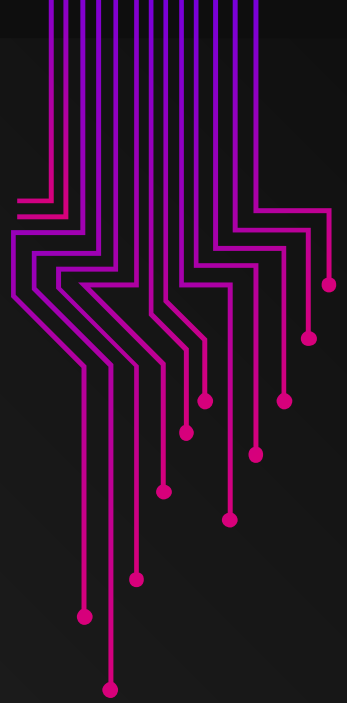
PyTorch

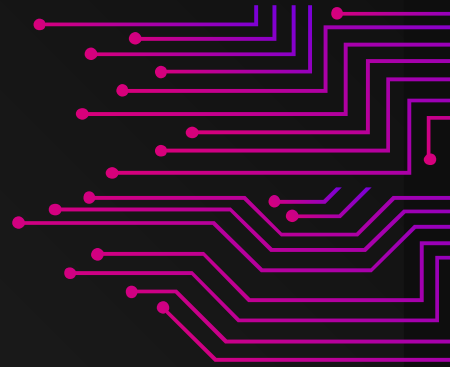
Rebekah McLatcher, Ella Wileman, &
Eleanor Madderra

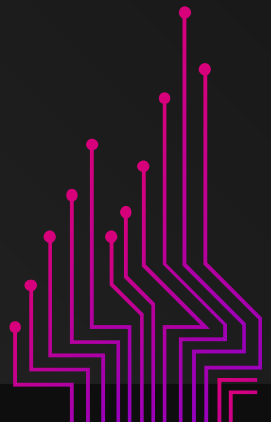


PyTorch at a Glance

PyTorch is an open source framework often used for building deep learning models

- Developed by Facebook's AI Research Lab (FAIR)
- Based on Torch
- First launched in 2016
- Merged with Caffe2 in 2018
- Continuously evolving through community effort and innovation



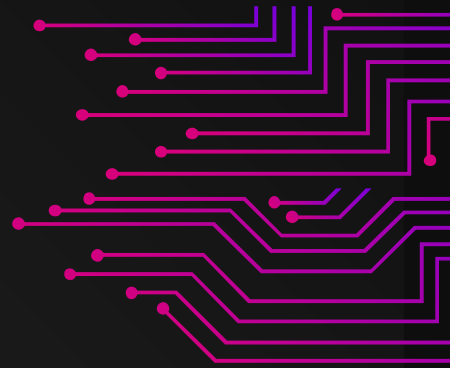


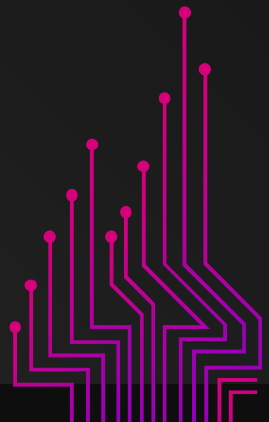
“Simple inference, dynamic computational graphs, back-end support for CPUs and GPUs, and first-class Python integration”

-https://ai.meta.com/blog/pytorch-builds-the-future-of-ai-and-machine-learning-at-facebook/

Why choose PyTorch?

- Flexibility during model building
  - Can modify models during runtime
- Fairly intuitive
  - Simple design and is easy to learn
- Fast training times
  - The autograd differentiation package that PyTorch uses for training means it has some of the fastest training times among machine learning frameworks
- Research
  - Currently used in academic research settings
  - Quick iterations on experimental models





Uses & Characteristics

PyTorch Capabilities and Strengths

Computer Vision

Packages like TorchVision have frameworks for building computer vision systems

Reverse-mode auto differentiation

Improves efficiency in optimization of deep learning models

Tensors

Core data structure of PyTorch, enabling efficient computation and optimization

Pythonic

Adheres to the principles of Python coding, making it simple to use and integrate

Computer Vision

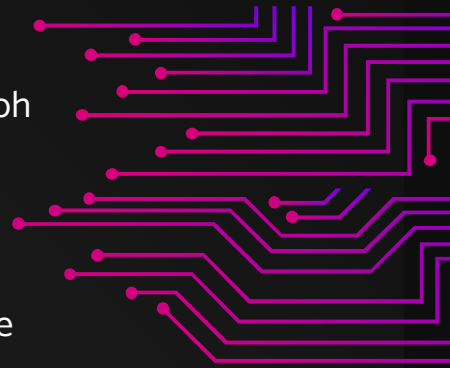
PyTorch allows developers to easily build deep learning models with flexibility to accommodate tasks including segmentation, image classification, and object detection

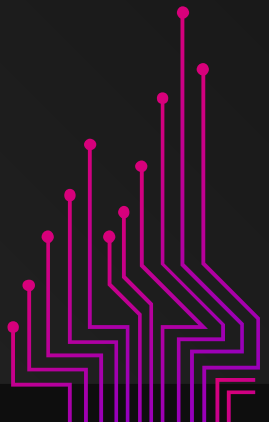
TorchVision is a PyTorch Package that offers:

