Introduction to

Logistics

HW 1 is out! (Due by November 8th)

Tutorial Supplementary Notebooks are in Moodle

Deep Learning Frameworks

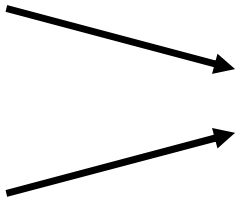
NYU

Caffe

UC Berkeley

theano

Montreal



Facebook

Google

Amazon

Microsoft

Eclipse (IBM)

Google

 Slide Idea: Justin Johnson

Tensors

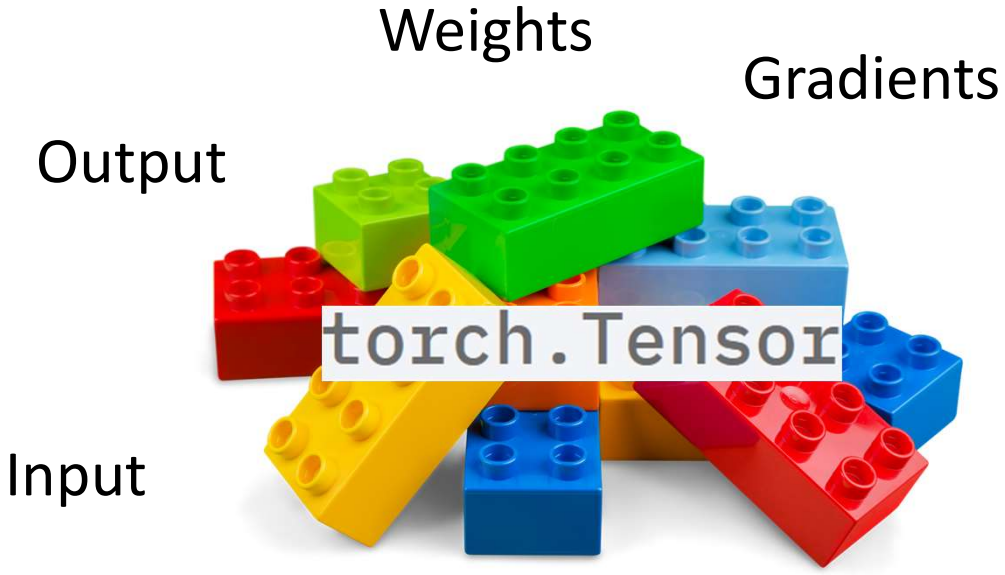
(Almost) Just like NumPy!

```
# creation from data
data = [[1, 2],[3, 4]]
x_data = torch.tensor(data)
```

```
tensor([[1, 2],
        [3, 4]])
```

```
# creation from numpy array
np_array = np.array(data)
x_np = torch.from_numpy(np_array)
```

```
tensor([[1, 2],
        [3, 4]])
```



Tensor Attributes

$$x\_data = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

shape

```
x_data.shape
```

```
torch.Size([2, 2])
```

dtype

```
x_data.dtype
```

```
torch.int64
```

device

```
x_data.device
```

```
device(type='cpu')
```

Managing Device

Do we have GPU resources available?

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

```
True
```

How many GPU resources are available?

```
torch.cuda.device_count()
```

```
1
```

Moving a Tensor from one device to another.

```
print(x_data.device)
x_gpu = x_data.to('cuda')
print(x_gpu.device)
```

```
cpu
cuda:0
```


Based on Pytorch tutorials

Tensor Operations

```
# initialization
a = torch.tensor([[1, 2, 3],[4, 5, 6],[7, 8, 9]])
b = torch.eye(3)

print(f"a: {a}")
print(f"b: {b}")
```

```
a: tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
b: tensor([[1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

Basic Operations

Just like NumPy!

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
a + b
```

```
tensor([[ 2.,  2.,  3.],
        [ 4.,  6.,  6.],
        [ 7.,  8., 10.]])
```

```
a - b
```

```
tensor([[0., 2., 3.],
        [4., 4., 6.],
        [7., 8., 8.]])
```

```
a * b
```

```
tensor([[1., 0., 0.],
        [0., 5., 0.],
        [0., 0., 9.]])
```

```
a / b
```

```
tensor([[1., inf, inf],
        [inf, 5., inf],
        [inf, inf, 9.]])
```

```
a ** b
```

```
tensor([[1., 1., 1.],
        [1., 5., 1.],
        [1., 1., 9.]])
```

```
a @ b
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
```

In-place Operations

Just like NumPy!

