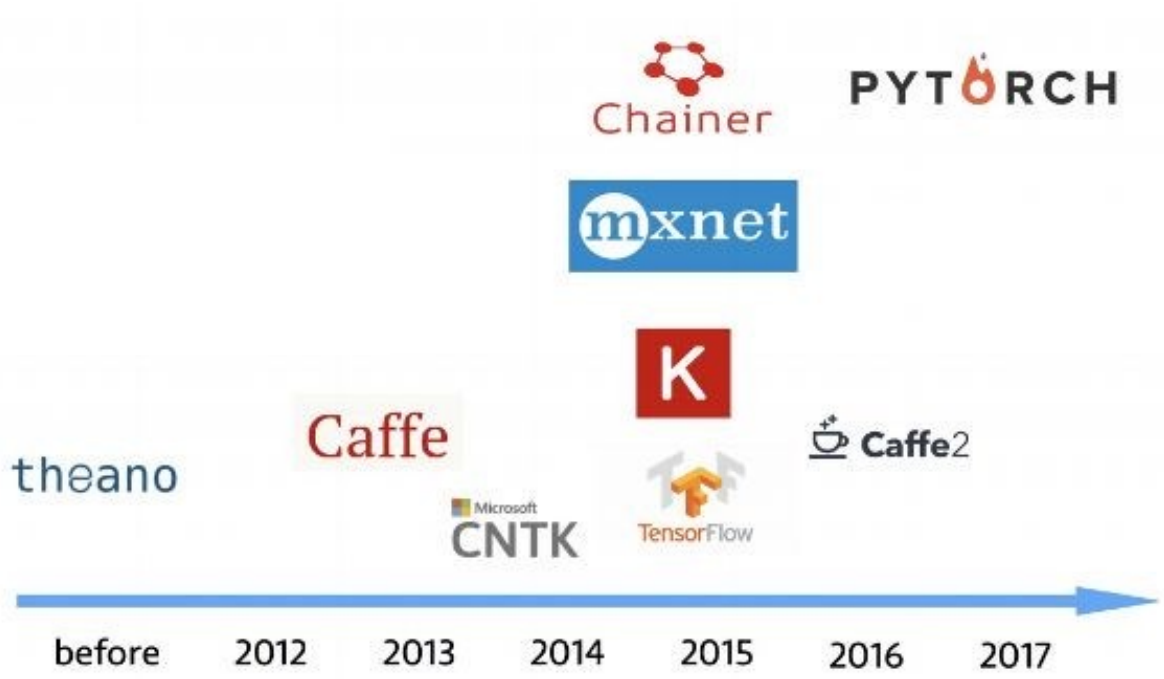
Network Training

Baotong Tian

ECE 208/408 – The Art of Machine Learning

(Slides adapted from Huiran Yu's version last year, and http://cs231n.stanford.edu/slides/2022/lecture_7_ruohan.pdf as well as official documentation from PyTorch and NVIDIA.)

Popular Deep Learning Frameworks



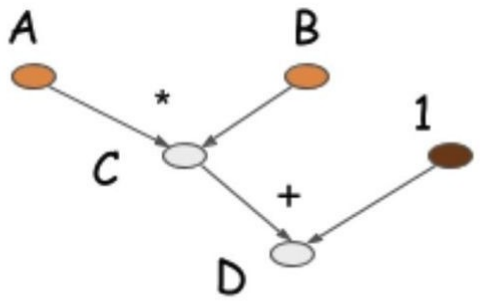
https://mli.github.io/cvpr17/

Computational graph

Imperative: Imperative-style programs perform computation as you run them

```
import numpy as np
a = np.ones(10)
b = np.ones(10) * 2
c = b * a
d = c + 1
```