- Pre-trained models: Ready-to-use models like ResNet and EfficientNet that can be used as a strong baseline models or starting points for fine-tuning
- Datasets: Provides access to popular databases like ImageNet, COCO, and MNIST that can be used for either training or pre-training before fine-tuning
- Tools: Built- in functionalities for image preprocessing such as cropping, resizing, and normalization that improve training process

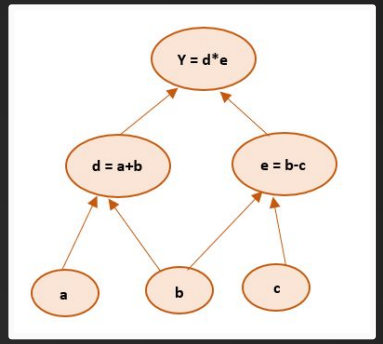
Reverse-mode auto differentiation

- Efficiently calculates the gradient by traversing through the computational graph backwards to compute derivatives
- Helpful in high dimensional spaces and deep learning where the model input often has many inputs and few outputs
- Relies on recording the function outputs on the forward pass and the use of the chain rule and the dynamic computational graph to propagate backwards from the output
- Autograd is the subpackage for automatic differentiation





Computational Graphs



- Directed acyclic graphs store and visualize the operations applied to the tensors - nodes convey calculations (Nodes = Tensors) and edges represent flow (functions of Tensors)
- Crucial tool in backwards propagation as it tracks how outputs depend on inputs
- PyTorch utilizes Dynamic Computational Trees to calculate gradients of loss functions
    - Static tree requires the full computational graph to be defined before execution, while dynamic computation trees are adaptable, changing during runtime
    - Static trees are symbolic representations of operations making debugging more difficult

Why Reverse AD vs. Just Tensors?

- **Tensorflow:** manually implementing the backward passes through the network
  - Not difficult to implement for lower dimensional data, becomes very computationally expensive (near impossible) with higher dimensions
- **Reverse AD:** automate the computation of backward passes in neural network

Example Use Case: fit a third order polynomial to sine function

(After Tensors created to hold inputs/outputs, and random Tensors for weights) –>

https://pytorch.org/tutorials/beginner/pytorch_with_examples.html

```
learning_rate = 1e-6
for t in range(2000):
    # Forward pass: compute predicted y using operations on Tensors.
    y_pred = a + b * x + c * x ** 2 + d * x ** 3

    # Compute and print loss using operations on Tensors.
    # Now loss is a Tensor of shape (1,)
    # loss.item() gets the scalar value held in the loss.
    loss = (y_pred - y).pow(2).sum()
    if t % 100 == 99:
        print(t, loss.item())

    # Use autograd to compute the backward pass. This call will compute the
    # gradient of loss with respect to all Tensors with requires_grad=True.
    # After this call a.grad, b.grad. c.grad and d.grad will be Tensors holding
    # the gradient of the loss with respect to a, b, c, d respectively.
    loss.backward()

    # Manually update weights using gradient descent. Wrap in torch.no_grad()
    # because weights have requires_grad=True, but we don't need to track this
    # in autograd.
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad

        # Manually zero the gradients after updating weights
        a.grad = None
        b.grad = None
        c.grad = None
        d.grad = None

print(f'Result: y = {a.item()} + {b.item()} x + {c.item()} x^2 + {d.item()} x^3')
```

MSE for loss

Related gradients now stored in computational graph

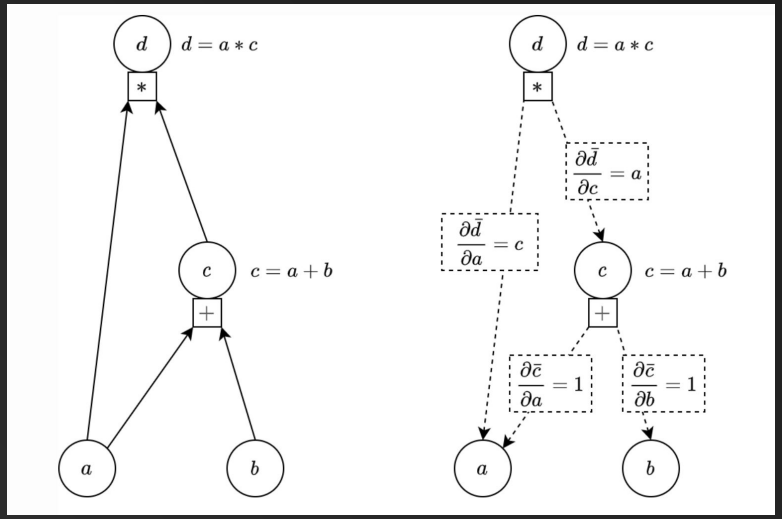
Because Tensor now doesn't track gradients

Reverse-mode AD Example

- Gradient of d?

```
a = 4
b = 3
c = a + b   # = 4 + 3 = 7
d = a * c   # = 4 * 7 = 28
```