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

```
a + b
a
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
```

```
a += b
a
```

```
tensor([[ 2.,  2.,  3.],
        [ 4.,  6.,  6.],
        [ 7.,  8., 10.]])
```

```
a -= b
a
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
```

```
a.clamp(2, 5)
a
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
```

```
a.clamp_(2, 5)
a
```

```
tensor([[2., 2., 3.],
        [4., 5., 5.],
        [5., 5., 5.]])
```

Tensor Operations

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Indexing and slicing – just like NumPy!

Indexing

```
a[0,0]
```

```
tensor(1.)
```

```
a[1,2]
```

```
tensor(6.)
```

Slicing

```
a[0,:] row
```

```
tensor([1., 2., 3.])
```

```
a[0,:].shape
```

```
torch.Size([3])
```

```
a[:,0] column
```

```
tensor([1., 4., 7.])
```

```
a[:,0].shape
```

```
torch.Size([3])
```

Tensor Operations

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Concatenation, splitting, stacking, etc. – just like NumPy!

Concatenation

```
torch.cat((a, b))
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.],
        [1., 0., 0.],
        [0., 1., 0.],
        [0., 0., 1.]])
```

```
torch.cat((a, b)).shape
```

```
torch.Size([6, 3])
```

```
torch.cat((a, b), dim=1)
```

```
tensor([[1., 2., 3., 1., 0., 0.],
        [4., 5., 6., 0., 1., 0.],
        [7., 8., 9., 0., 0., 1.]])
```

```
torch.cat((a, b), dim=1).shape
```

```
torch.Size([3, 6])
```

dim → axis

Pytorch → Numpy

Tensor Operations

$$a = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \quad b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

reshape – just like NumPy!

```
a.shape
```

```
torch.Size([3, 3])
```

```
a.reshape((1,9))
```

```
tensor([[1., 2., 3., 4., 5., 6., 7., 8., 9.]])
```

```
a.reshape((1,9)).shape
```

```
torch.Size([1, 9])
```

squeeze & unsqueeze

```
a_unsqueeze = a.unsqueeze(dim=0)
```

```
a_unsqueeze.shape
```

```
torch.Size([1, 3, 3])
```

```
a_unsqueeze.squeeze_()
```

```
tensor([[1., 2., 3.],
        [4., 5., 6.],
        [7., 8., 9.]])
```

```
a_unsqueeze.shape
```

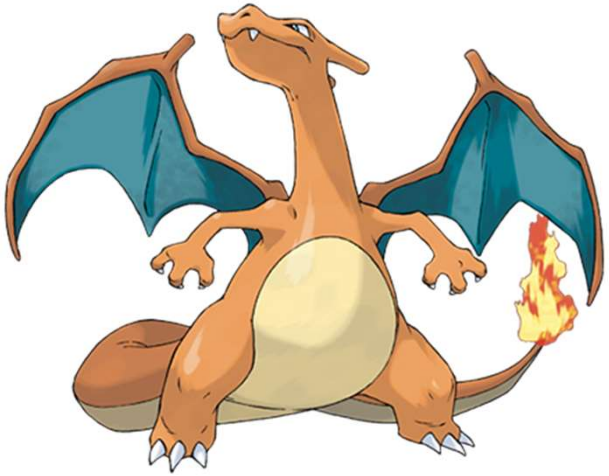
```
torch.Size([3, 3])
```

DON'T Do It Yourself!

- Use built-in vectorized operations instead of implementing them yourself!
- Use GPU resources when possible.

