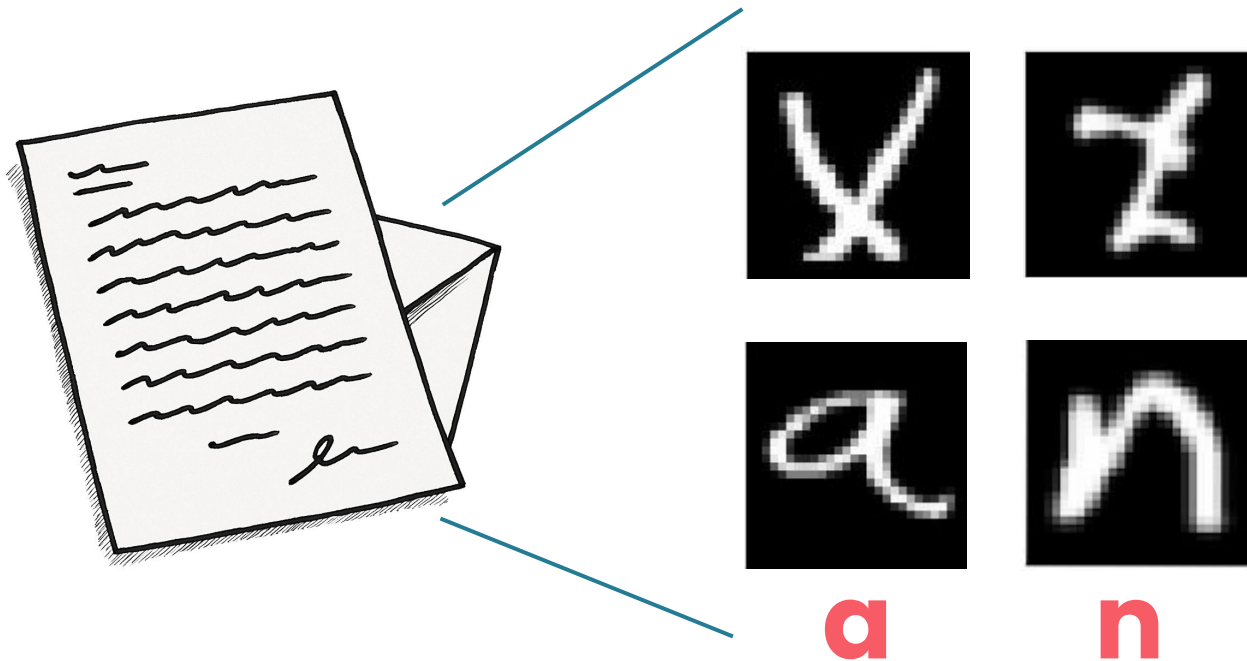




DeepLearning.AI

Decoding a Secret Message

The PyTorch Workflow



Tcsq Zsoqcdgc,

Br1c fnc IuHrqgn groqzc mz ermde xcpp.

Tr drf wrqecf fr iccl fnc psaz
mdfcqczfmde sdy cdesemde. Ksuac uro gropy
nstc fnc zfoycdfz fqu fr ycgryc bu bczzu
nsdyxqmfmdc. Hnsf bmenf ac s amf frr
gnsppcdemde fnroen.

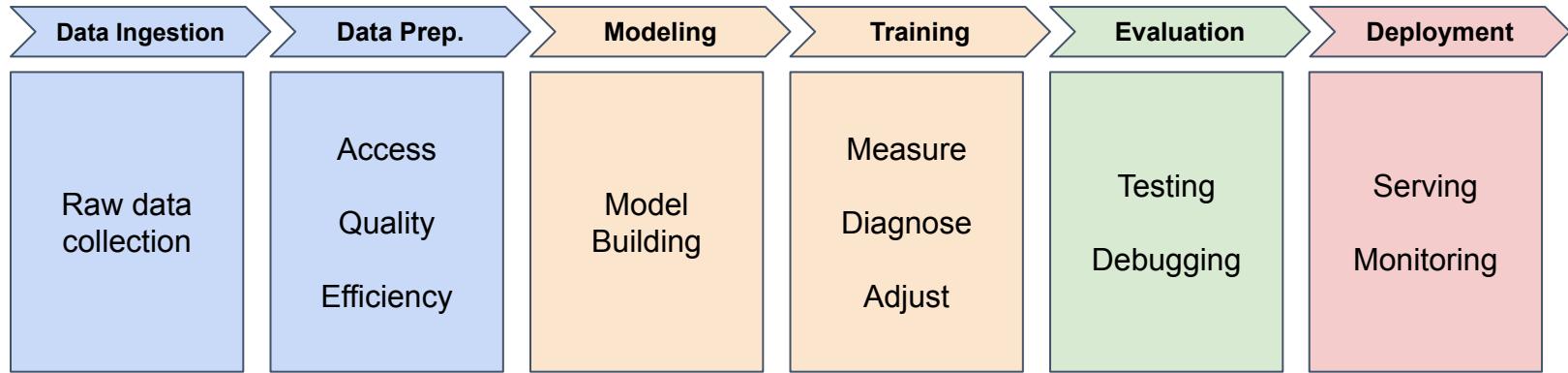
P sb mblqczzcy uro sqc sapc fr qcsy fnmz.