Using local derivatives calculated in graph:



$$\frac{\partial d}{\partial a} = \frac{\partial \bar{d}}{\partial a} + \frac{\partial \bar{d}}{\partial c} * \frac{\partial \bar{c}}{\partial a}$$

$$\frac{\partial d}{\partial a} = c + a * 1$$

$$\frac{\partial d}{\partial a} = a + b + a$$

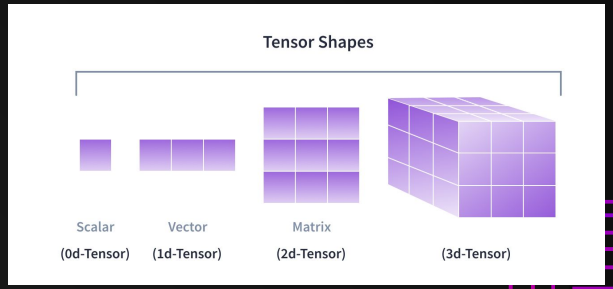
$$\frac{\partial d}{\partial a} = 2a + b$$

$$\frac{\partial d}{\partial a} = 11$$

Go over graph and find paths from d to a, multiply along path and add along.

https://sidsite.com/posts/autodiff/

Tensors



Fundamental data structure in PyTorch, used to store and manipulate multidimensional data

- Similar to NumPy arrays, but with the added ability to run on GPUs, significantly improving computational speed (50x or more)
  - PyTorch tensors and NumPy arrays can converted back and forth
- Support multiple data types and efficient for operations like addition, multiplication, reshaping, indexing, etc. over entire data structure
- Essential for deep learning models, serving as
  - The foundation of neural network architectures
  - Format of training data and model parameters
  - Structure of gradient computation during backpropagation

Commonly used in processing image and audio data, which are commonly converted in tensors for deep learning tasks

https://www.dataquest.io/blog/tutorial-introduction-to-tensorflow/

Tensors

https://www.learnpytorch.io/pytorch_cheatsheet/

In [4]:
```
# Create a single number tensor (scalar)
scalar = torch.tensor(7)
```

In [5]:
```
# Create a random tensor
random_tensor = torch.rand(size=(3, 4)) # this will create a tensor of s
```

 # this will create a tensor of size 3x4 but you can manipulate the shape however you want

In [6]:
```
# Multiply two random tensors
random_tensor_1 = torch.rand(size=(3, 4))
random_tensor_2 = torch.rand(size=(3, 4))
random_tensor_3 = random_tensor_1 * random_tensor_2 # PyTorch has suppo
```

 # PyTorch has support for most math operators in Python (+, *, -, /)

Tensors

https://www.learnpytorch.io/pytorch_cheatsheet/

- Computing on tensors typically happen much faster on GPUs than CPUs
- Generally: NVIDIA GPU ("cuda") > MPS device ("mps") > CPU ("cpu")
- It is advised to setup device-agnostic code at the beginning of your workflow

In [11]:
```
# Setup device-agnostic code
if torch.cuda.is_available():
    device = "cuda" # NVIDIA GPU
elif torch.backends.mps.is_available():
    device = "mps" # Apple GPU
else:
    device = "cpu" # Defaults to CPU if NVIDIA GPU/Apple GPU aren't avai

print(f"Using device: {device}")
```

```
Using device: mps
```

Tensors

https://www.learnpytorch.io/pytorch_cheatsheet/

- You can move objects (models and tensors) in PyTorch to different devices via the .to("device_name") method

In [12]:
```
# Create a tensor
x = torch.tensor([1, 2, 3])
print(x.device) # defaults to CPU

# Send tensor to target device
x = x.to(device)
print(x.device)
```

```
cpu
mps:0
```

Pythonic

- PyTorch adheres to the standards and principles of python programming like readability and simplicity and follows native python constructs like data structures, data types, and programming constructs (goes beyond just using syntax that works, it is written the way it is meant to be)
- Works naturally with Python data structures and control flow so it can be seamlessly integrated into a Python ecosystem, and easily understood by a team of programmers by upholding “best practices” and utilizing the most straightforward solutions
- Can also be used in alongside other popular Python libraries including NumPy, SciPy, Pandas, etc.

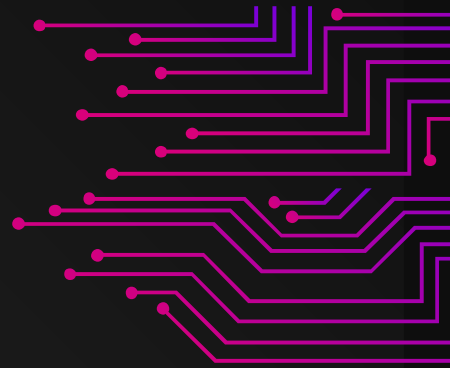
https://builtin.com/data-science/pythonic#:~:text=Pythonic%20describes%20code%20that%20doesn,to%20optimize%20your%20Python%20code.

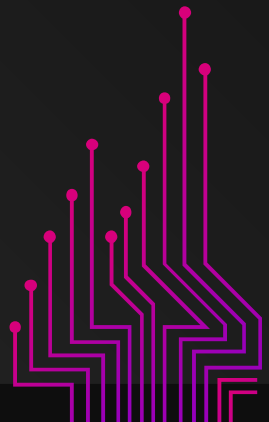
Pythonic Rules (The Zen of Python)

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
*Although never is often better than *right* now.*
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!

Code that upholds these principles is considered "pythonic"

PyTorch is meant to be pythonic in how its users interact with its syntax.