Your Implementation

Pytorch's Implementation



Pytorch's Implementation on GPU

Case Study: Matrix Multiplication

Case 1: Nested for loops

```
res = torch.zeros((size, size))
for i in range(size):
  for j
    row
    col = y[:,
    res[i, j] = torch.sum(
```

18.8 seconds

Case 2: Single for loop

```
res =
for i
    res[i, :]
```

132 milliseconds

Case 3: Vectorized Operation

```
torc
```

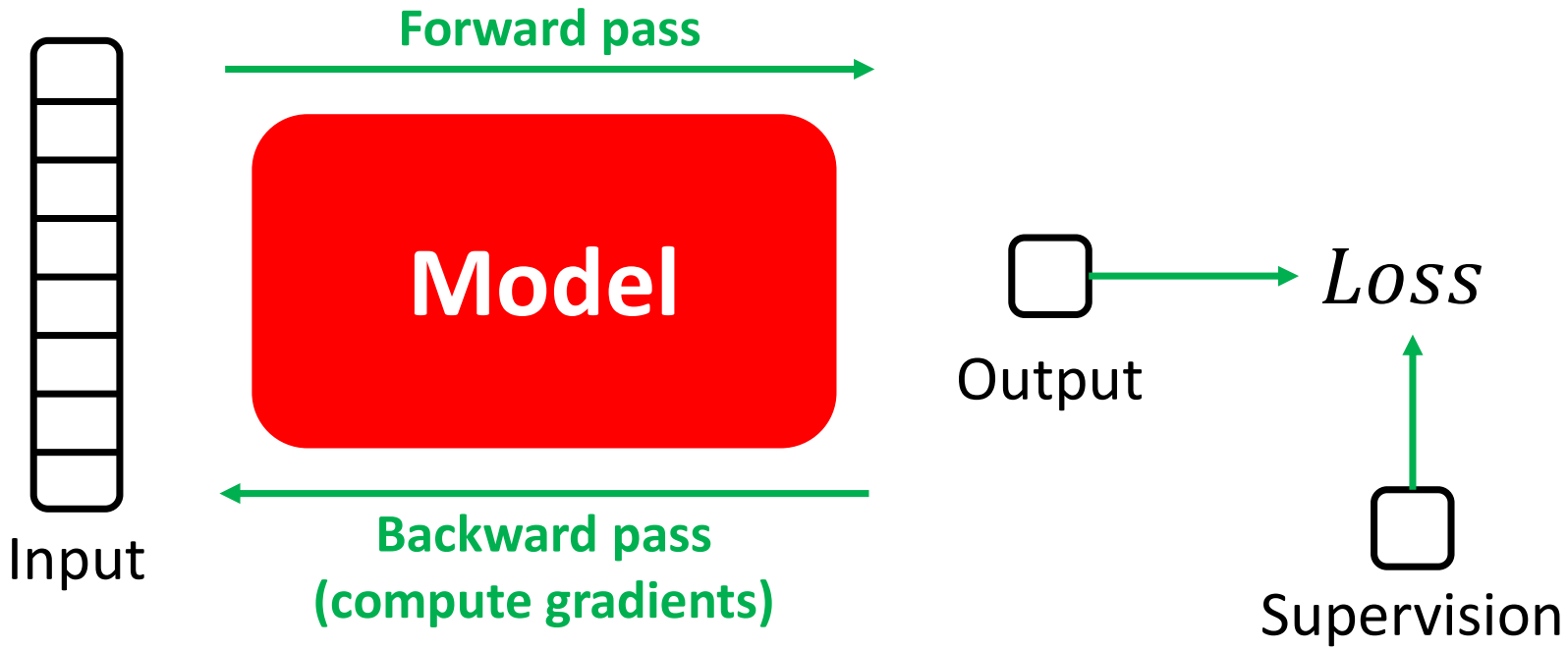
25.8 milliseconds

Case 4: using a GPU

```
x_g
y_g
torch.mm(x_g
```

0.57 milliseconds

Backpropagation



Building a Neural Network

Building a Neural Network

```
INPUT_SIZE = 224
C = 10 # num classes

class MLP(nn.Module):

  def __init__(self):
    super(MLP, self).__init__()
    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(INPUT_SIZE*INPUT_SIZE, 256)
    self.relu1 = nn.ReLU()
    self.fc2 = nn.Linear(256, 512)
    self.relu2 = nn.ReLU()
    self.fc3 = nn.Linear(512, C)

  def forward(self, x):
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.relu1(x)
    x = self.fc2(x)
    x = self.relu2(x)
    x = self.fc3(x)
    return x
```