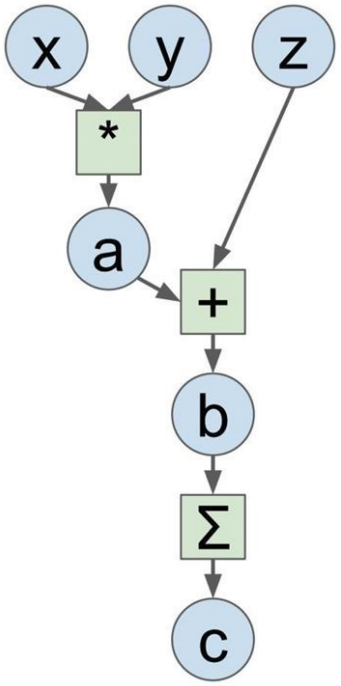
C = A * B
D = C + 1



Gluon: new MXNet interface to accelerate research

Tensor

Computational graph



NumPy

```
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

PyTorch

```
import torch

N, D = 3, 4

x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D, requires_grad=True)
z = torch.randn(N, D, requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad)
print(x.grad)
print(x.grad)
```

Tensor

Computational graph

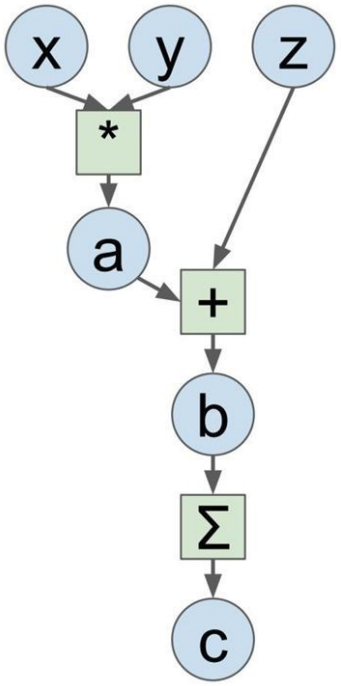
Define **tensor** with gradient required, which will be added to the computational graph

PyTorch

```
import torch

N, D = 3, 4

x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D, requires_grad=True)
z = torch.randn(N, D, requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad)
print(x.grad)
print(x.grad)
```

Tensor

Computational graph



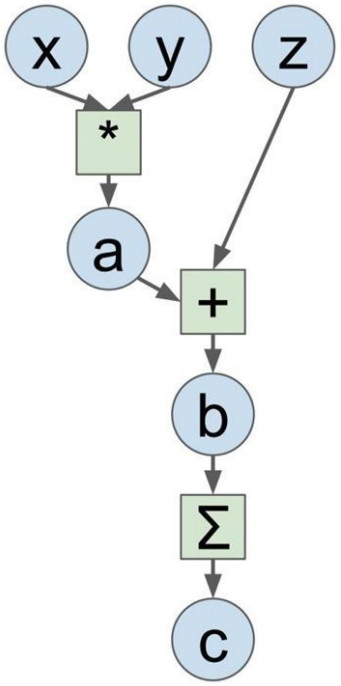
The forward pass looks just like numpy.
Remember, the function *, +, torch.sum() here are **pytorch functions**, these functions will build the dependency between tensors.

PyTorch

```
import torch

N, D = 3, 4

x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D, requires_grad=True)
z = torch.randn(N, D, requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad)
print(x.grad)
print(x.grad)
```

Tensor

Computational graph



Recall HW5, we do the backpropagation manually, but pytorch can do it automatically, due to the computational graphs.

c.backward() will calculate the derivative of c with respect to x, y, z, and will write the gradient to the grad attribute of x, y, z.

PyTorch

```
import torch

N, D = 3, 4

x = torch.randn(N, D, requires_grad=True)
y = torch.randn(N, D, requires_grad=True)
z = torch.randn(N, D, requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()