-Fdyqcx

Images



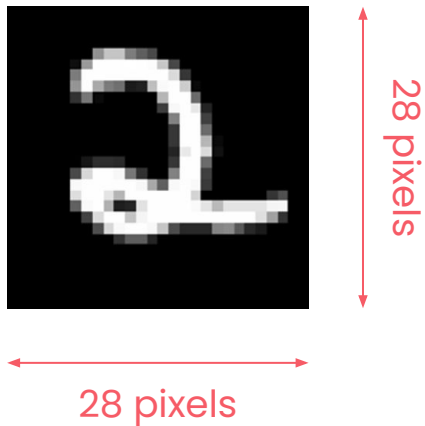
The ML Pipeline



MNIST Database

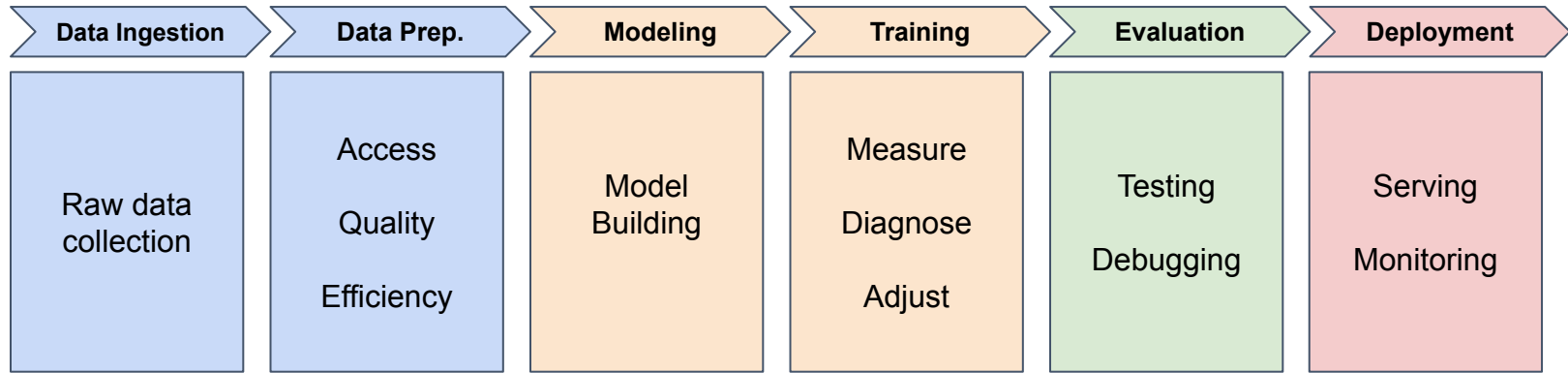


MNIST Database

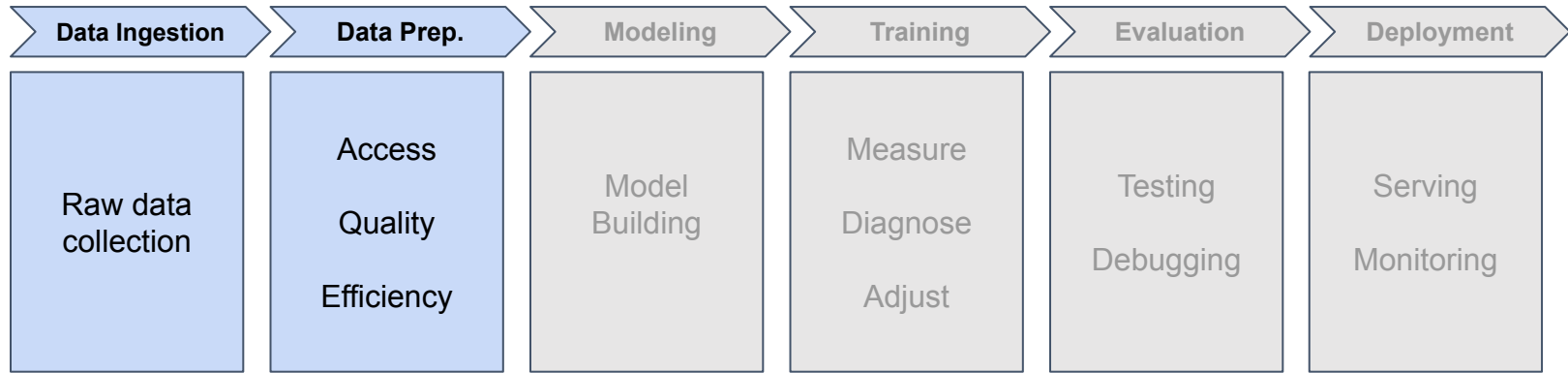


- Grayscale
- Centered
- 0 to 9 digits

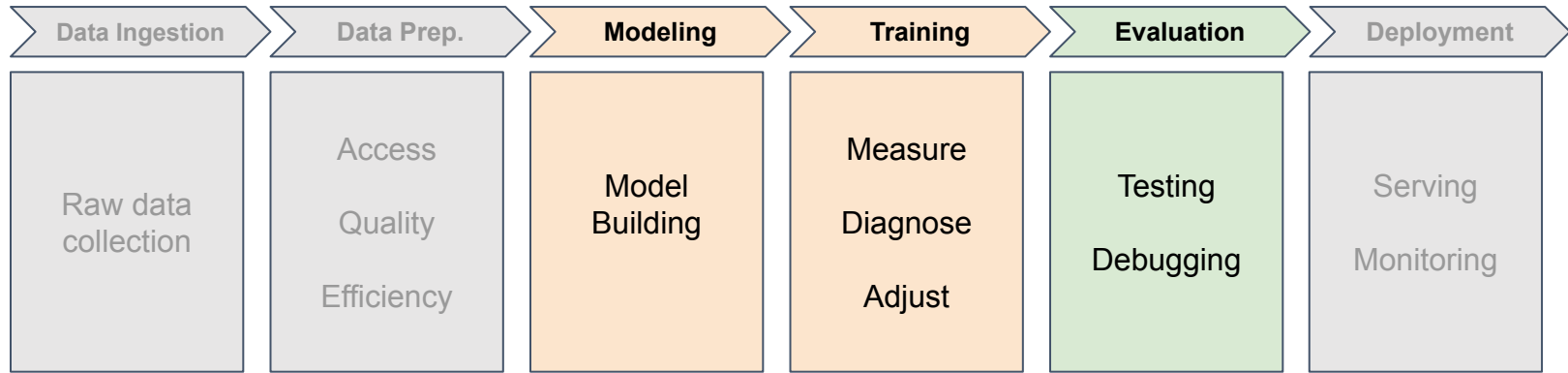
The ML Pipeline



The ML Pipeline



The ML Pipeline





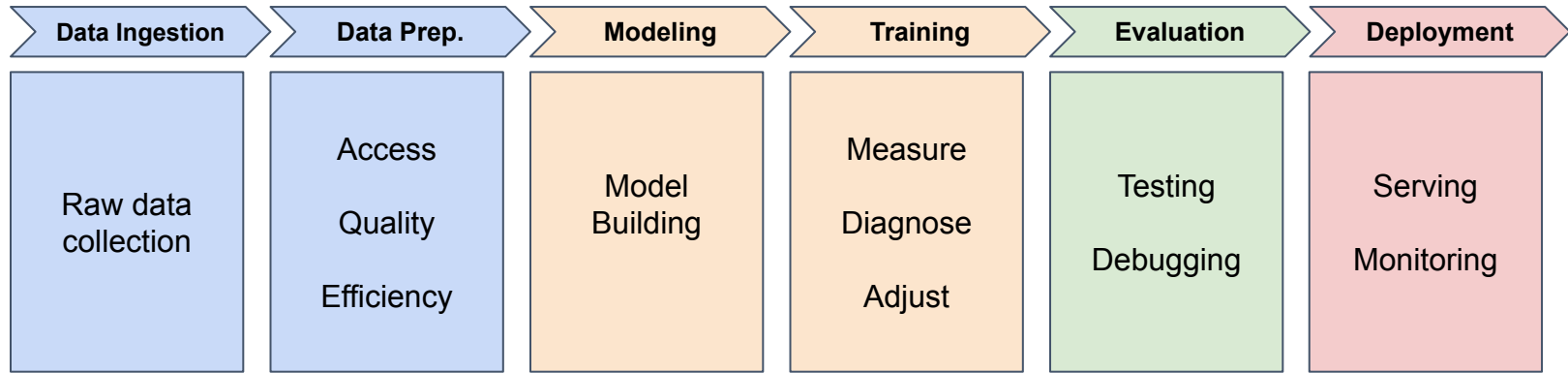
DeepLearning.AI

Overview of the ML Pipeline with PyTorch

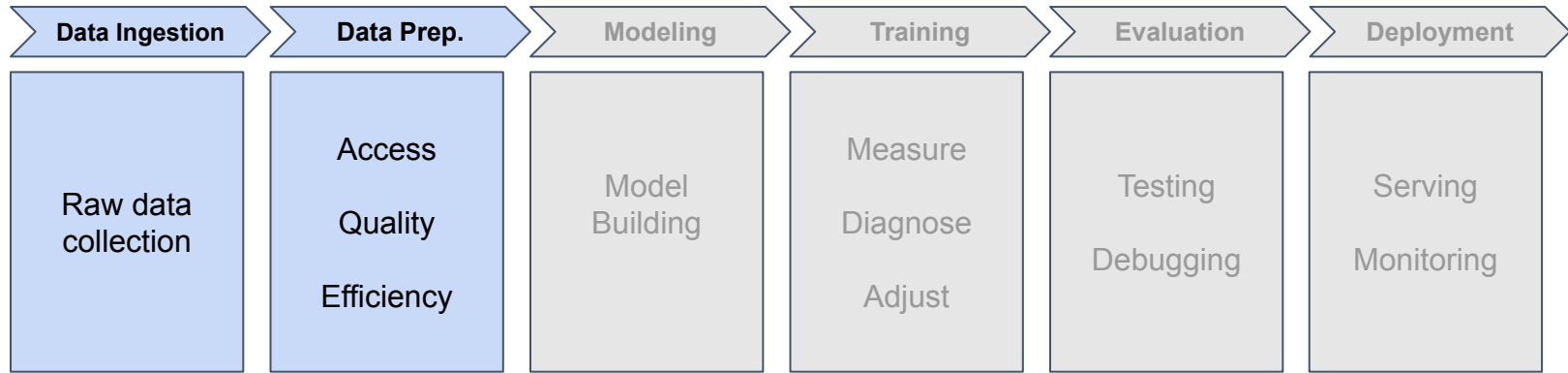
Part 1: Data

The PyTorch Workflow

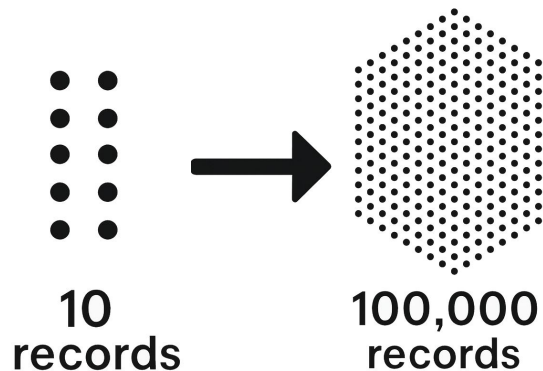
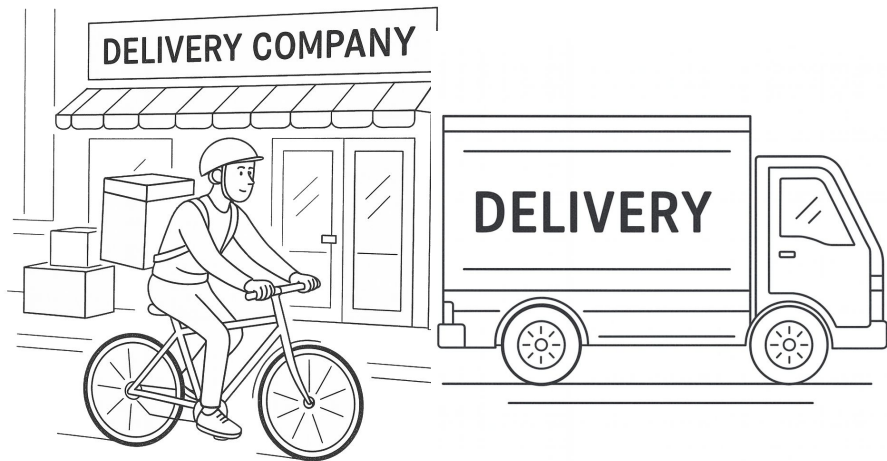
The ML Pipeline



The ML Pipeline



Data Ingestion

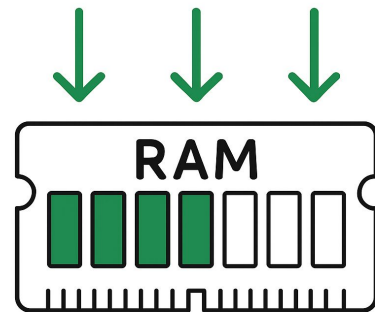


Data Ingestion



```
# Try to load all delivery data into memory
all_distances = []
all_times = []

for i in range(100000):
    distance, time = load_delivery_record(i)
    all_distances.append(distance)
    all_times.append(time)
```

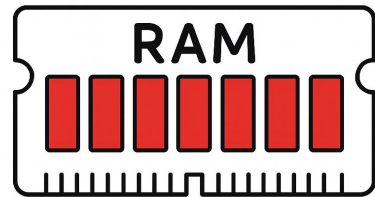


Data Ingestion



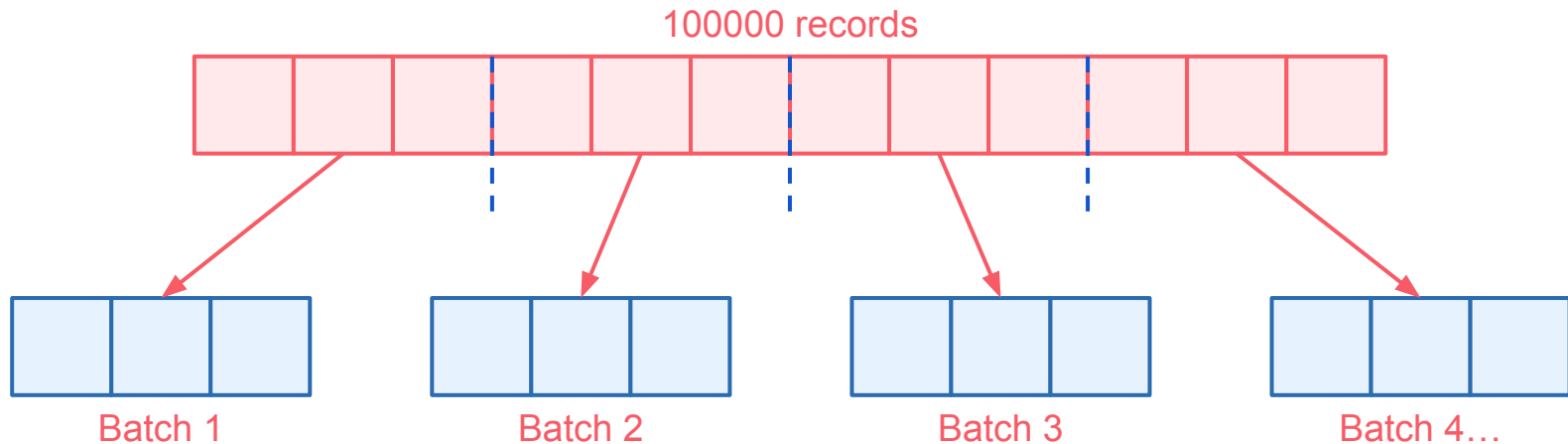
```
# Try to load all delivery data into memory
all_distances = []
all_times = []

for i in range(100000):
    distance, time = load_delivery_record(i)
    all_distances.append(distance)
    all_times.append(time)
```



CRASH!

Data Ingestion



Core Data Tools



transforms

Dataset

DataLoader

PyTorch: transforms



```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((mean,), (std,))
])
```

PyTorch: transforms



```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((mean,), (std,))
])
```

PyTorch: transforms



```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((mean,), (std,))
])
```

PyTorch: transforms



```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((mean,), (std,))
])
```

PyTorch: transforms

Data Ingestion

Data Prep.

Modeling

Training

Evaluation

Deployment

```
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((mean,), (std,))
])
```

[2, 4, 6, 8, 10]

Divide each value by max (10) and turn into tensor

tensor([.2, .4, .6, .8, 1])

PyTorch: transforms

Data Ingestion

Data Prep.