Initialize

```
B = 10 # batch size

input = torch.randn(B, 1, INPUT_SIZE, INPUT_SIZE)
model = MLP()
```

input shape: $(B, 1, H, W)$

output shape: (B, C)

Run with CPU

```
output = model(input)
```

7.96 milliseconds

Run with GPU

```
gpu_model = model.to('cuda')
gpu_input = input.to('cuda')

gpu_output = gpu_model(gpu_input)
```

0.58 milliseconds

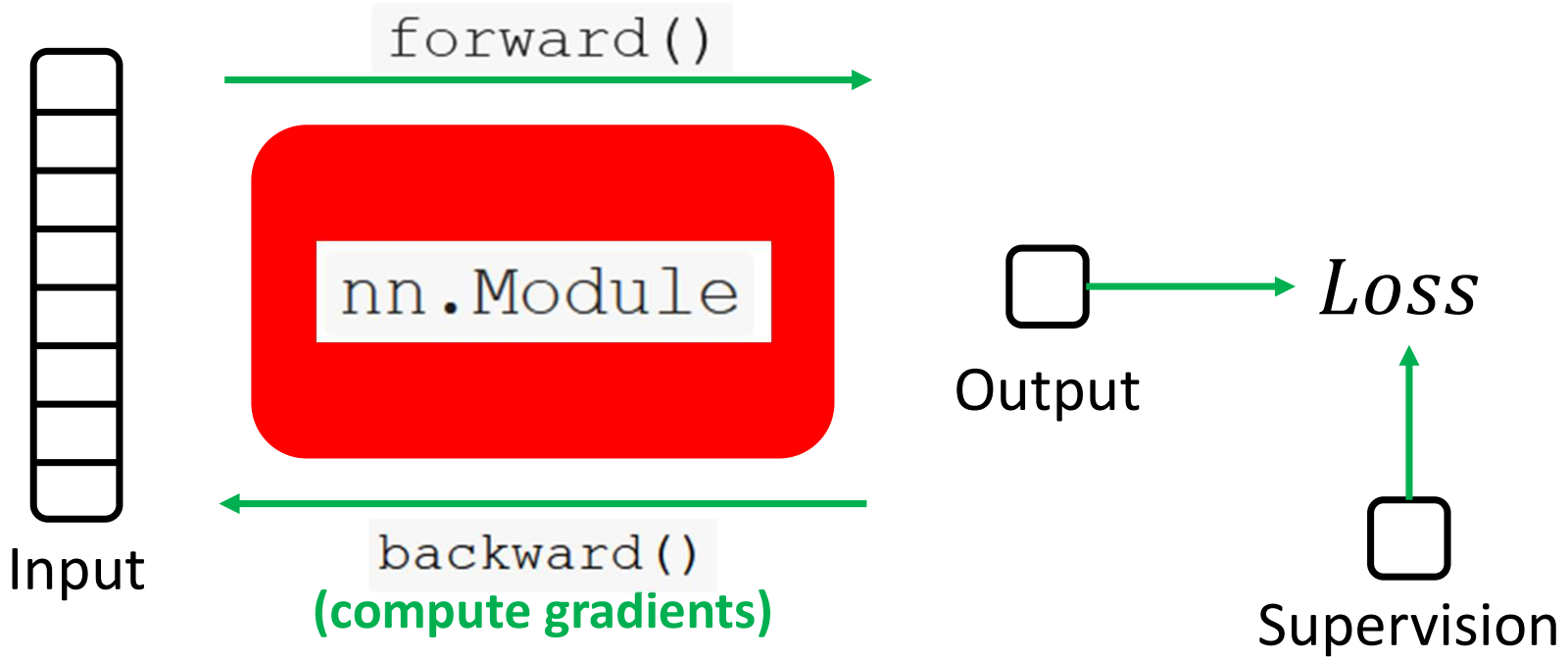
Building a Neural Network

```
INPUT_SIZE = 224
C = 10 # num classes

class MLP(nn.Module):

  def __init__(self):
    super(MLP, self).__init__()
    self.flatten = nn.Flatten()
    self.fc1 = nn.Linear(INPUT_SIZE*INPUT_SIZE, 256)
    self.relu1 = nn.ReLU()
    self.fc2 = nn.Linear(256, 512)
    self.relu2 = nn.ReLU()
    self.fc3 = nn.Linear(512, C)

  def forward(self, x):
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.relu1(x)
    x = self.fc2(x)
    x = self.relu2(x)
    x = self.fc3(x)
    return x
```