print(x.grad)
print(x.grad)
print(x.grad)
```

Module

A neural network layer is a module, such as a convolution layer (torch.nn.Conv2d). It often has the following:

- **Weight** attribute: store the weight of convolution kernel.
- Weight **Initialization** method: initialize the weight.
- **Forward** method: the computation build in the forward part
- **Backward** : this part is invisible to users, but is implemented by Pytorch already.
- **Buffer** attribute: weights but don’t require grad

Pytorch: Two levels of abstraction

Tensor:

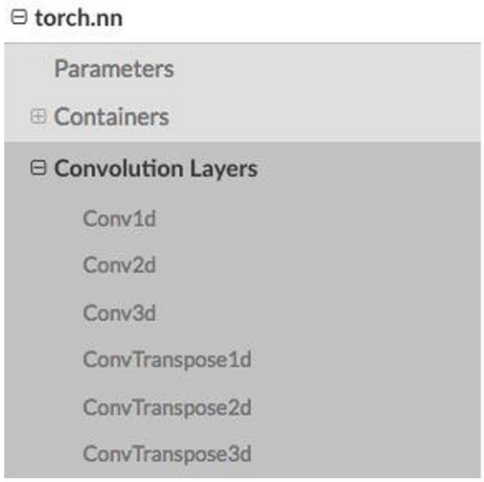
- if **`Tensor.requires_grad==False`**, it is imperative ndarray, but no gradient will be computed

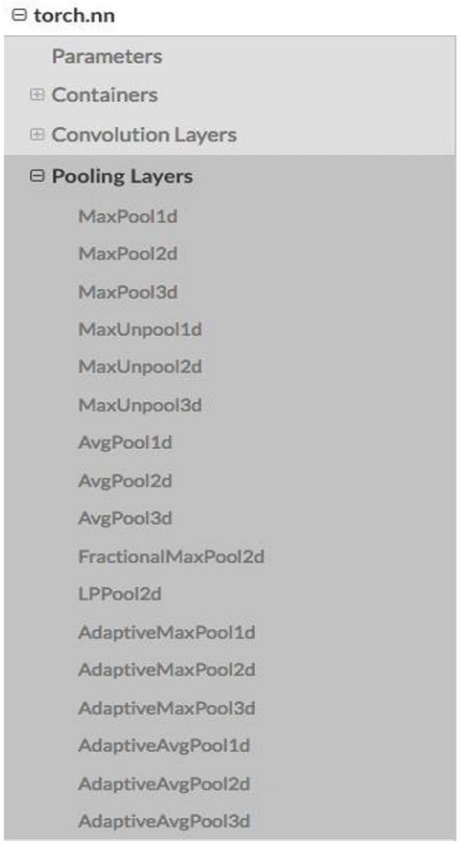
- if **`Tensor.requires_grad==True`**, it is a node in a computational graph; stores data and gradient

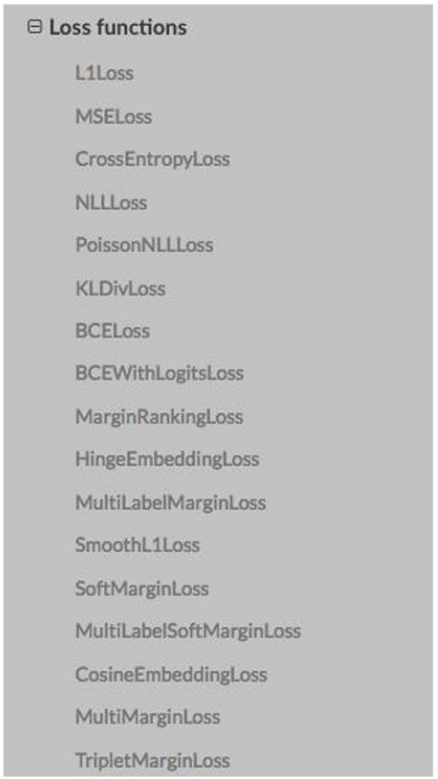
Module:

neural network layer(s); store learnable weights.

Module

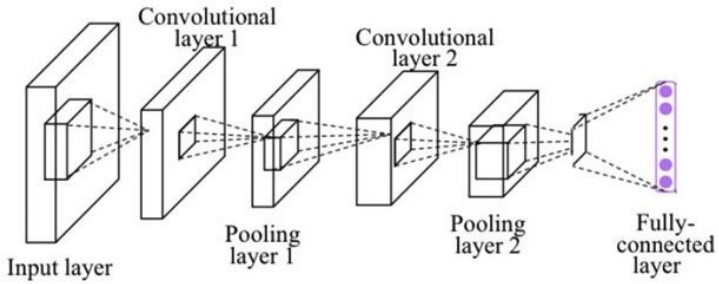






Other layers:
Dropout, Linear,
Normalization Layer

Module




```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.mp = nn.MaxPool2d(2)
        self.fc = nn.Linear(320, 10) # 320 -> 10

    def forward(self, x):
        in_size = x.size(0)
        x = F.relu(self.mp(self.conv1(x)))
        x = F.relu(self.mp(self.conv2(x)))
        x = x.view(in_size, -1) # flatten the tensor
        x = self.fc(x)
        return F.log_softmax(x)
```

Define a CNN

- Net class is a CNN defined by user, it inherits from `torch.nn.Module`
- Initialize the basic layers in `__init__`. Usually we only use the basic layer provided by pytorch to build our own network
- Define the `forward` method to build your computational graph
- When you call a module, it will automatically call the `forward` method of the module.

Check the model structure

[torchinfo · PyPI]

