`model.eval()`
Purpose	Set the model in training mode.	Set the model in evaluation (testing) mode.
Behaviors Affected	- Dropout layers: Enabled and applied during training, randomly drops out neurons. - Batch Normalization layers: Enabled and uses mini-batch statistics for normalization. - Other layers: Can enable training-specific behaviors.	- Dropout layers: Disabled during evaluation to use all neurons for predictions. - Batch Normalization layers: Uses learned statistics from training for normalization. - Other layers: Can enable evaluation-specific behaviors.
Gradients Tracking	Enabled, allows computation of gradients during backpropagation.	Disabled, no gradients are computed during evaluation for memory efficiency.
Usage during Phases	Used during the training phase.	Used during the evaluation/testing phase.

However, the `model.train()` method itself does not load or modify the training data directly. The actual training data is typically passed to the model during the training loop, where forward and backward passes, gradient computations, and weight updates occur.

So, in short, calling `model.train()` doesn't affect the training data but instead activates specific training-related behaviors within the model. The actual training data is used during the training loop for learning and updating the model's parameters (weights and biases) through optimization techniques like gradient descent.

`optimizer.zero_grad()`: This function resets all gradients of the model's parameters to zero before computing gradients for the current batch, ensuring fresh gradients for each iteration.

`loss.backward()`: This line performs backpropagation to compute gradients of the model's parameters with respect to the loss, propagating errors backward through the network.

`optimizer.step()`: This statement applies the optimization algorithm (e.g., stochastic gradient descent) to update the model's parameters using the computed gradients, effectively adjusting the weights and biases.

Forward Pass: During the forward pass, the input data is fed through the neural network, and the model makes predictions. The difference between the predicted output and the actual target (ground truth) is computed using the chosen loss function. This difference is the loss.

Backward Pass (Backpropagation): After computing the loss, you call `loss.backward()`, and this is where backpropagation happens. During backpropagation, PyTorch automatically computes the gradients of the loss with respect to each parameter (weights and biases) in the model. It uses the chain rule of calculus to efficiently propagate the gradients from the loss through all the layers of the network.

Gradient Accumulation: The gradients are accumulated over the current batch of data. This means that when you call `loss.backward()` for each batch, the gradients are computed and stored, and they are not applied for parameter updates immediately. The gradients from each batch are added together during the optimization process (optimizer step) to get the overall gradient.

optimizer.step(): After accumulating the gradients from all batches in the epoch, you call `optimizer.step()`. This step is responsible for applying the optimization algorithm (e.g., stochastic gradient descent, Adam) to update the model's parameters using the computed gradients. The optimizer adjusts the parameters in the direction that reduces the loss, effectively improving the model's performance.

Example Data Points:

```
(x=1, y_true=5)
(x=2, y_true=7)
(x=3, y_true=9)
(x=4, y_true=11)
(x=5, y_true=13)
```

Initialization:

```
w = 1.0    # Initial weight
b = 0.0    # Initial bias
learning_rate = 0.01
```

Now, let's perform 3 iterations of gradient descent:

Iteration 1:

1. Forward Pass:

```
For x=1:
y_pred = (1 * w) + b = (1 * 1.0) + 0.0 = 1.0

For x=2:
y_pred = (2 * w) + b = (2 * 1.0) + 0.0 = 2.0

For x=3:
y_pred = (3 * w) + b = (3 * 1.0) + 0.0 = 3.0

For x=4:
y_pred = (4 * w) + b = (4 * 1.0) + 0.0 = 4.0

For x=5:
y_pred = (5 * w) + b = (5 * 1.0) + 0.0 = 5.0
```

2. Loss Computation:

```
For x=1:
loss = (y_true - y_pred)**2 = (5 - 1)**2 = 16

For x=2:
loss = (y_true - y_pred)**2 = (7 - 2)**2 = 25

For x=3:
loss = (y_true - y_pred)**2 = (9 - 3)**2 = 36

For x=4:
loss = (y_true - y_pred)**2 = (11 - 4)**2 = 49

For x=5:
loss = (y_true - y_pred)**2 = (13 - 5)**2 = 64
```

3. Backward Pass (Gradient Computation):

```
For x=1:
∂loss/∂w = -2 * (y_true - y_pred) * x = -2 * (5 - 1) * 1 = -8
∂loss/∂b = -2 * (y_true - y_pred) = -2 * (5 - 1) = -8

For x=2:
∂loss/∂w = -2 * (y_true - y_pred) * x = -2 * (7 - 2) * 2 = -10
∂loss/∂b = -2 * (y_true - y_pred) = -2 * (7 - 2) = -10

For x=3:
∂loss/∂w = -2 * (y_true - y_pred) * x = -2 * (9 - 3) * 3 = -12
∂loss/∂b = -2 * (y_true - y_pred) = -2 * (9 - 3) = -12
```

```
For x=4:
∂loss/∂w = -2 * (y_true - y_pred) * x = -2 * (11 - 4) * 4 = -14
∂loss/∂b = -2 * (y_true - y_pred) = -2 * (11 - 4) = -14

For x=5:
∂loss/∂w = -2 * (y_true - y_pred) * x = -2 * (13 - 5) * 5 = -16
∂loss/∂b = -2 * (y_true - y_pred) = -2 * (13 - 5) = -16
```

4. Parameter Update (Optimization Step):

```
Updated weight w = w - learning_rate * (∂loss/∂w)
Updated bias b = b - learning_rate * (∂loss/∂b)
```

Let's assume `learning_rate = 0.01` for this example.

```
Updated weight w = 1.0 - 0.01 * (-8) = 1.08
Updated bias b = 0.0 - 0.01 * (-8) = 0.08
```

Iteration 2:

1. Forward Pass: Same as Iteration 1, but now with the updated weight and bias values.
2. Loss Computation: Same as Iteration 1.
3. Backward Pass (Gradient Computation): Same as Iteration 1, but now with the updated weight and bias values.
4. Parameter Update (Optimization Step):

```
Updated weight w = w - learning_rate * (∂loss/∂w)
Updated bias b = b - learning_rate * (∂loss/∂b)
```

```
Updated weight w = 1.08 - 0.01 * (-7.16) = 1.15
Updated bias b = 0.08 - 0.01 * (-7.16) = 0.09
```

Iteration 3:

1. Forward Pass: Same as Iteration 1 and 2, but now with the updated weight and bias values.
2. Loss Computation: Same as Iteration 1 and 2.
3. Backward Pass (Gradient Computation): Same as Iteration 1 and 2, but now with the updated weight and bias values.
4. Parameter Update (Optimization Step):

```
Updated weight w = w - learning_rate * (∂loss/∂w)
Updated bias b = b - learning_rate * (∂loss/∂b)
```

```
Updated weight w = 1.15 - 0.01 * (-6.31) = 1.
```