Using Sequential

```
seq_model = nn.Sequential(
    nn.Flatten(),
    nn.Linear(INPUT_SIZE*INPUT_SIZE, 256),
    nn.ReLU(),
    nn.Linear(256, 512),
    nn.ReLU(),
    nn.Linear(512, 10),
)

output = seq_model(input)
```

Backpropagation



Tensor Attributes - Autograd

shape

dtype

device

grad

grad_fn

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
print(x.grad)
print(x.grad_fn)
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
None
None
```

Autograd - backward

```
x = torch.ones(2, 2, requires_grad=True)
print(x)
print(x.grad)
print(x.grad_fn)
```

```
tensor([[1., 1.],
        [1., 1.]], requires_grad=True)
None
None
```

```
y = x + 2
print(y)
```

```
tensor([[3., 3.],
        [3., 3.]], grad_fn=<AddBackward0>)
```

```
z = y * y * 3
out = z.mean()
print(z)
print(out)
```

```
tensor([[27., 27.],
        [27., 27.]], grad_fn=<MulBackward0>)
tensor(27., grad_fn=<MeanBackward0>)
```

```
out.backward()
print(x.grad)
```

```
tensor([[4.5000, 4.5000],
        [4.5000, 4.5000]])
```

Autograd

Disable Gradients

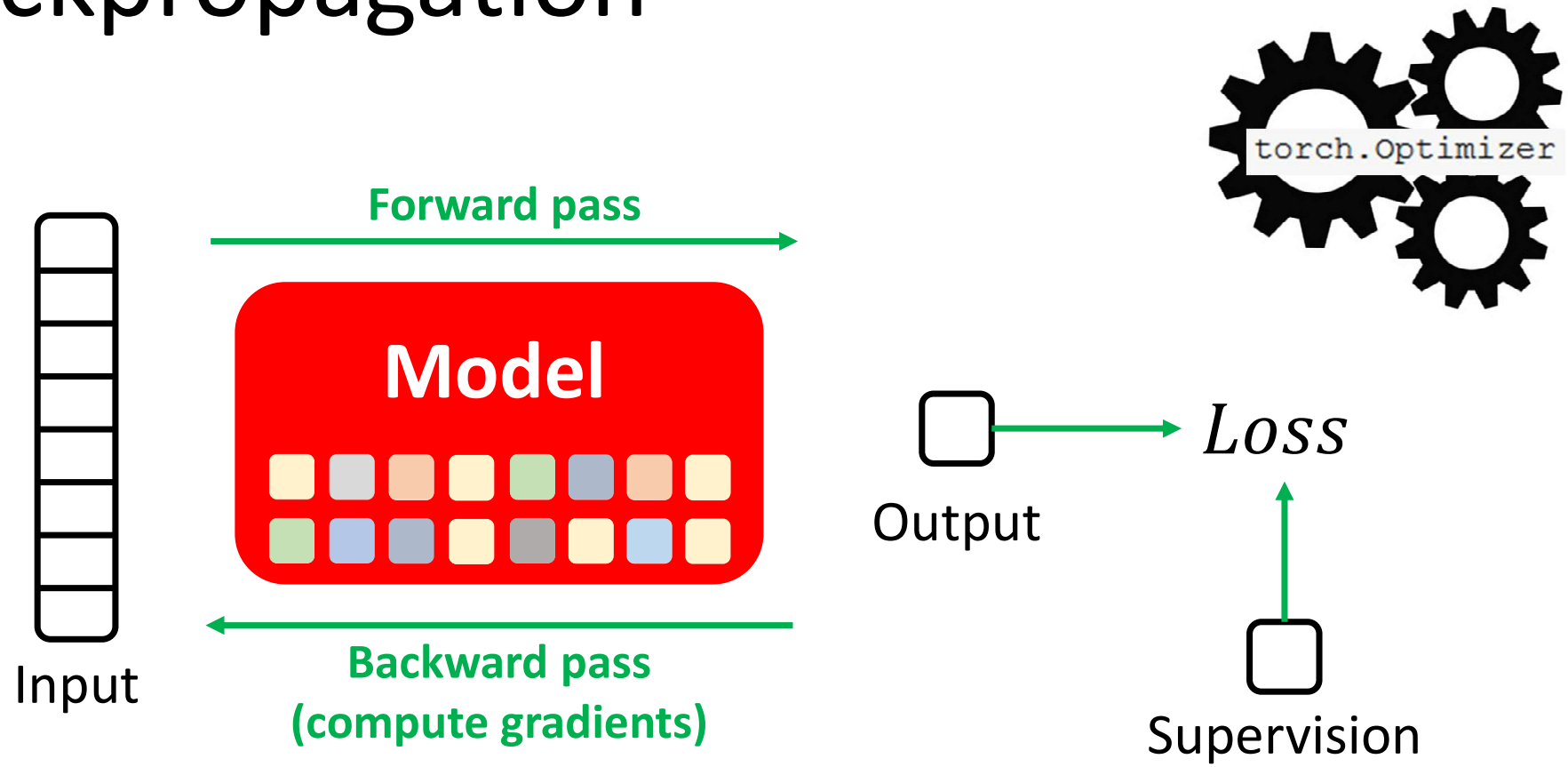
```
print(x.requires_grad)
print((x ** 2).requires_grad)