```python
from torchinfo import summary

model = Net()
batch_size = 16
summary(model, input_size=(batch_size, 1, 28, 28))
```

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
Net                                      [16, 10]                  --
├─Conv2d: 1-1                            [16, 10, 24, 24]          260
├─MaxPool2d: 1-2                         [16, 10, 12, 12]          --
├─Conv2d: 1-3                            [16, 20, 8, 8]            5,020
├─MaxPool2d: 1-4                         [16, 20, 4, 4]            --
├─Linear: 1-5                            [16, 10]                  3,210
==========================================================================================
Total params: 8,490
Trainable params: 8,490
Non-trainable params: 0
Total mult-adds (M): 7.59
==========================================================================================
Input size (MB): 0.05
Forward/backward pass size (MB): 0.90
Params size (MB): 0.03
Estimated Total Size (MB): 0.99
==========================================================================================
```

Dataset and Dataloader

```python
class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None, target_transform=None):
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
        return len(self.img_labels)

    def __getitem__(self, idx):
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = read_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        sample = {"image": image, "label": label}
        return sample
```

Dataset and Dataloader

The `Dataset` retrieves our dataset's features and labels one sample at a time. While training a model, we typically want to pass samples in "minibatches", reshuffle the data at every epoch to reduce model overfitting, and use Python's `multiprocessing` to speed up data retrieval.

`DataLoader` is an iterable that abstracts this complexity for us in an easy API.

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
```

Define a loss function and optimizer

Put `net.parameter()` to `optim.SGD`, so the gradient descent can be applied to the parameters of CNN.

`CrossEntropyLoss()` is also a module.

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

`Adam` optimizer is recommended. Reference to GBC Ch. 8.5

Train a neural network

set the model to the training mode, but it will not do any training. It inform layers such as `Dropout` and `BatchNorm`

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

https://stackoverflow.com/questions/51433378/what-does-model-train-do-in-pytorch

Train a neural network

Iterate the dataloader to get mini-batches of data

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Train a neural network

Equivalent to calling `model.forward(X)`

Compute loss

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

Train a neural network

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

set the gradients to zero

https://stackoverflow.com/questions/48001598/why-do-we-need-to-call-zero-grad-in-pytorch

Train a neural network

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

loss.backward() calculate all the gradient of loss w.r.t parameters

Train a neural network

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

optimizer.step() will update the parameter using SGD:
weight = weight - lr*weight.grad

Train a neural network

```python
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), (batch + 1) * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

loss.item() is to convert a torch.float type scaler into float

Save and load the trained network

When the training reaches the end or some particular conditions, for example the highest precision in the validation set, we would like to save the current parameters of the model. For more information, please check: Saving and Loading Models - PyTorch Tutorials

Save:

```
torch.save(model.state_dict(), PATH)
```

Load:

```
model = TheModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.eval()
```

Save and Load the trained network: Takeaways

- When loading models, use map_location='cpu' in most cases to conserve VRAM.
- In distributed training, ensure that checkpoints are saved only on the master process.
- To resume training, include the optimizer state in the checkpoint. For inference purposes, omit the optimizer state to reduce storage requirements.

PyTorch tutorials

https://pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html

http://cs231n.stanford.edu/slides/2022/discussion_4_pytorch.pdf

https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

GPU acceleration

https://transfer.d2.mpi-inf.mpg.de/rshetty/hlcv/Pytorch_tutorial.pdf

Using GPU

Bluehive: BluehiveInfo.pdf

Google Colab: Runtime / change runtime type

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

Move Tensors and Modules to GPU: `.cuda()` or `.to(device)`

Monitor GPU usage: `nvidia-smi` or gpustat · PyPI

How to prevent **overfitting** when training deep neural networks?