Modeling

Training

Evaluation

Deployment

```
transform = transforms.Compose([  
    transforms.ToTensor(),  
    transforms.Normalize((mean,), (std,))  
])
```

tensor([.2, .4, .6, .8, 1])

Normalize to standard distribution

tensor([-1.41, -0.70, 0, .70, 1.41])

PyTorch: Dataset



PyTorch has many pre-built datasets

```
dataset = SomeDataset('./data', train=True, download=True, transform=transform)
```

- Where your data lives on disk
- How to load a specific sample
- How many total samples
- How to apply **transforms**

PyTorch: Dataset



```
# PyTorch has many pre-built datasets  
dataset = SomeDataset('./data', train=True, download=True, transform=transform)
```

Path to store the data locally on your computer

PyTorch: Dataset



```
# PyTorch has many pre-built datasets  
dataset = SomeDataset('./data', train=True, download=True, transform=transform)
```

Training/test sets of data

PyTorch: Dataset



```
# PyTorch has many pre-built datasets  
dataset = SomeDataset('./data', train=True, download=True, transform=transform)
```

Downloads the data

PyTorch: Dataset



```
# PyTorch has many pre-built datasets
dataset = SomeDataset('./data', train=True, download=True, transform=transform)

first_item = dataset[0] # Just gets one
```

PyTorch: Dataloader



```
dataset_loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

PyTorch: Dataloader



```
dataset_loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Size of your batches

PyTorch: Dataloader



```
dataset_loader = DataLoader(dataset, batch_size=32, shuffle=True)
```

Shuffle your data

PyTorch Data Pipeline



1. Define your transforms

```
transform = SomeTransform() # ToTensor + Normalize
```

2. Create Datasets with transforms

```
train_dataset = YourDataset('./data', train=True, transform=transform)
```

3. Create DataLoaders

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

PyTorch Data Pipeline



1. Define your transforms

```
transform = SomeTransform() # ToTensor + Normalize
```

2. Create Datasets with transforms

```
train_dataset = YourDataset('./data', train=True, transform=transform)
```

3. Create DataLoaders

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

4. Training

```
for batch_idx, (data, labels) in enumerate(train_loader):  
    # Data arrives in batches, already transformed  
    # Your model processes this batch  
    output = model(data)
```



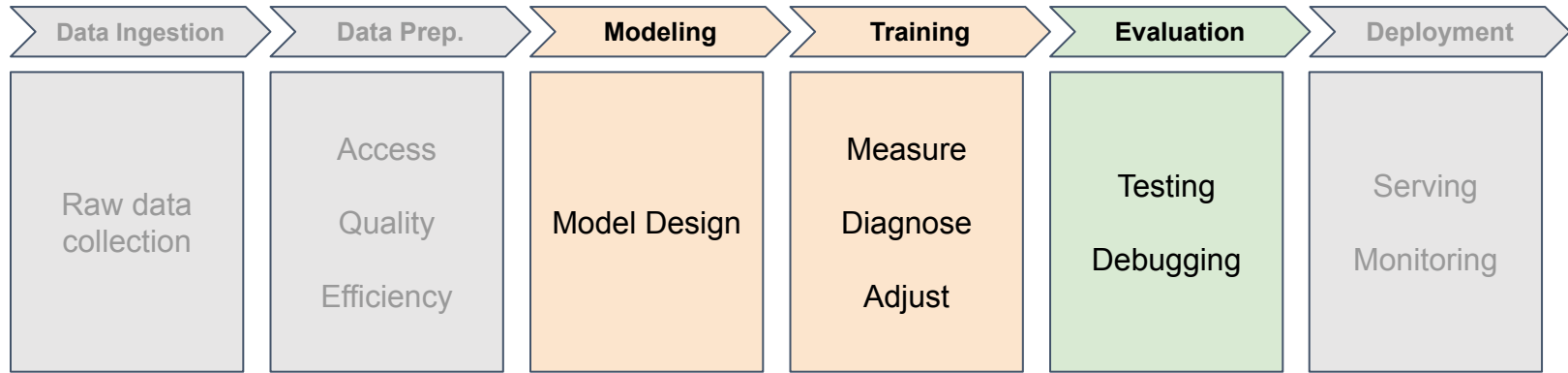
DeepLearning.AI

Overview of the ML Pipeline with PyTorch

Part 2: Models

The PyTorch Workflow

The ML Pipeline



Model Building



```
model = nn.Sequential(  
    nn.Linear(1, 20),  
    nn.ReLU(),  
    nn.Linear(20, 1)  
)
```

```
class ExampleModel(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.layer_1 = nn.Linear(1, 20)  
        self.relu = nn.ReLU()  
        self.layer_2 = nn.Linear(20, 1)  
  
    def forward(self, x):  
        x = self.layer_1(x)  
        x = self.relu(x)  
        x = self.layer_2(x)  
        return x
```

Model Building



```
model = nn.Sequential(  
    nn.Linear(1, 20),  
    nn.ReLU(),  
    nn.Linear(20, 1)  
)
```

```
class ExampleModel(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.layer_1 = nn.Linear(1, 20)  
        self.relu = nn.ReLU()  
        self.layer_2 = nn.Linear(20, 1)  
  
    def forward(self, x):  
        x = self.layer_1(x)  
        x = self.relu(x)  
        x = self.layer_2(x)  
        return x
```

Model Building



```
model = nn.Sequential(  
    nn.Linear(1, 20),  
    nn.ReLU(),  
    nn.Linear(20, 1)  
)
```


```
class ExampleModel(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.layer_1 = nn.Linear(1, 20)  
        self.relu = nn.ReLU()  
        self.layer_2 = nn.Linear(20, 1)
```

```
def forward(self, x):  
    x = self.layer_1(x)  
    x = self.relu(x)  
    x = self.layer_2(x)  
    return x
```


Model Building



```
model = nn.Sequential(  
    nn.Linear(1, 20),  
    nn.ReLU(),  
    nn.Linear(20, 1)  
)
```



```
class ExampleModel(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.layer_1 = nn.Linear(1, 20)  
        self.relu = nn.ReLU()  
        self.layer_2 = nn.Linear(20, 1)  
  
    def forward(self, x):  
        x = self.layer_1(x)  
        x = self.relu(x)  
        x = self.layer_2(x)  
        return x
```



Model Building



```
model = ExampleModel()  
output = model.forward(data)  # Call the forward method you just wrote?
```

```
model = nn.Sequential(nn.Linear(1, 20), nn.ReLU(), nn.Linear(20, 1))  
output = model(data)
```

Model Building



```
model = ExampleModel()  
output = model(data) model.forward(data)
```

```
model = nn.Sequential(nn.Linear(1, 20), nn.ReLU(), nn.Linear(20, 1))  
output = model(data)
```

Model Building



```
model = ExampleModel()  
output = model(data)
```

- Internal checks
- Tracks the necessary math
- Setup model updating

Model Building



```
class ExampleModel(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.fc1 = nn.Linear(1, 20)  
  
        ...
```

Model Building



```
class ExampleModel(nn.Module):  
    def __init__(self):  
        # super().__init__() # What if you skip this?  
        self.fc1 = nn.Linear(784, 128)  
  
        ...
```

Model Building

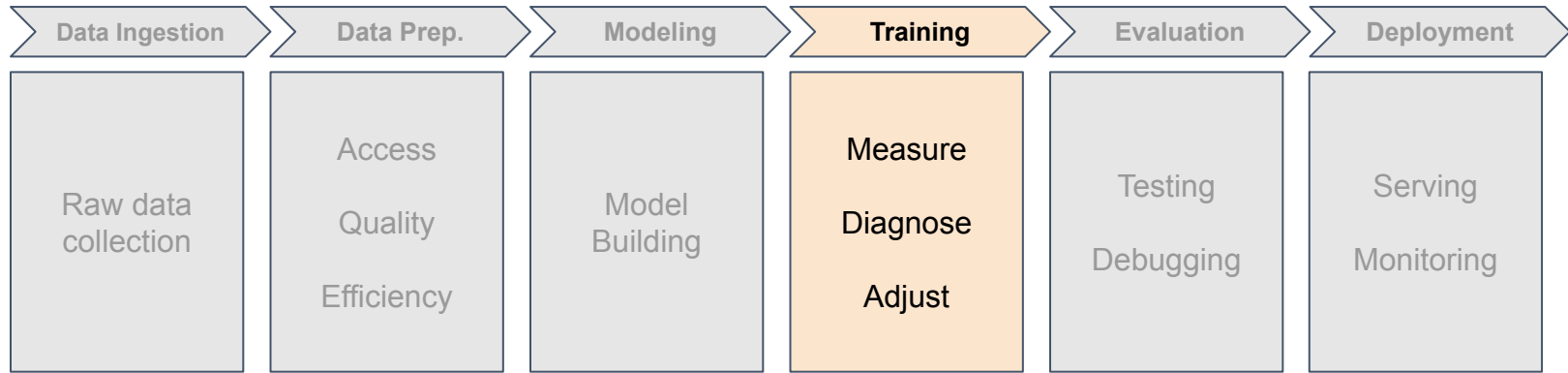


```
class ExampleModel(nn.Module):
    def __init__(self):
        # super().__init__() # What if you skip this?
        self.fc1 = nn.Linear(784, 128)

        ...

# AttributeError: cannot assign parameter before Module.__init__() call
model = ExampleModel()
```

The ML pipeline



Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    optimizer.step()           # Trying to update...  
    loss.backward()        # ...before calculating what to update!
```

Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

Training



```
for epoch in range(epochs):  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.zero_grad()  
    optimizer.step()
```

zero_grad() -> *after* backward()

Training



~~optimizer.zero_grad()~~ ← `zero_grad()` -> *outside the loop*

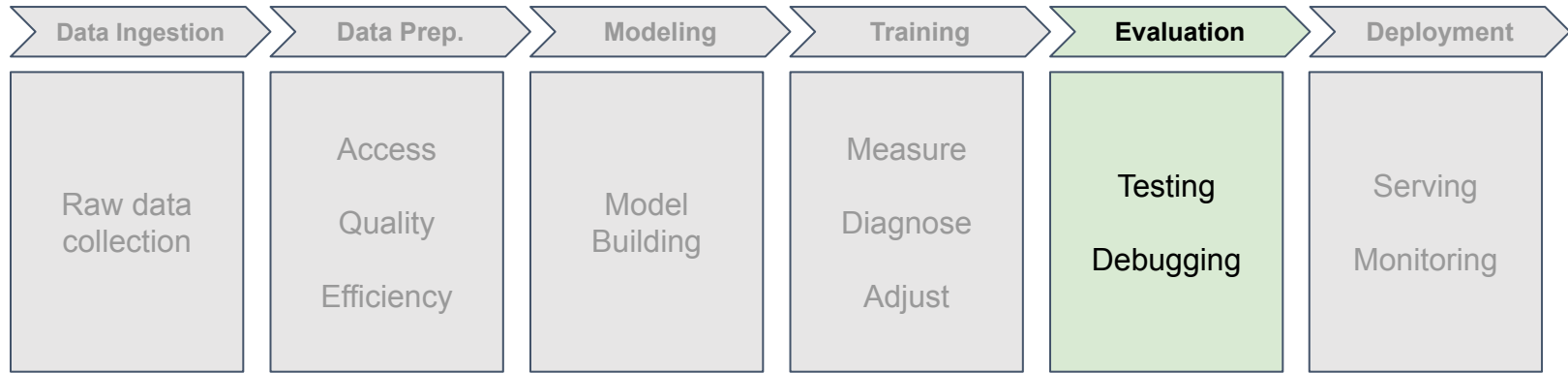
```
for epoch in range(epochs):  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

The ML pipeline



Evaluation



```
model.eval() # Set evaluation mode (NOT "evaluate my model"!)
with torch.no_grad(): # Disable gradient tracking
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1) # Get the class with highest score
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy}%')
```

Evaluation



```
model.eval() # Set evaluation mode (NOT "evaluate my model!")
```

```
with torch.no_grad(): # Disable gradient tracking
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1) # Get the class with highest score
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy}%')
```

Evaluation



```
model.eval() # Set evaluation mode (NOT "evaluate my model!")
with torch.no_grad(): # Disable gradient tracking
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1) # Get the class with highest score
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy}%')
```

Evaluation



```
model.eval() # Set evaluation mode (NOT "evaluate my model"!)
with torch.no_grad(): # Disable gradient tracking
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs, 1) # Get the class with highest score
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

    accuracy = 100 * correct / total
    print(f'Accuracy: {accuracy}%')
```

Evaluation



```
model.train() # Set the model back to training mode
```



DeepLearning.AI

LOSS

The PyTorch Workflow

```
loss = loss_function(outputs, targets)
loss.backward()
optimizer.step()
```