with torch.no_grad():
    print((x ** 2).requires_grad)
```

```
True
True
False
```

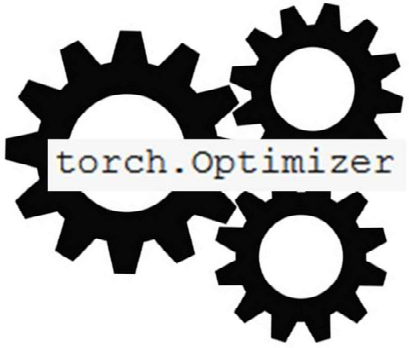
Back to NumPy

```
np_out = out.detach().cpu().numpy()
```

Backpropagation



Optimizers



Create an SGD optimizer

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

Reset Gradients

```
optimizer.zero_grad()
```

Perform an optimization step

```
optimizer.step()
```

Putting it All Together – Fashion MNIST Classification

Putting it All Together

1. Hyperparameters
2. Handling Data
3. Model, Loss, Optimizer
4. Training and Inference
5. Visualization
6. Overall Training Process

1. Hyperparameters

```
# configurations
LR = 1e-3  # learning rate
B_SQRT = 8
B = B_SQRT**2  # batch size
EPOCHS = 100  # num epochs
INPUT_SIZE = 28  # input size

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

 Based on Pytorch tutorials

2. Handling Data

```
# handling data
training_data = datasets.FashionMNIST(
  root="data",
  train=True,
  download=True,
  transform=ToTensor()
)

val_data = datasets.FashionMNIST(
  root="data",
  train=False,
  download=True,
  transform=ToTensor()
)
```