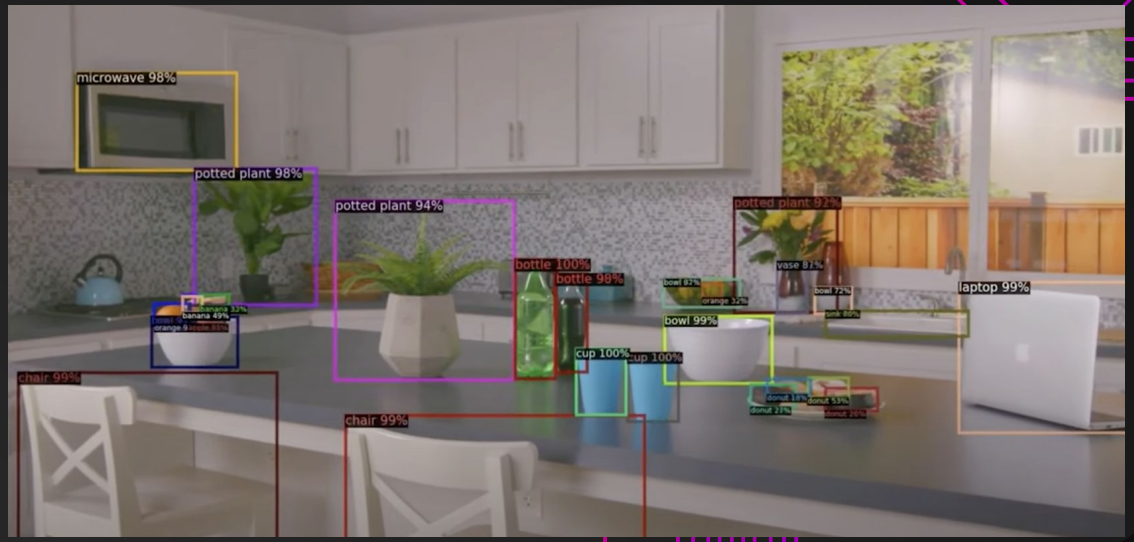


https://peps.python.org/pep-0020/#the-zen-of-python

Real World Examples

Detectron2

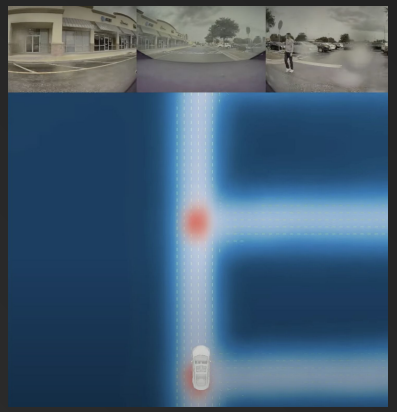
- An open source framework developed by Facebook AI Research
- Identifies objects in images and determines each objects boundaries
- Can detect human body key points
- Categorize each pixel in an image into semantic classes which can be used to understand scene context

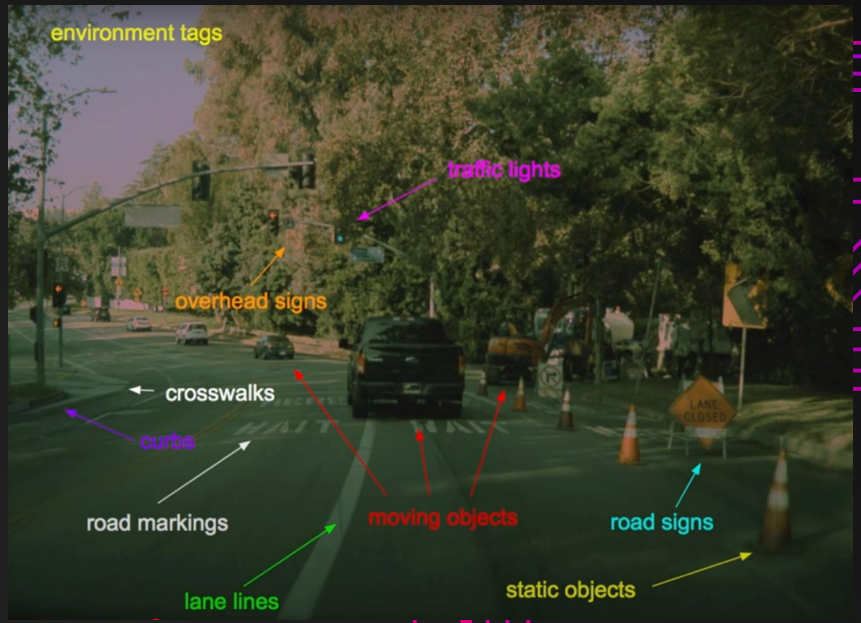


https://www.youtube.com/watch?v=oBklltKXtDE&ab_channel=PyTorch

Tesla Autopilot

- Tesla uses PyTorch to train networks to complete tasks for their computer vision applications
  - Including object detection and depth modeling