Training



```
for epoch in range(epochs):  
    optimizer.zero_grad()  
    outputs = model(X)  
    loss = loss_function(outputs, y)  
    loss.backward()  
    optimizer.step()
```

Measure, Diagnose, Update

```
loss = loss_function(outputs, targets)
loss.backward()
optimizer.step()
```

Measure, Diagnose, Update

```
loss = loss_function(outputs, targets) -> measure  
loss.backward()  
optimizer.step()
```

Measure, Diagnose, Update

```
loss = loss_function(outputs, targets) -> measure  
loss.backward() -> diagnose  
optimizer.step()
```

Measure, Diagnose, Update

```
loss = loss_function(outputs, targets) -> measure  
loss.backward() -> diagnose  
optimizer.step() -> update
```

Measure, Diagnose, Update

```
loss = loss_function(outputs, targets) -> measure  
loss.backward() -> diagnose  
optimizer.step() -> update
```

Measure, Diagnose, Update

```
loss = loss_function(outputs, targets) -> measure  
loss.backward() -> diagnose  
optimizer.step() -> update
```

Measuring Loss

```
loss = loss_function(outputs, targets) -> measure  
loss.backward() -> diagnose  
optimizer.step() -> update
```

Measuring Loss

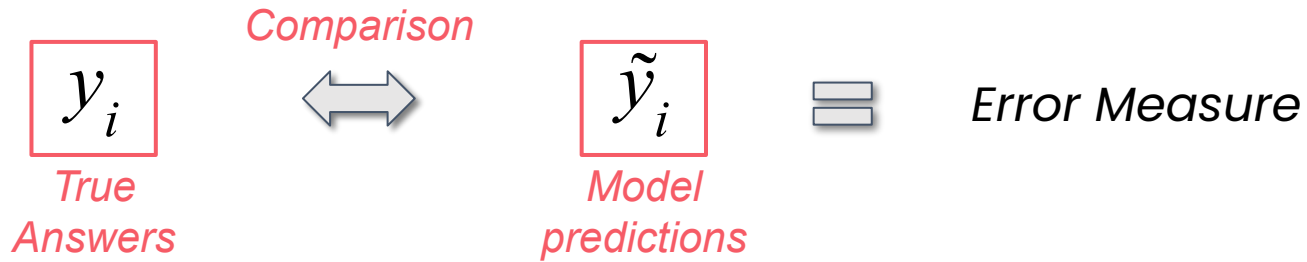
```
loss_function = nn.MSELoss()
```

Measuring Loss

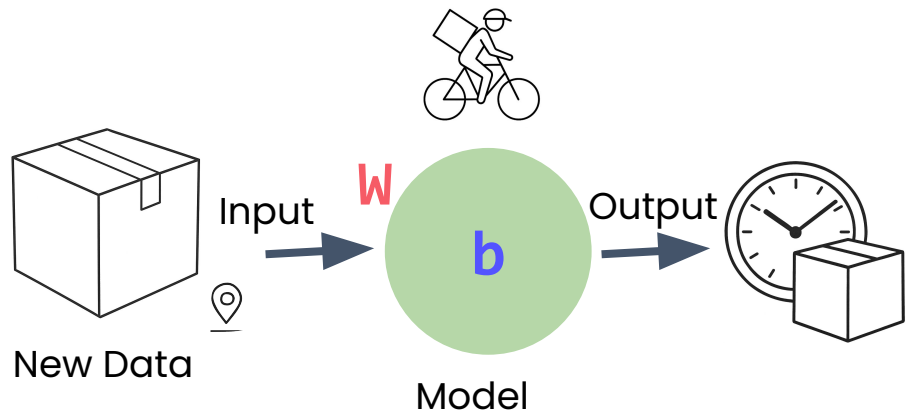
```
loss_function = nn.MSELoss()
```

```
loss_function = nn.CrossEntropyLoss()
```

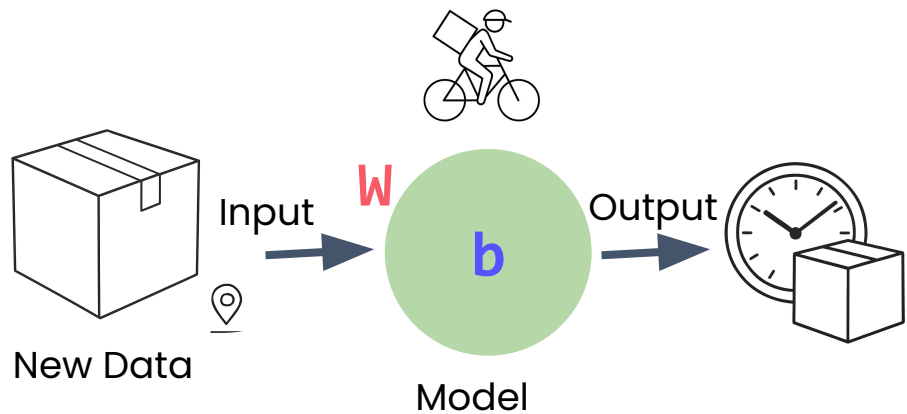
Loss Functions



Mean Squared Error

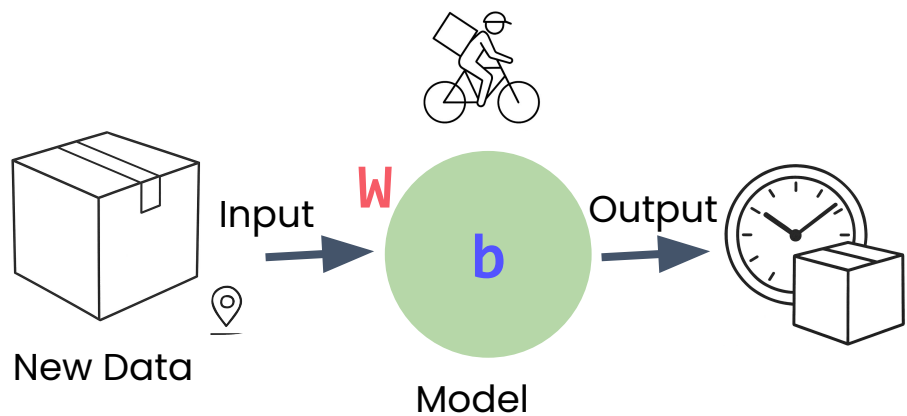


Mean Squared Error



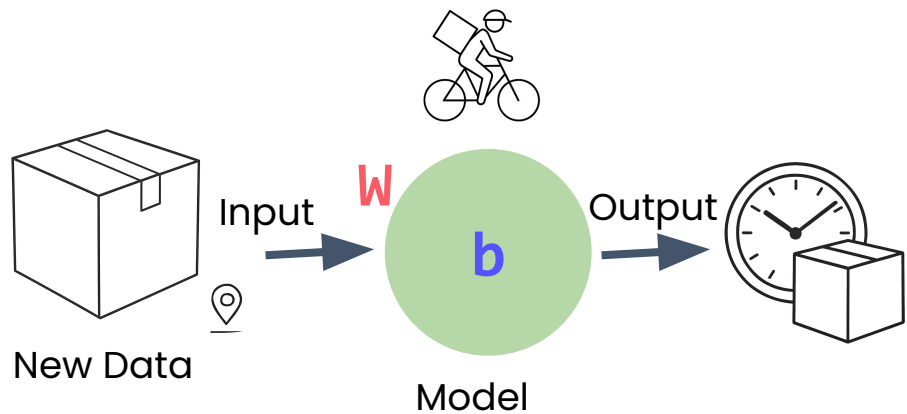
y_i	\tilde{y}_i
<i>Real</i>	<i>Predicted</i>

Mean Squared Error



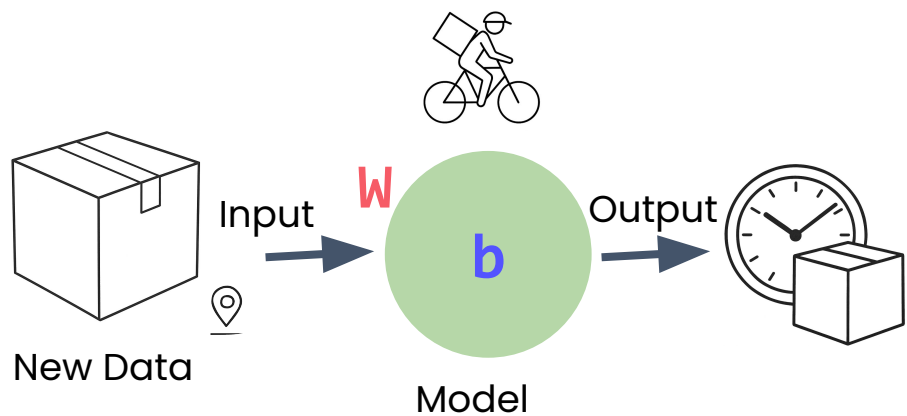
y_i	\tilde{y}_i
<i>Real</i>	<i>Predicted</i>
4	6

Mean Squared Error



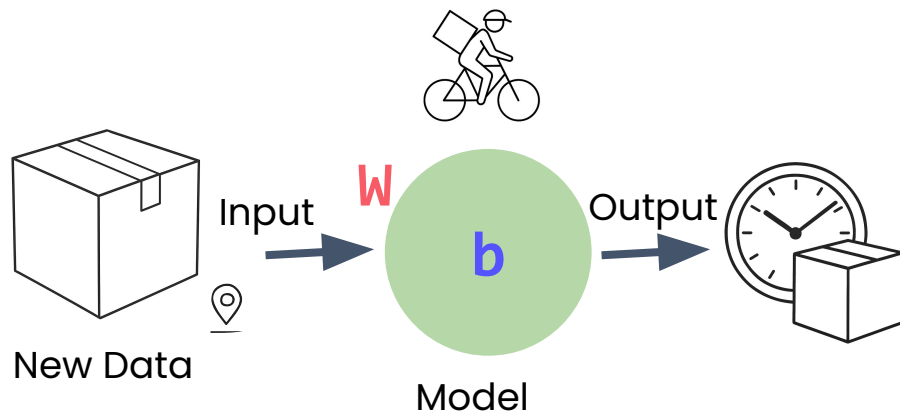
y_i	\tilde{y}_i
<i>Real</i>	<i>Predicted</i>
4	6
5	3

Mean Squared Error



y_i	\tilde{y}_i	$y_i - \tilde{y}_i$
<i>Real</i>	<i>Predicted</i>	<i>Difference</i>
4	6	-2
5	3	+2

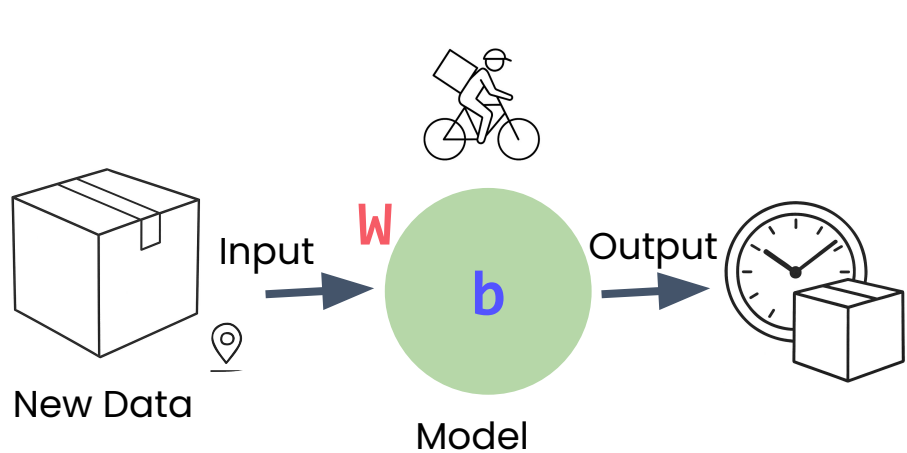
Mean Squared Error



Loss  Average Error

y_i	\tilde{y}_i	$y_i - \tilde{y}_i$
<i>Real</i>	<i>Predicted</i>	<i>Difference</i>
4	6	-2
5	3	+2