Downloading datasets

Setting dataset manager

- Automatic batching
- Customized data loading order
- …

3. Model, Loss, Optimizer

```
# define model
model = nn.Sequential(
  nn.Flatten(),
  nn.Linear(INPUT_SIZE*INPUT_SIZE, 512),
  nn.ReLU(),
  nn.Linear(512, 512),
  nn.ReLU(),
  nn.Linear(512, 10),
)

# move to correct device
model.to(device)
```

```
# define loss
criterion = nn.CrossEntropyLoss()
```

```
# define optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=LR)
```

4. Training and Inference

Training Loop

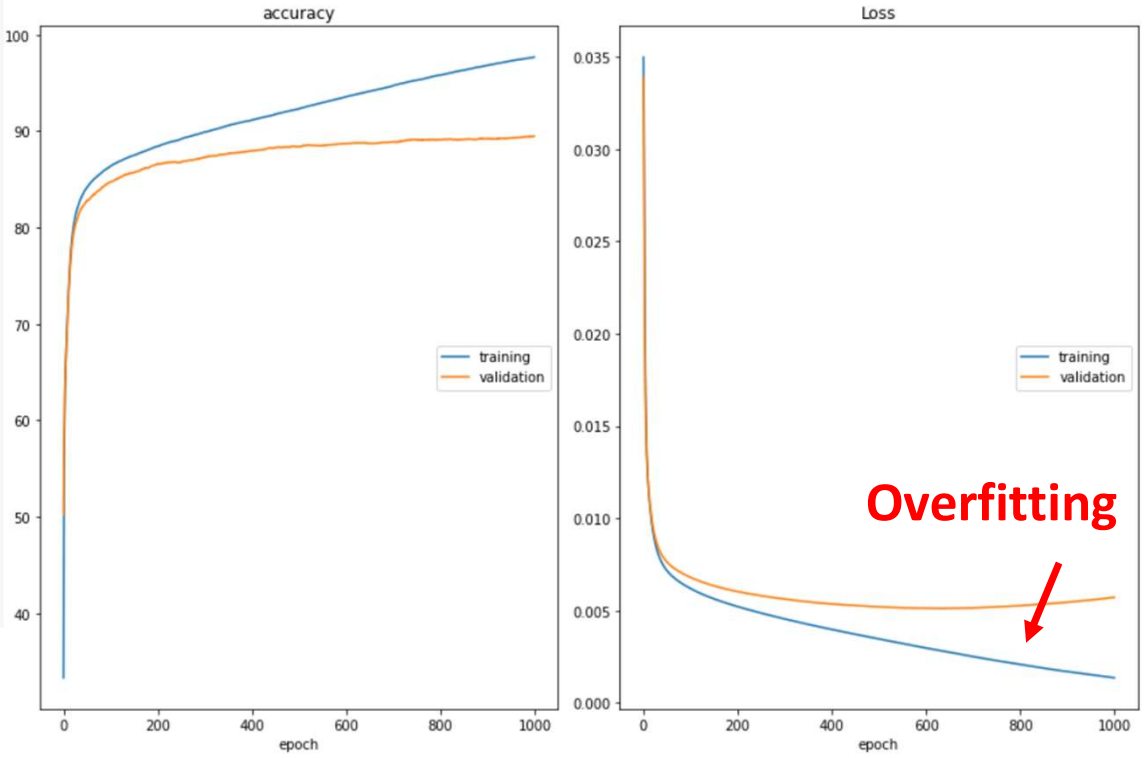
```
# iterate through all batches
for batch, (X, y) in enumerate(dataloader):
  # move data to device
  X, y = X.to(device), y.to(device)
  # forward pass
  pred = model(X)
  loss = criterion(pred, y)
  # new gradients per batch
  optimizer.zero_grad()
  # backward pass
  loss.backward()
  # update
  optimizer.step()
```

Inference Loop

```
# disregard gradients when not training
with torch.no_grad():
  # iterate through all batches
  for X, y in dataloader:
    # move data to device
    X, y = X.to(device), y.to(device)
    # forward pass
    pred = model(X)
```