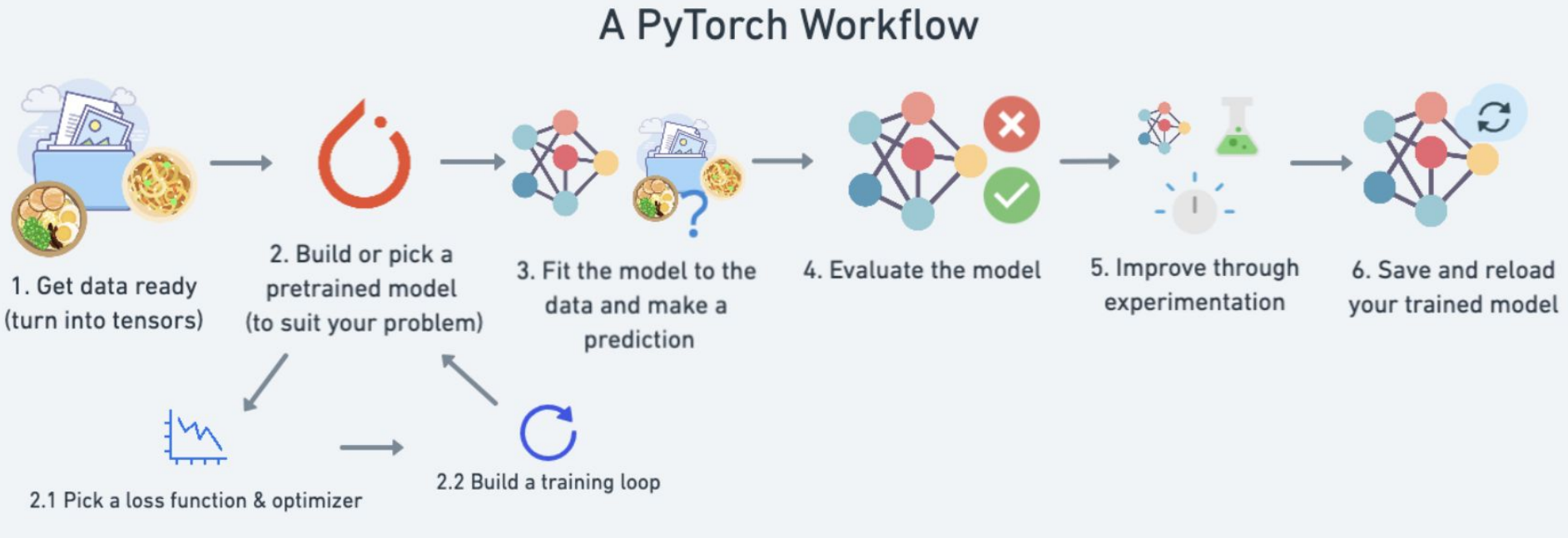




PyTorch In Action

Example Workflow

https://www.learnpytorch.io/pytorch_cheatsheet/



Create Data

https://www.learnpytorch.io/pytorch_cheatsheet/

- X and y are tensors
- X represents a series of values from 0 to 1 with a step size of 0.02
- y represents the target output for each of those values
- Create train/test split

Create data

In [34]:
```
# Create *known* parameters
weight = 0.7
bias = 0.3

# Create data
start = 0
end = 1
step = 0.02
X = torch.arange(start, end, step).unsqueeze(dim=1) # data
y = weight * X + bias # labels (want model to learn from data to predict these)

X[:10], y[:10]
```

Out[34]:
```
(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]]))
```

In [35]:
```
# Create train/test split
train_split = int(0.8 * len(X)) # 80% of data used for training set, 20% for testing
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)
```

Out[35]:
```
(40, 40, 10, 10)
```

https://www.learnpytorch.io/pytorch_cheatsheet/

Build a Model

- Create a model by stacking layers sequentially using Sequential from PyTorch's torch.nn module
- Set up loss and optimizer
  - L1Loss() is one of PyTorch's built-in loss functions which is often robust to outliers
  - Optimize using SGD - an algorithm that updates model parameters using gradients computed from randomly selected mini-batches of data

```
from torch import nn

# Option 2 - use torch.nn.Sequential
model_1 = torch.nn.Sequential(
    nn.Linear(in_features=1,
              out_features=1))

model_1, model_1.state_dict()
```

```
(Sequential(
   (0): Linear(in_features=1, out_features=1, bias=True)
 ),
 OrderedDict([('0.weight', tensor([[0.9905]])), ('0.bias', tensor([0.9053]))]))
```

Setup loss function and optimizer

```
# Create loss function
loss_fn = nn.L1Loss()

# Create optimizer
optimizer = torch.optim.SGD(params=model_1.parameters(), # optimize newly created model's parameters
                            lr=0.01)
```

https://www.learnpytorch.io/pytorch_cheatsheet/

Fit Model to the Data and Make a Prediction

1. Forward pass
- X_train is passed through the model
- Computes the predicted output value (y_pred) for the given input data
2. Calculate loss
- Calculate how far the model's predictions (y_pred) are from the actual values (y_train)
3. Zero gradient
- Gradients are reset to 0 to ensure that gradients from previous iterations do not accumulate

```
torch.manual_seed(42)

# Set the number of epochs
epochs = 1000

# Put data on the available device
# Without this, an error will happen (not all data on target device)
X_train = X_train.to(device)
X_test = X_test.to(device)
y_train = y_train.to(device)
y_test = y_test.to(device)

# Put model on the available device
# With this, an error will happen (the model is not on target device)
model_1 = model_1.to(device)

for epoch in range(epochs):
    ### Training
    model_1.train() # train mode is on by default after construction

    # 1. Forward pass
    y_pred = model_1(X_train)

    # 2. Calculate loss
    loss = loss_fn(y_pred, y_train)

    # 3. Zero grad optimizer
    optimizer.zero_grad()
```