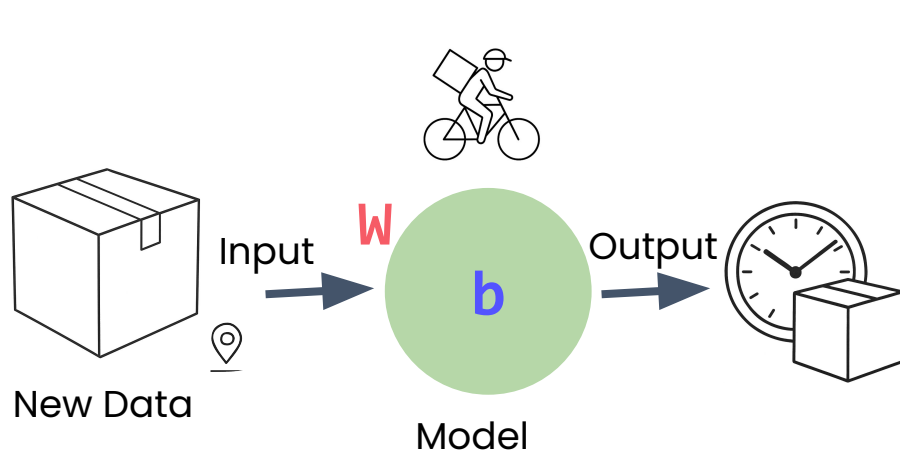
Mean Squared Error



Loss = Average Error

y_i	\tilde{y}_i	$y_i - \tilde{y}_i$
Real	Predicted	Difference
4	6	-2
5	3	+2
Average		0

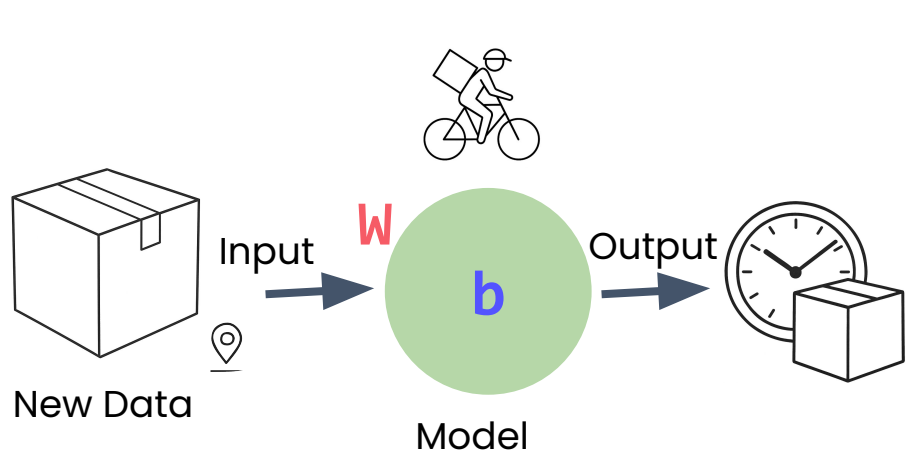
Mean Squared Error



Loss = Average Error

y_i	\tilde{y}_i	$(y_i - \tilde{y}_i)^2$
Real	Predicted	Difference ²
4	6	4
5	3	4

Mean Squared Error



Loss = Average Error

y_i	\tilde{y}_i	$(y_i - \tilde{y}_i)^2$
Real	Predicted	Difference ²
4	6	4
5	3	4
Average		4

Mean Squared Error

y_i	\tilde{y}_i	$y_i - \tilde{y}_i$	$(y_i - \tilde{y}_i)^2$
<i>Real</i>	<i>Predicted</i>	<i>Difference</i>	<i>Difference²</i>

Mean Squared Error

y_i	\tilde{y}_i	$y_i - \tilde{y}_i$	$(y_i - \tilde{y}_i)^2$
<i>Real</i>	<i>Predicted</i>	<i>Difference</i>	<i>Difference²</i>
2	12	10	$10^2 = 100$
2	3	1	$1^2 = 1$

**10 minute
difference is
100x worse
than 1 minute**

MSELoss

```
loss_function = nn.MSELoss()  
loss = loss_function(predictions, targets)
```

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \tilde{y}_i)^2$$

Mean Squared Error

- Makes sure all mistakes count
- Punishes bigger errors more than small ones

Cross Entropy Loss

```
loss_function = nn.CrossEntropyLoss()  
loss = loss_function(outputs, labels)
```



Output:

```
C:>  
Digit: 0 1 2 3 4 5 6 7 8 9  
Conf: 0.05 0.02 0.08 0.70 0.03 0.04 0.02 0.03 0.02 0.01
```

Cross Entropy Loss

```
loss_function = nn.CrossEntropyLoss()  
loss = loss_function(outputs, labels)
```



Cross Entropy Loss

```
loss_function = nn.CrossEntropyLoss()  
loss = loss_function(outputs, labels)
```

- Punishes overconfident wrong answers

Cross Entropy Loss

```
loss_function = nn.CrossEntropyLoss()  
loss = loss_function(outputs, labels)
```



Output:

```
C:>  
Digit: 0      1      2      3      4      5      6      7      8      9  
Conf:  0.01  0.00  0.01  0.00  0.01  0.00  0.01  0.95  0.01  0.00
```

High Loss

Cross Entropy Loss

```
loss_function = nn.CrossEntropyLoss()  
loss = loss_function(outputs, labels)
```



Output:

```
C:>  
Digit: 0      1      2      3      4      5      6      7      8      9  
Conf:  0.01  0.10  0.01  0.10  0.01  0.10  0.01  0.55  0.01  0.10
```

Smaller Loss

Don't mix up your loss functions!

- **MSE** for classification works poorly
- **CrossEntropyLoss** for regression will break
- Do not compare raw results!

?

MSE = .08

vs.

CrossEntropy = 2.3

?

What loss function should I use?

- **MSE**: Predicting a number (such as temperature, price, or distance)
- **CrossEntropyLoss**: Predicting a category (like a digit, animal, or word)

Loss Functions

nn.MSELoss() *# Measures average squared difference - common for regression (predicting numbers)*

nn.CrossEntropyLoss() *# For multi-class classification (picking 1 out of many categories)*

nn.L1Loss() *# Measures average absolute difference - regression that's less sensitive to outliers*

nn.BCEWithLogitsLoss() *# For binary classification (yes/no, true/false decisions)*

nn.NLLLoss() *# Multi-class classification when you're using LogSoftmax activation*

nn.SmoothL1Loss() *# Regression that balances MSE and L1Loss benefits (also called Huber loss)*

nn.KLDivLoss() *# Measures difference between two probability distributions*

Loss Functions

nn.MSELoss() *# Measures average squared difference - common for regression (predicting numbers)*

nn.CrossEntropyLoss() *# For multi-class classification (picking 1 out of many categories)*

nn.L1Loss() *# Measures average absolute difference - regression that's less sensitive to outliers*

nn.BCEWithLogitsLoss() *# For binary classification (yes/no, true/false decisions)*

nn.NLLLoss() *# Multi-class classification when you're using LogSoftmax activation*

nn.SmoothL1Loss() *# Regression that balances MSE and L1Loss benefits (also called Huber loss)*

nn.KLDivLoss() *# Measures difference between two probability distributions*

LOSS

```
loss = loss_function(outputs, targets) -> measure
```

```
loss.backward() -> diagnose
```

```
optimizer.step() -> update
```



DeepLearning.AI

Optimizers and Gradients

The PyTorch Workflow

The Training Loop

```
loss = loss_function(outputs, targets) -> measure  
loss.backward() -> diagnose  
optimizer.step() -> update
```

The Training Loop

```
loss = loss_function(outputs, targets)
loss.backward() -> diagnose
optimizer.step()
```

The Training Loop

```
loss = loss_function(outputs, targets) -> measure  
loss.backward()  
optimizer.step() -> update
```

The Training Loop

```
loss = loss_function(outputs, targets) -> measure
```

```
loss.backward() -> diagnose
```

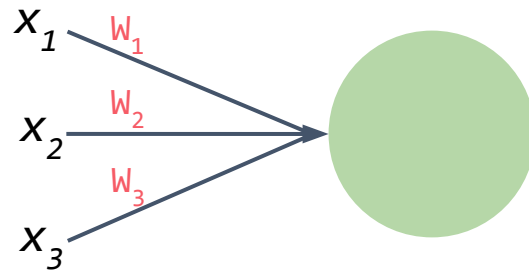
```
optimizer.step() -> update
```

The Training Loop

```
loss = loss_function(outputs, targets)
loss.backward() -> diagnose
optimizer.step()
```

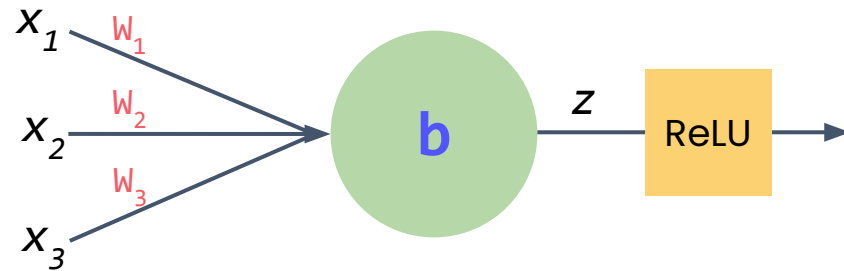
*What caused
the high loss?*

Neural Networks



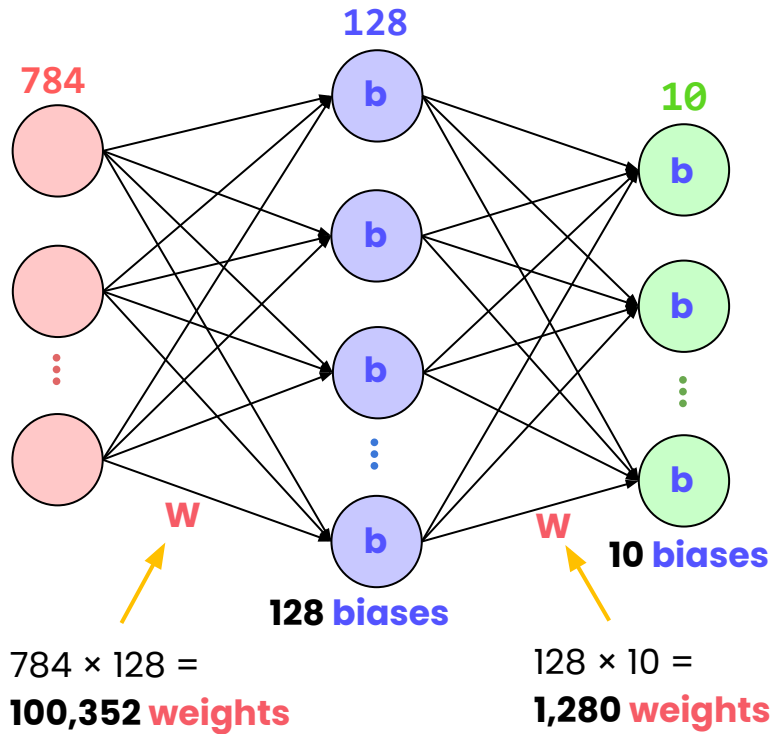
$$w_1 * x_1 + w_2 * x_2 + w_3 * x_3$$

Neural Networks



$$z = w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + b$$


Neural Networks



Total:
101,770 parameters!