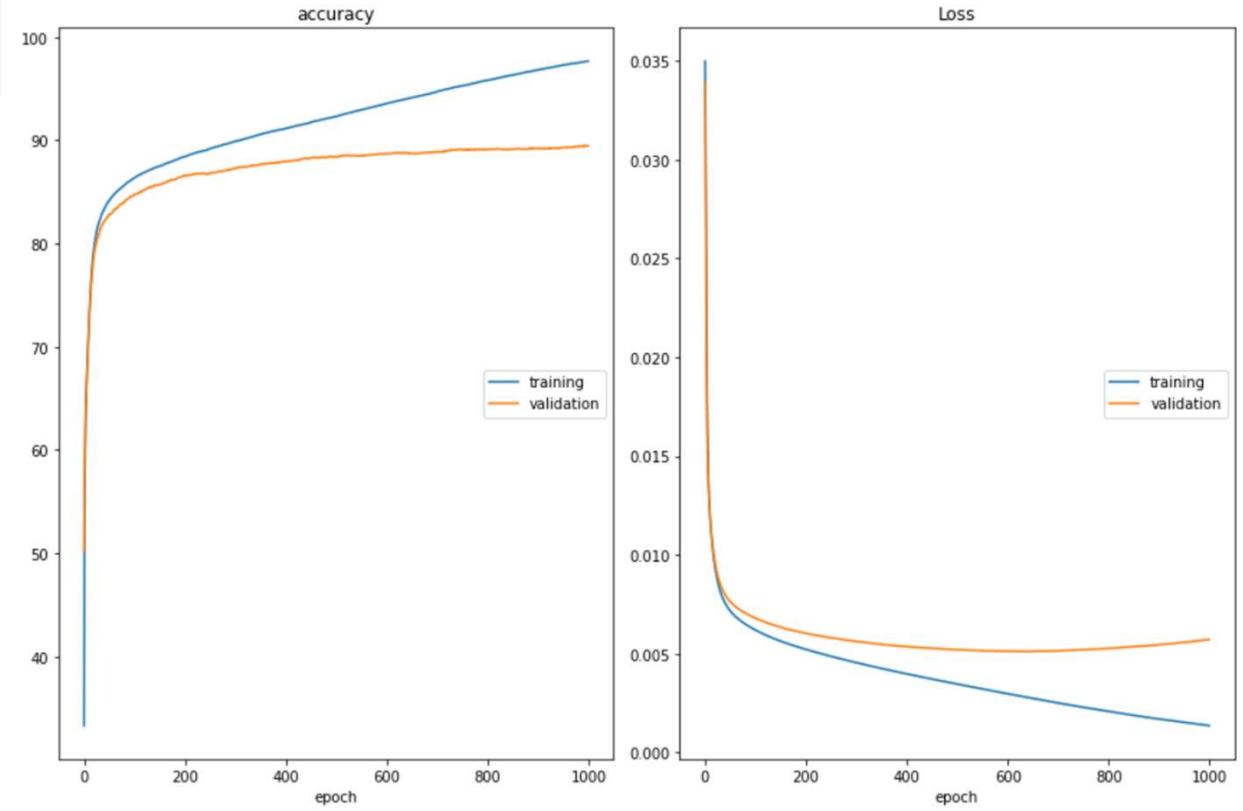
5. Visualizer

```
# arrange visualization using liveplotloss library
class Visualizer:
  def __init__(self):
    self.liveloss = PlotLosses()

  def update(self, train_res, val_res):
    train_epoch_loss, train_epoch_accuracy = train_res
    val_epoch_loss, val_epoch_accuracy = val_res
    logs = {}
    logs[f'loss'] = train_epoch_loss
    logs[f'accuracy'] = train_epoch_accuracy
    logs[f'val_loss'] = val_epoch_loss
    logs[f'val_accuracy'] = val_epoch_accuracy
    self.liveloss.update(logs)
    self.liveloss.send()

visualizer = Visualizer()
```



6. Overall Training Process

```
for t in range(EPOCHS):
  train_res = train_loop(train_dataloader, model, criterion, optimizer)
  val_res = inference_loop(val_dataloader, model, criterion)
  visualizer.update(train_res, val_res)

print("Done!")
```



 Based on Pytorch tutorials

To Be Continued…

Next Week: CNNs