https://www.learnpytorch.io/pytorch_cheatsheet/

Fit Model to the Data and Make a Prediction

4. Backward pass
   - Computes the gradient of the loss with respect to each parameter (weights and biases) in the model
   - These gradients indicate how much each parameter should change to reduce the loss
5. Optimizer step
   - The optimizer uses the gradients found in the backwards pass to adjust and update the parameters of the model
   - Minimize the loss by iteratively adjusting the parameters in the direction that reduces error

```
# Put model on the available device
# With this, an error will happen (the model is not on target device)
model_1 = model_1.to(device)

for epoch in range(epochs):
    ### Training
    model_1.train() # train mode is on by default after construction

    # 1. Forward pass
    y_pred = model_1(X_train)

    # 2. Calculate loss
    loss = loss_fn(y_pred, y_train)

    # 3. Zero grad optimizer
    optimizer.zero_grad()

    # 4. Loss backward
    loss.backward()

    # 5. Step the optimizer
    optimizer.step()
```

https://www.learnpytorch.io/pytorch_cheatsheet/

Evaluate the Model

- Calling eval() on the model changes its behavior for certain layers
  - This ensures the model is based on the learned parameters without additional noise or variability introduced during training
- Make predictions on the test data (test_pred)
- Calculate the test loss
  - Compare model predictions (test_pred) with the actual target values (y_test)

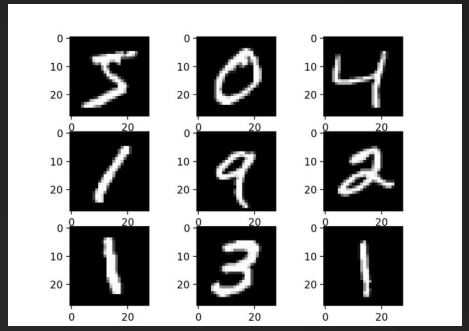
```
### Testing
model_1.eval() # put the model in evaluation mode for testing (inference)
# 1. Forward pass
with torch.inference_mode():
    test_pred = model_1(X_test)

    # 2. Calculate the loss
    test_loss = loss_fn(test_pred, y_test)

if epoch % 100 == 0:
    print(f"Epoch: {epoch} | Train loss: {loss} | Test loss: {test_loss}")
```

```
Epoch: 0 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
Epoch: 100 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
Epoch: 200 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
Epoch: 300 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
Epoch: 400 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
Epoch: 500 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
Epoch: 600 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
Epoch: 700 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
Epoch: 800 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
Epoch: 900 | Train loss: 0.008362661115825176 | Test loss: 0.005596190690994263
```

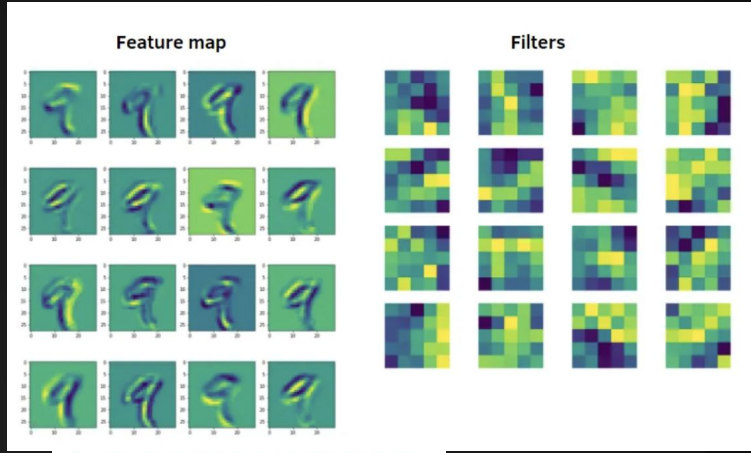
MNSIT Handwritten Image Classification: Pytorch Implementation

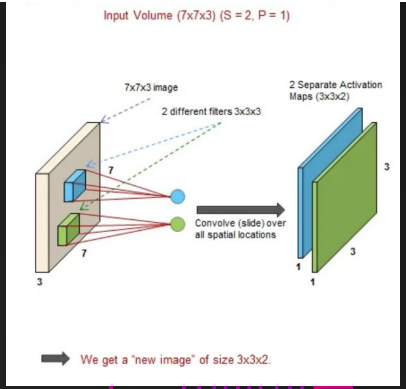


- Falls under computer vision (image classification) use case previously discussed
- Goal: use PyTorch Implementation of Convolutional Neural Network (CNN) for classifying MNSIT handwritten digits
- CNNs useful here for automatic feature selection & recognize edges, textures, patterns, etc.