Gradients

```
loss = loss_function(outputs, targets)
loss.backward() -> diagnose
optimizer.step()
```

$$z = w_1 * x_1 + w_2 * x_2 + w_3 * x_3 + b$$


How much did you contribute to the loss?

Gradients

Positive

+2.5: Decrease to reduce loss

Negative

-2.5: Increase to reduce loss

Large Values

+10 or -8.5: very influential

Small Values

+0.001 or -0.0005: barely matters


Gradients

```
loss = loss_function(outputs, targets)
loss.backward() -> diagnose
optimizer.step()
```

*Only calculates
gradients*

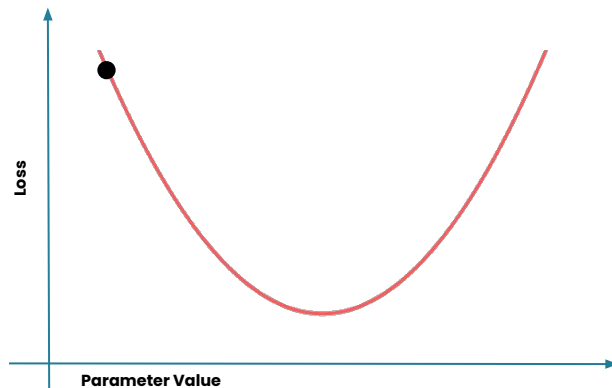
Gradients

```
loss = loss_function(outputs, targets) -> measure  
loss.backward() -> diagnose  
optimizer.step() -> update
```



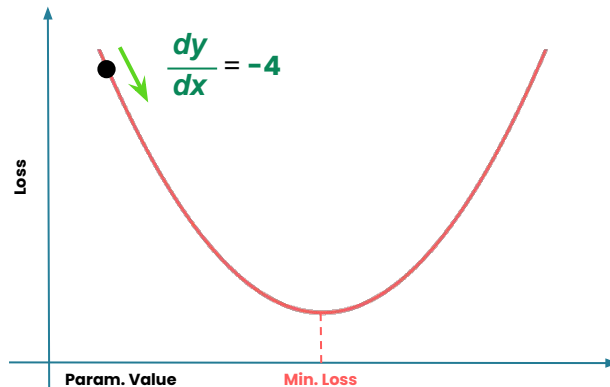
*Adjusts the parameters
to reduce the loss*

Gradients



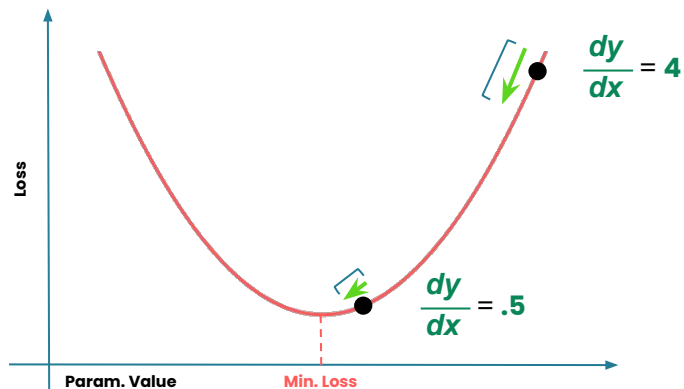
Gradients

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



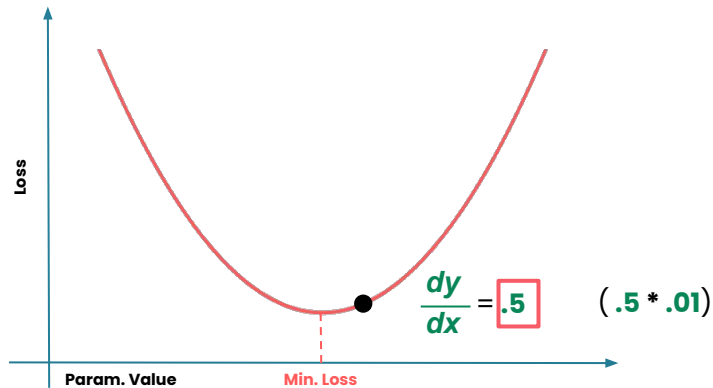
Gradients

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



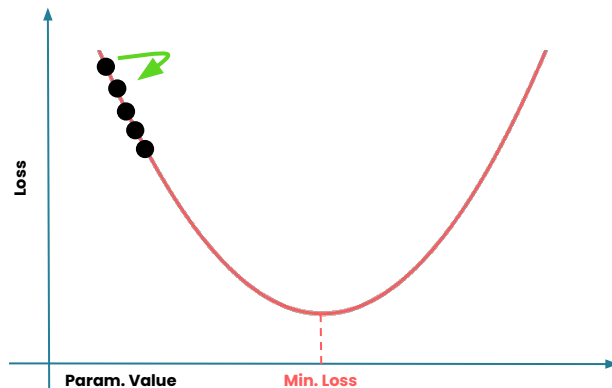
Gradients

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



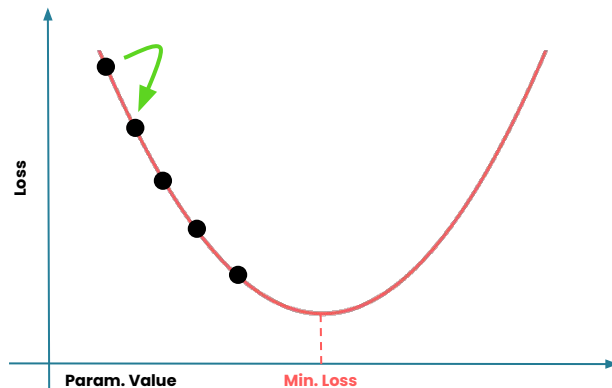
Gradients

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



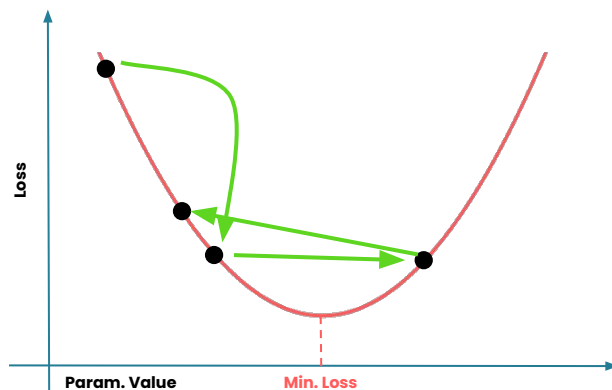
Gradients

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



Gradients

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



Adam

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Adam & SGD

```
# Smaller learning rate  
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)  
  
optimizer = torch.optim.SGD(model.parameters(), lr=0.01)
```

PyTorch Optimizers

Adadelta
Adafactor
Adagrad
Adam
AdamW
SparseAdam
Adamax
ASGD
LBFGS
NAdam
RAdam
RMSprop
Rprop
SGD

Training

```
# Training loop
for epoch in range(500):
    # 1. Reset the optimizer
    optimizer.zero_grad()
    # 2. Make predictions
    outputs = model(inputs)
    # 3. Calculate the loss - how bad was this guess?
    loss = loss_function(outputs, times)
    # 4. Calculate adjustments
    loss.backward()
    # 5. Update the model
    optimizer.step()
```

Training

```
# Training loop
for epoch in range(500):
    # 1. Reset the optimizer
    optimizer.zero_grad()
    # 2. Make predictions
    outputs = model(inputs)
    # 3. Calculate the loss - how bad was this guess?
    loss = loss_function(outputs, times)
    # 4. Calculate adjustments
    loss.backward()
    # 5. Update the model
    optimizer.step()
```

Training

```
# Training loop
for epoch in range(500):
    # 1. Reset the optimizer
    optimizer.zero_grad()
    # 2. Make predictions
    outputs = model(inputs)
    # 3. Calculate the loss - how bad was this guess?
    loss = loss_function(outputs, times)
    # 4. Calculate adjustments
    loss.backward()
    # 5. Update the model
    optimizer.step()
```

The Training Loop

```
loss = loss_function(outputs, targets) -> measure  
loss.backward() -> diagnose  
optimizer.step() -> update
```



DeepLearning.AI

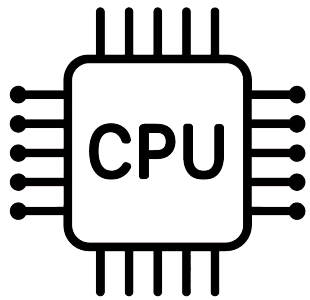
Device Management

The PyTorch Workflow

Device Error

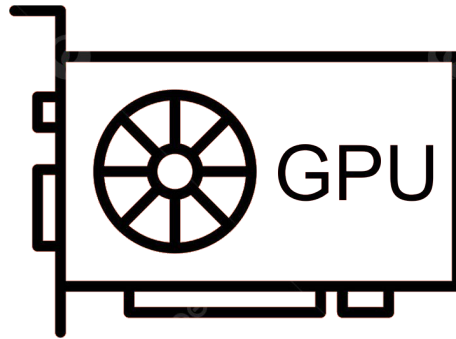
```
RuntimeError: Expected all tensors to be on the same device
```

Devices



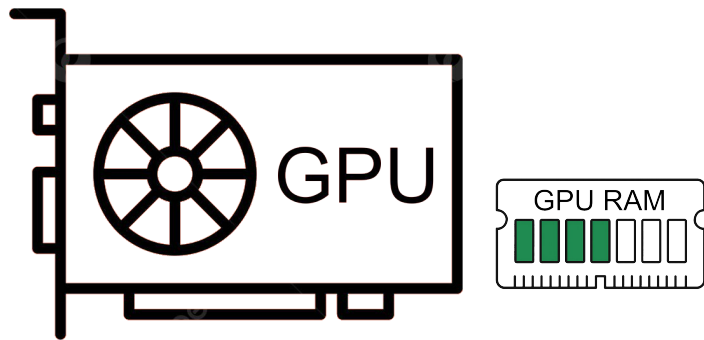
- General-purpose work
- Run operations sequentially

GPU can be 10 to 15 times faster than on a CPU!



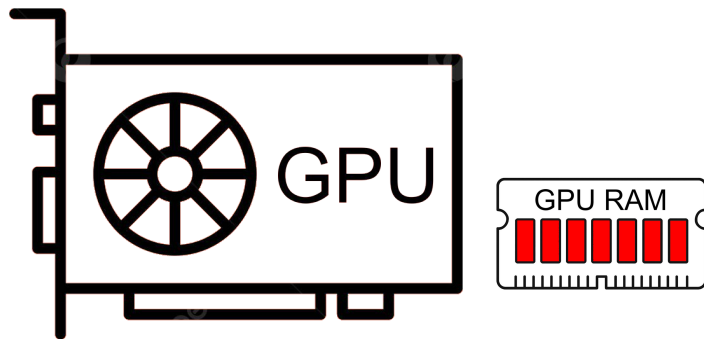
- Parallel operations

Limited GPU Memory



GPU Memory

CUDA out of memory

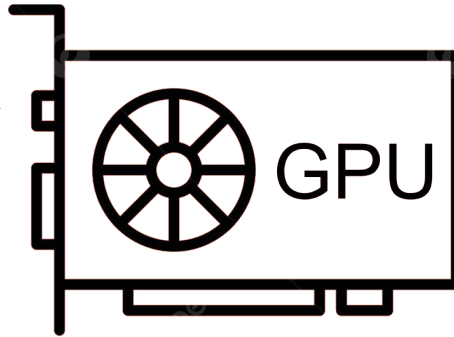


Batch Size of 32-64

Devices

```
torch.cuda.is_available()
```

If True -> GPU available

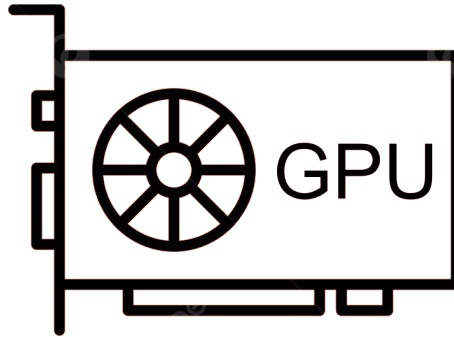


Devices

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

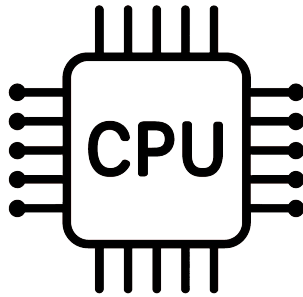
Devices

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



Devices

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```



Devices

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Model .to(device)

```
model = MyModel().to(device)
```

Training Loop

```
for inputs, targets in dataloader:  
    inputs = inputs.to(device)  
    targets = targets.to(device)
```

Check The Device

```
# For tensors  
print(inputs.device)
```

Check The Device

```
# For tensors
print(inputs.device)

# For models
print(next(model.parameters()).device)
```

.to(device)

```
x.to(device)    # This looks like it moves x—but it doesn't
```

.to(device)

```
x.to(device)    # This looks like it moves x—but it doesn't
```

```
x = x.to(device)
```

Training Loop

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MyModel().to(device)
optimizer = optim.Adam(model.parameters())
loss_function = nn.CrossEntropyLoss()

for inputs, targets in dataloader:
    inputs = inputs.to(device)
    targets = targets.to(device)

    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()
    optimizer.step()
```

Training Loop

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MyModel().to(device)
optimizer = optim.Adam(model.parameters())
loss_function = nn.CrossEntropyLoss()

for inputs, targets in dataloader:
    inputs = inputs.to(device)
    targets = targets.to(device)

    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()
    optimizer.step()
```

Training Loop

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MyModel().to(device)
optimizer = optim.Adam(model.parameters())
loss_function = nn.CrossEntropyLoss()

for inputs, targets in dataloader:
    inputs = inputs.to(device)
    targets = targets.to(device)

    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()
    optimizer.step()
```

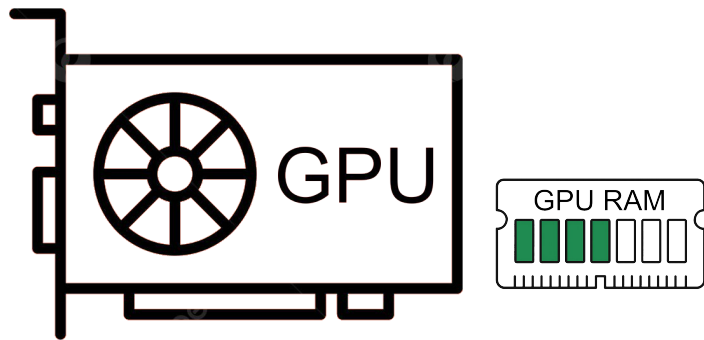
Training Loop

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = MyModel().to(device)
optimizer = optim.Adam(model.parameters())
loss_function = nn.CrossEntropyLoss()

for inputs, targets in dataloader:
    inputs = inputs.to(device)
    targets = targets.to(device)

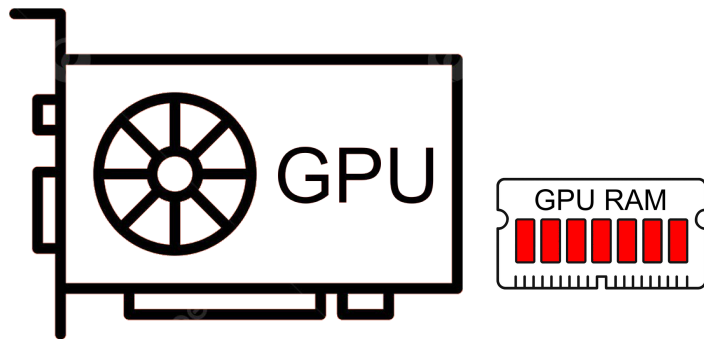
    optimizer.zero_grad()
    outputs = model(inputs)
    loss = loss_function(outputs, targets)
    loss.backward()
    optimizer.step()
```

Limited GPU Memory



GPU Memory

CUDA out of memory



Batch Size of 32-64



DeepLearning.AI

Image Classification

Part 1: Preparing the Data and Building the Model

The PyTorch Workflow

MNIST



Import Libraries

```
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
```

Data Pipeline

```
# Data preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

Data Pipeline

```
# Data preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

Data Pipeline

```
# Data preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

Data Pipeline

```
# Data preprocessing
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

Load MNIST Dataset

```
# Load MNIST dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(
    root='./data', train=False, download=True, transform=transform)
```

Load MNIST Dataset

```
# Load MNIST dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(
    root='./data', train=False, download=True, transform=transform)
```

Load MNIST Dataset

```
# Load MNIST dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(
    root='./data', train=False, download=True, transform=transform)
```

Load MNIST Dataset

```
# Load MNIST dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(
    root='./data', train=False, download=True, transform=transform)
```

Load MNIST Dataset

```
# Load MNIST dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(
    root='./data', train=False, download=True, transform=transform)
```

Load MNIST Dataset

```
# Load MNIST dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(
    root='./data', train=False, download=True, transform=transform)
```

Load MNIST Dataset

```
# Load MNIST dataset
train_dataset = torchvision.datasets.MNIST(
    root='./data', train=True, download=True, transform=transform)
test_dataset = torchvision.datasets.MNIST(
    root='./data', train=False, download=True, transform=transform)
```

Create Data Loaders

```
# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

Create Data Loaders

```
# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

Create Data Loaders

```
# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

Create Data Loaders

```
# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

Create Data Loaders

```
# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)
```

Create the Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.layers(x)  
        return x
```

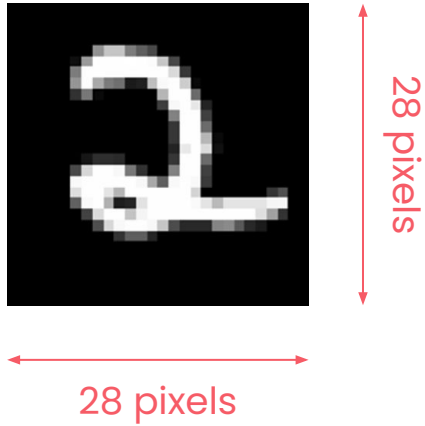
Create the Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.layers(x)  
        return x
```

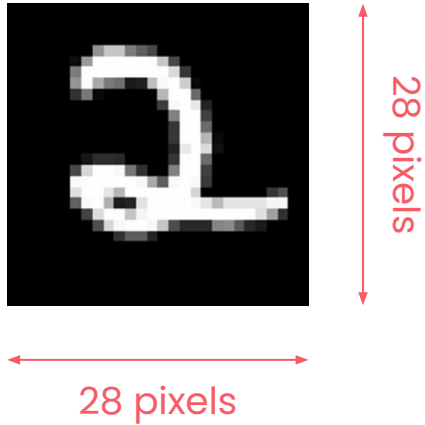
Create the Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.layers(x)  
        return x
```

MNIST Single Image

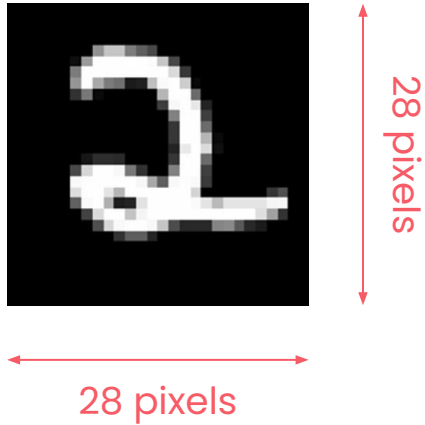


MNIST Single Image



$[1, 28, 28]$ tensor

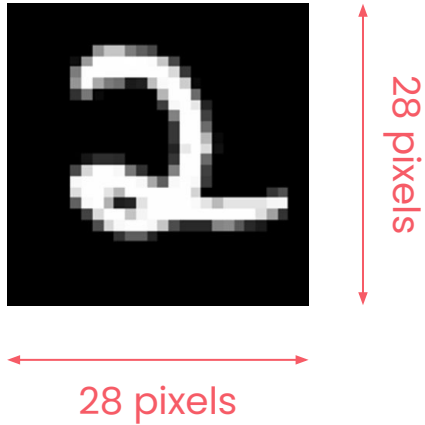
MNIST Single Image



[1, 28, 28] tensor

1 channel -> grayscale

MNIST Single Image

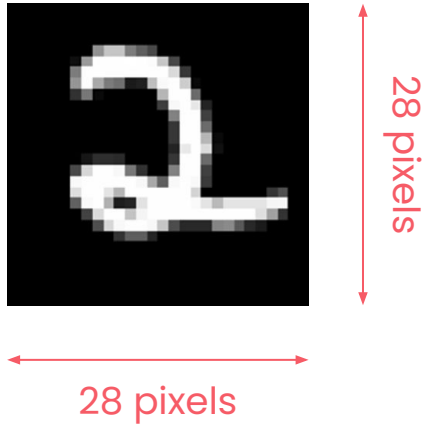


[1, 28, 28] tensor

1 channel -> grayscale

28x28 pixels

MNIST Single Image



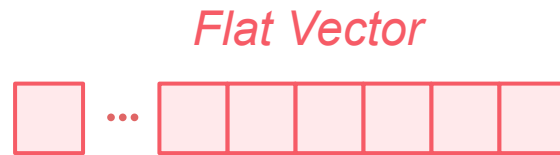
[64, 1, 28, 28] tensor

Batch size

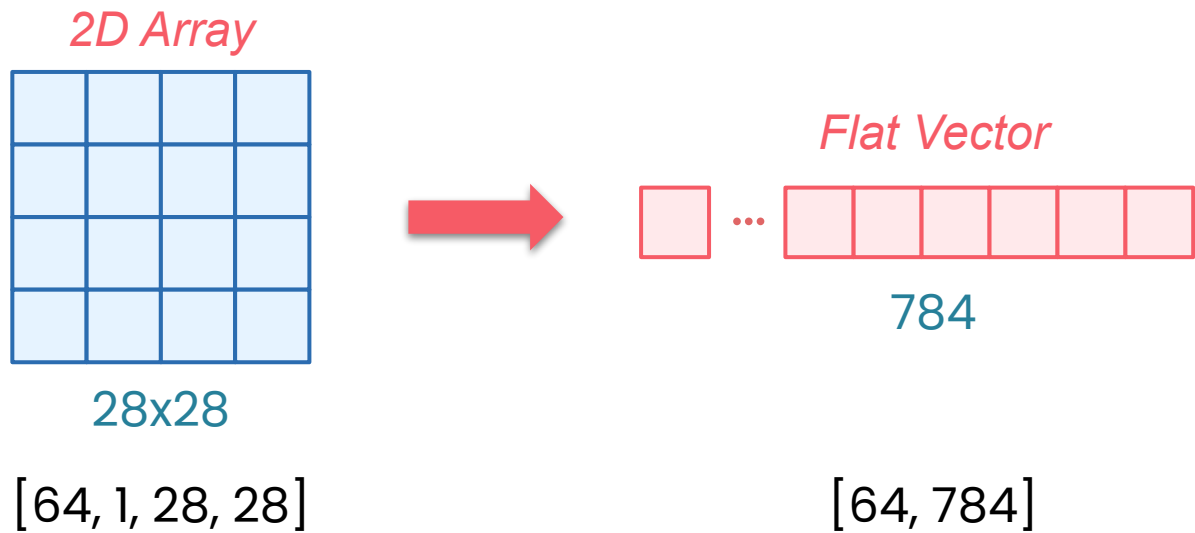
Flatten

Linear Layers

Expect flat vectors



Flatten



Create The Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.layers(x)  
        return x
```