CNN Architecture (need to type up summary of this info)

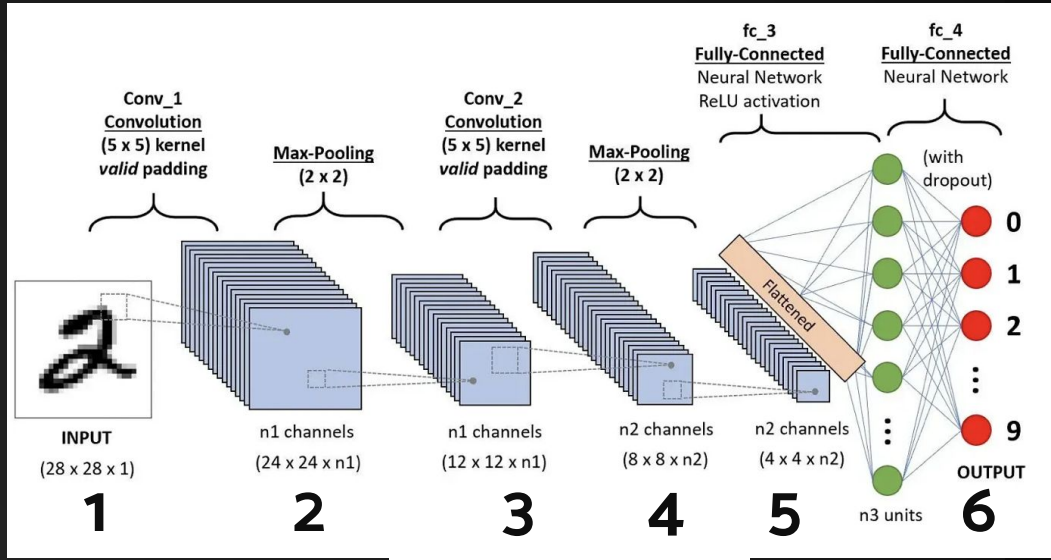
- Filters/Kernels:
  - Small matrices “slide” over number image to perform element-wise multiplication & summation
  - Each designed to detect some specific feature/edge/texture
  - Each yields feature/activation map
- Feature Map:
  - Result of CNN applying filter to image
  - Stack multiple feature maps to form multi-channel output -> input for next layer

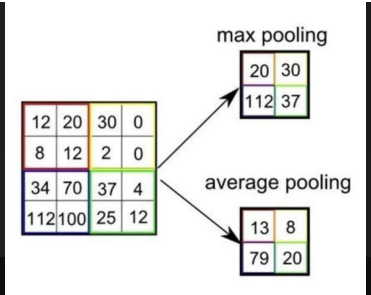




Overall Data Structures & Methodology

1. 1st CNN Layer: applies multiple 5x5 kernels to scan over image, outputs feature maps
2. Max pooling on feature maps to downsample, outputs feature maps of halved dimensions
3. 2nd CNN Layer: more 5x5 filters applied to each feature map, yields new (larger) set of feature maps
4. Apply max pooling again to reduce dimensions
5. Reshape (flatten) set of feature maps to one 1D vector, passed into fully connected layers (now network is learning from global features/patterns instead of local ones)
    a. ReLU applied to weighted sum from layer
    b. Introduces nonlinearity (complex relationships)
6. Output layer uses Softmax to convert 10 logits (1 for each class) -> probabilities
    a. Randomly drop fraction of neurons to avoid overfitting @ each layer
    b. No dependency on one neuron/pathway





Initializing Network

Input channel: 1 for grayscale, 3 for RGB. Each gets feature map.
Output channel: # of feature maps produced by layer (= # filters)

```
class Net(nn.Module):
    #define the layers of the neural network model
    def __init__(self):
        super(Net, self).__init__()
        #creates 2D convolutional layer
        #takes in 1 input channel, outputs 32 channels, kernel size 3x3, stride 1
        self.conv1 = nn.Conv2d(1, 32, 3, 1)
        #creates a second convolutional layer
        #takes in 32 channel output from previous layer
        self.conv2 = nn.Conv2d(32, 64, 3, 1)

        #defines a dropout layer with a 25% chance of zeroing out elements
        #helps prevent overfitting
        self.dropout1 = nn.Dropout(0.25)
        #defines a second dropout layer with a 50% chance of zeroing out elements
        self.dropout2 = nn.Dropout(0.5)
```

```
    #defines a fully connected layer with 9216 input features and 128 output features
    self.fc1 = nn.Linear(9216, 128)
    #defines a second fully connected layer with 128 input features and 10 output
    #(corresponding to the 10 digits 0-9)
    self.fc2 = nn.Linear(128, 10)
```

Dropout layers improve model's ability to generalize from learning redundant representations

Forward Pass Through Network (seen in diagram)

```python
#defines the forward pass of the model
#takes in input x and passes it through the layers of the model
def forward(self, x):
    x = self.conv1(x) #pass input through first convolutional layer
    x = F.relu(x) #apply ReLU activation function
    x = self.conv2(x) #pass output through second convolutional layer
    x = F.relu(x) #apply ReLU activation function
    x = F.max_pool2d(x, 2) #apply max pooling with a 2x2 kernel
    x = self.dropout1(x) #apply dropout to randomly zero out elements
    x = torch.flatten(x, 1) #flatten the tensor so it can be fed into the fully connected layer
    x = self.fc1(x) #pass flattened tensor through first fully connected layer
    x = F.relu(x) #apply ReLU activation function
    x = self.dropout2(x) #apply second dropout for additional regularization
    x = self.fc2(x) #processes the output through the final linear layer to produce raw class scores
    output = F.log_softmax(x, dim=1) #apply log softmax to output
    return output
```