Create The Neural Network

```
class MNISTClassifier(nn.Module):
    def __init__(self):
        super().__init__()
        self.flatten = nn.Flatten()
        self.layers = nn.Sequential(
            nn.Linear(784, 128),
            nn.ReLU(),
            nn.Linear(128, 10)
        )

    def forward(self, x):
        x = self.flatten(x)
        x = self.layers(x)
        return x
```

Create The Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.layers(x)  
        return x
```

Create The Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.layers(x)  
        return x
```

Create The Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )
```

```
def forward(self, x):  
    x = self.flatten(x)  
    x = self.layers(x)  
    return x
```

Create The Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.layers(x)  
        return x
```

Create The Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.layers(x)  
        return x
```

Create The Neural Network

```
class MNISTClassifier(nn.Module):  
    def __init__(self):  
        super().__init__()  
        self.flatten = nn.Flatten()  
        self.layers = nn.Sequential(  
            nn.Linear(784, 128),  
            nn.ReLU(),  
            nn.Linear(128, 10)  
        )  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.layers(x)  
        return x
```



DeepLearning.AI

Image Classification

Part 2: Training and Evaluating the Model

The PyTorch Workflow

Device and Optimizer

```
# Check for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using {device}')

# Initialize model and move to device
model = MNISTClassifier().to(device)

# Loss function and optimizer
loss_function= nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Device and Optimizer

```
# Check for GPU
```

```
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')  
print(f'Using {device}')
```

```
# Initialize model and move to device
```

```
model = MNISTClassifier().to(device)
```

```
# Loss function and optimizer
```

```
loss_function= nn.CrossEntropyLoss()
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Device and Optimizer

```
# Check for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using {device}')

# Initialize model and move to device
model = MNISTClassifier().to(device)

# Loss function and optimizer
loss_function= nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Device and Optimizer

```
# Check for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using {device}')

# Initialize model and move to device
model = MNISTClassifier().to(device)

# Loss function and optimizer
loss_function= nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

Device and Optimizer

```
# Check for GPU
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f'Using {device}')

# Initialize model and move to device
model = MNISTClassifier().to(device)

# Loss function and optimizer
loss_function= nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
```

```

def train_epoch(model, train_loader, loss_function, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model(data)
        loss = loss_function(output, target)
        loss.backward()
        optimizer.step()

        # Track progress
        running_loss += loss.item()
        _, predicted = output.max(1)
        total += target.size(0)
        correct += predicted.eq(target).sum().item()

    # Print every 100 batches
    if batch_idx % 100 == 0 and batch_idx > 0:
        avg_loss = running_loss / 100
        accuracy = 100. * correct / total
        print(f' [{batch_idx * 64}/{60000}] '
              f'Loss: {avg_loss:.3f} | Accuracy: {accuracy:.1f}%')
        running_loss = 0.0

```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
def train_epoch(model, train_loader, loss_function, optimizer, device):  
    model.train()  
    running_loss = 0.0  
    correct = 0  
    total = 0  
  
    for batch_idx, (data, target) in enumerate(train_loader):  
        data, target = data.to(device), target.to(device)  
  
        optimizer.zero_grad()  
        output = model(data)  
        loss = loss_function(output, target)  
        loss.backward()  
        optimizer.step()
```

Training

```
# Track progress
running_loss += loss.item()
_, predicted = output.max(1)
total += target.size(0)
correct += predicted.eq(target).sum().item()

# Print every 100 batches
if batch_idx % 100 == 0 and batch_idx > 0:
    avg_loss = running_loss / 100
    accuracy = 100. * correct / total
    print(f' [{batch_idx * 64}/{60000}] '
          f'Loss: {avg_loss:.3f} | Accuracy: {accuracy:.1f}%')
    running_loss = 0.0
```

Training

```
# Track progress
running_loss += loss.item()
_, predicted = output.max(1)
total += target.size(0)
correct += predicted.eq(target).sum().item()

# Print every 100 batches
if batch_idx % 100 == 0 and batch_idx > 0:
    avg_loss = running_loss / 100
    accuracy = 100. * correct / total
    print(f' [{batch_idx * 64}/{60000}] '
          f'Loss: {avg_loss:.3f} | Accuracy: {accuracy:.1f}%')
    running_loss = 0.0
```

Training

```
# Track progress
running_loss += loss.item()
_, predicted = output.max(1)
total += target.size(0)
correct += predicted.eq(target).sum().item()

# Print every 100 batches
if batch_idx % 100 == 0 and batch_idx > 0:
    avg_loss = running_loss / 100
    accuracy = 100. * correct / total
    print(f' [{batch_idx * 64}/{60000}] '
          f'Loss: {avg_loss:.3f} | Accuracy: {accuracy:.1f}%')
    running_loss = 0.0
```

Training

```
# Track progress
running_loss += loss.item()
_, predicted = output.max(1)
total += target.size(0)
correct += predicted.eq(target).sum().item()

# Print every 100 batches
if batch_idx % 100 == 0 and batch_idx > 0:
    avg_loss = running_loss / 100
    accuracy = 100. * correct / total
    print(f' [{batch_idx * 64}/{60000}] '
          f'Loss: {avg_loss:.3f} | Accuracy: {accuracy:.1f}%')
    running_loss = 0.0
```

Training

Output:

```
C:>
```

```
Epoch: 1
```

```
Loss: 0.641 | Acc: 81.58%
```

```
Loss: 0.321 | Acc: 90.25%
```

```
Loss: 0.284 | Acc: 91.84%
```

```
Loss: 0.235 | Acc: 93.23%
```

```
Loss: 0.200 | Acc: 93.94%
```

```
Loss: 0.182 | Acc: 94.52%
```

```
Loss: 0.180 | Acc: 94.31%
```

```
Loss: 0.175 | Acc: 95.00%
```

```
Loss: 0.168 | Acc: 94.91%
```

```

def train_epoch(model, train_loader, loss_function, optimizer, device):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0

    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

        optimizer.zero_grad()
        output = model(data)
        loss = loss_function(output, target)
        loss.backward()
        optimizer.step()

        # Track progress
        running_loss += loss.item()
        _, predicted = output.max(1)
        total += target.size(0)
        correct += predicted.eq(target).sum().item()

    # Print every 100 batches
    if batch_idx % 100 == 0 and batch_idx > 0:
        avg_loss = running_loss / 100
        accuracy = 100. * correct / total
        print(f' [{batch_idx * 64}/{60000}] '
              f'Loss: {avg_loss:.3f} | Accuracy: {accuracy:.1f}%')
        running_loss = 0.0

```

Evaluation

```
def evaluate(model, test_loader, device):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    return 100. * correct / total
```

Evaluation

```
def evaluate(model, test_loader, device):  
    model.eval()  
    correct = 0  
    total = 0  
  
    with torch.no_grad():  
        for inputs, targets in test_loader:  
            inputs, targets = inputs.to(device), targets.to(device)  
            outputs = model(inputs)  
            _, predicted = outputs.max(1)  
            total += targets.size(0)  
            correct += predicted.eq(targets).sum().item()  
  
    return 100. * correct / total
```

Evaluation

```
def evaluate(model, test_loader, device):  
    model.eval()  
    correct = 0  
    total = 0  
  
    with torch.no_grad():  
        for inputs, targets in test_loader:  
            inputs, targets = inputs.to(device), targets.to(device)  
            outputs = model(inputs)  
            _, predicted = outputs.max(1)  
            total += targets.size(0)  
            correct += predicted.eq(targets).sum().item()  
  
    return 100. * correct / total
```

Evaluation

```
def evaluate(model, test_loader, device):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    return 100. * correct / total
```

Evaluation

```
def evaluate(model, test_loader, device):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    return 100. * correct / total
```

Evaluation

```
def evaluate(model, test_loader, device):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    return 100. * correct / total
```

Evaluation

```
def evaluate(model, test_loader, device):
    model.eval()
    correct = 0
    total = 0

    with torch.no_grad():
        for inputs, targets in test_loader:
            inputs, targets = inputs.to(device), targets.to(device)
            outputs = model(inputs)
            _, predicted = outputs.max(1)
            total += targets.size(0)
            correct += predicted.eq(targets).sum().item()

    return 100. * correct / total
```

Evaluation

```
def evaluate(model, test_loader, device):  
    model.eval()  
    correct = 0  
    total = 0  
  
    with torch.no_grad():  
        for inputs, targets in test_loader:  
            inputs, targets = inputs.to(device), targets.to(device)  
            outputs = model(inputs)  
            _, predicted = outputs.max(1)  
            total += targets.size(0)  
            correct += predicted.eq(targets).sum().item()  
  
    return 100. * correct / total
```

Putting It All Together

```
# Training loop
num_epochs = 10
for epoch in range(num_epochs):
    print(f'\nEpoch: {epoch+1}')
    train_epoch(model, train_loader, loss_function, optimizer, device)
    accuracy = evaluate(model, test_loader, device)
    print(f'Test Accuracy:{accuracy:.2f}%')
```

Training

Epoch: 1

Loss: 0.627	Acc: 81.97%
Loss: 0.336	Acc: 90.28%
Loss: 0.251	Acc: 92.44%
Loss: 0.235	Acc: 93.31%
Loss: 0.213	Acc: 93.64%
Loss: 0.190	Acc: 94.66%
Loss: 0.167	Acc: 95.12%
Loss: 0.159	Acc: 95.34%
Loss: 0.155	Acc: 95.52%
Test Accuracy: 95.48%	

Epoch: 2

Loss: 0.115	Acc: 96.78%
Loss: 0.112	Acc: 96.66%
Loss: 0.122	Acc: 96.39%
Loss: 0.116	Acc: 96.59%
Loss: 0.105	Acc: 96.91%
Loss: 0.104	Acc: 96.78%
Loss: 0.116	Acc: 96.48%
Loss: 0.105	Acc: 96.64%
Loss: 0.095	Acc: 97.05%
Test Accuracy: 97.02%	

Epoch: 10

Loss: 0.013	Acc: 99.69%
Loss: 0.015	Acc: 99.50%
Loss: 0.017	Acc: 99.45%
Loss: 0.019	Acc: 99.41%
Loss: 0.026	Acc: 99.08%
Loss: 0.022	Acc: 99.11%
Loss: 0.028	Acc: 99.00%
Loss: 0.028	Acc: 99.02%
Loss: 0.023	Acc: 99.22%
Test Accuracy: 97.58%	