Training + Testing

- Training:
  - Zero out gradient buffers
  - Runs forward and backward passes (built in backward function)
  - Update model params
- Testing:
  - Set model to evaluation mode
  - Disable gradient calcs (memory friendly, no backprop)
  - Loop over test batches
  - Forward pass (predicts probs)
  - Compute loss
  - Calculate predictions and classification metrics

```python
def train(args, model, device, train_loader, optimizer, epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = F.nll_loss(output, target)
        loss.backward()
        optimizer.step()
        if batch_idx % args.log_interval == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.item()))
            if args.dry_run:
                break
```

```python
def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += F.nll_loss(output, target, reduction='sum').item()  # sum up batch loss
            pred = output.argmax(dim=1, keepdim=True)  # get the index of the max log-probability
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)

    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```

Main Method (Overall Steps Taken)

- Command line args specify network architecture, num epochs, etc.
- Normalize images, initialize model & optimizer
- Set learning rate scheduler
  - Decrease learning rate after each epoch
- Call train() and test()

```
    args = parser.parse_args()
    use_cuda = not args.no_cuda and torch.cuda.is_available()
    use_mps = not args.no_mps and torch.backends.mps.is_available()

    torch.manual_seed(args.seed)

    if use_cuda:
        device = torch.device("cuda")
    elif use_mps:
        device = torch.device("mps")
    else:
        device = torch.device("cpu")

    train_kwargs = {'batch_size': args.batch_size}
    test_kwargs = {'batch_size': args.test_batch_size}
    if use_cuda:
        cuda_kwargs = {'num_workers': 1,
                       'pin_memory': True,
                       'shuffle': True}
        train_kwargs.update(cuda_kwargs)
        test_kwargs.update(cuda_kwargs)

    transform=transforms.Compose([
        transforms.ToTensor(),
        transforms.Normalize((0.1307,), (0.3081,))
        ])
dataset1 = datasets.MNIST('../data', train=True, download=True,
                    transform=transform)
dataset2 = datasets.MNIST('../data', train=False,
                    transform=transform)
train_loader = torch.utils.data.DataLoader(dataset1,**train_kwargs)
test_loader = torch.utils.data.DataLoader(dataset2, **test_kwargs)

model = Net().to(device)
optimizer = optim.Adadelta(model.parameters(), lr=args.lr)

scheduler = StepLR(optimizer, step_size=1, gamma=args.gamma)
for epoch in range(1, args.epochs + 1):
    train(args, model, device, train_loader, optimizer, epoch)
    test(model, device, test_loader)
    scheduler.step()

if args.save_model:
    torch.save(model.state_dict(), "mnist_cnn.pt")
```

Output

```
Train Epoch: 14 [49280/60000 (82%)]     Loss: 0.008296
Train Epoch: 14 [49920/60000 (83%)]     Loss: 0.005037
Train Epoch: 14 [50560/60000 (84%)]     Loss: 0.008428
Train Epoch: 14 [51200/60000 (85%)]     Loss: 0.057804
Train Epoch: 14 [51840/60000 (86%)]     Loss: 0.003749
Train Epoch: 14 [52480/60000 (87%)]     Loss: 0.003240
Train Epoch: 14 [53120/60000 (88%)]     Loss: 0.006414
Train Epoch: 14 [53760/60000 (90%)]     Loss: 0.099195
Train Epoch: 14 [54400/60000 (91%)]     Loss: 0.007836
Train Epoch: 14 [55040/60000 (92%)]     Loss: 0.000728
Train Epoch: 14 [55680/60000 (93%)]     Loss: 0.016054
Train Epoch: 14 [56320/60000 (94%)]     Loss: 0.008375
Train Epoch: 14 [56960/60000 (95%)]     Loss: 0.000876
Train Epoch: 14 [57600/60000 (96%)]     Loss: 0.016582
Train Epoch: 14 [58240/60000 (97%)]     Loss: 0.000615
Train Epoch: 14 [58880/60000 (98%)]     Loss: 0.001572
Train Epoch: 14 [59520/60000 (99%)]     Loss: 0.000660

Test set: Average loss: 0.0261, Accuracy: 9915/10000 (99%)
```

```
Train Epoch: 11 [48640/60000 (81%)]     Loss: 0.010271
Train Epoch: 11 [49280/60000 (82%)]     Loss: 0.002402
Train Epoch: 11 [49920/60000 (83%)]     Loss: 0.093729
Train Epoch: 11 [50560/60000 (84%)]     Loss: 0.010077
Train Epoch: 11 [51200/60000 (85%)]     Loss: 0.106828
Train Epoch: 11 [51840/60000 (86%)]     Loss: 0.005659
Train Epoch: 11 [52480/60000 (87%)]     Loss: 0.005455
Train Epoch: 11 [53120/60000 (88%)]     Loss: 0.027010
Train Epoch: 11 [53760/60000 (90%)]     Loss: 0.151537
Train Epoch: 11 [54400/60000 (91%)]     Loss: 0.009186
Train Epoch: 11 [55040/60000 (92%)]     Loss: 0.000848
Train Epoch: 11 [55680/60000 (93%)]     Loss: 0.030423
Train Epoch: 11 [56320/60000 (94%)]     Loss: 0.020851
Train Epoch: 11 [56960/60000 (95%)]     Loss: 0.014048
Train Epoch: 11 [57600/60000 (96%)]     Loss: 0.034926
Train Epoch: 11 [58240/60000 (97%)]     Loss: 0.000992
Train Epoch: 11 [58880/60000 (98%)]     Loss: 0.003815
Train Epoch: 11 [59520/60000 (99%)]     Loss: 0.001527

Test set: Average loss: 0.0262, Accuracy: 9912/10000 (99%)
```