Three perspectives

Data

Model

Training strategies

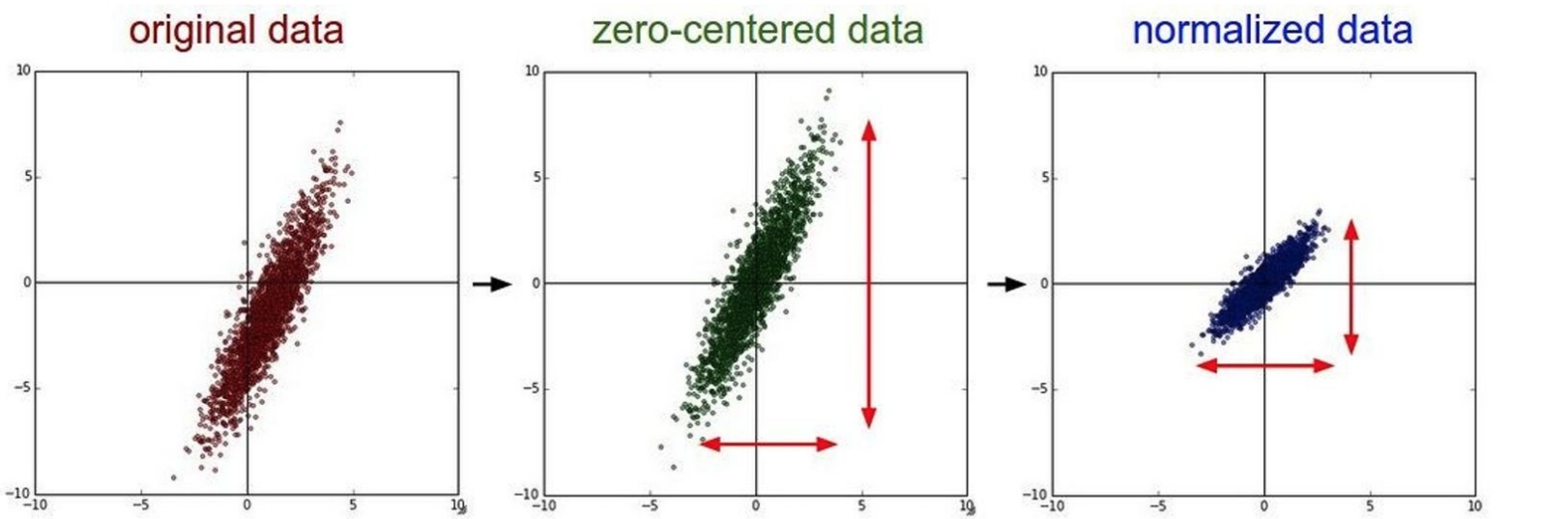
Data augmentation

Augment the training data

https://www.baeldung.com/cs/ml-data-augmentation

Data preprocessing

Make the optimization more stable and easier




```
X -= np.mean(X, axis = 0)
```

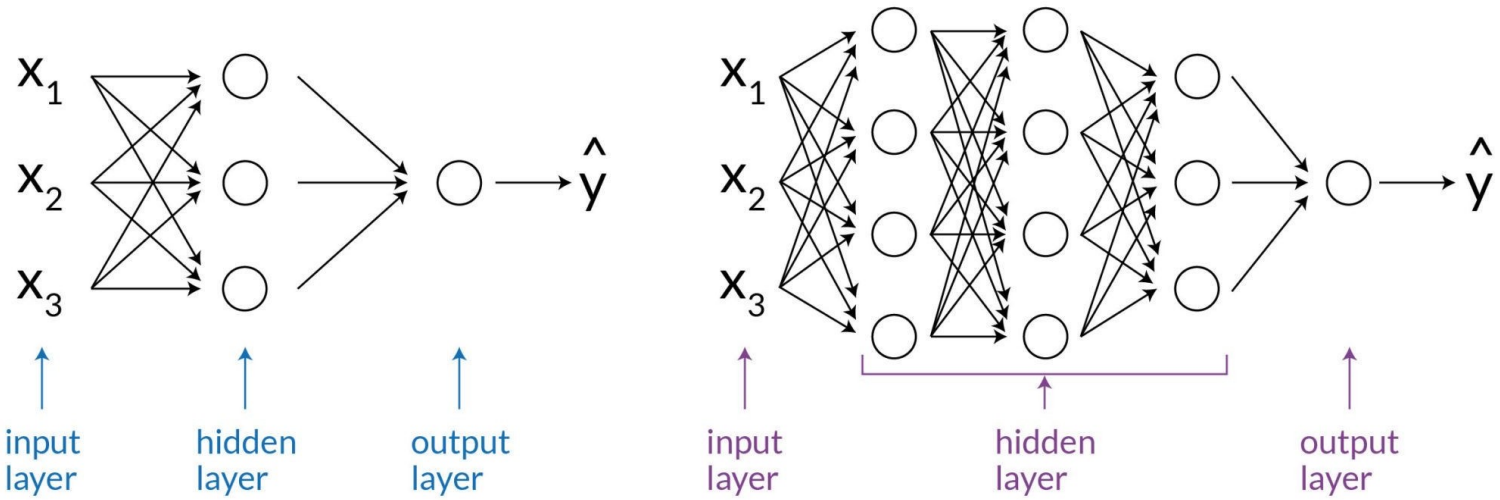
```
X /= np.std(X, axis = 0)
```

http://cs231n.stanford.edu/slides/2022/lecture_7_ruohan.pdf

Reduce model complexity

Reduce the number of layers or neurons in the network

Use simpler activation functions.



https://www.druva.com/blog/understanding-neural-networks-through-visualization/

Add regularization term to the loss function

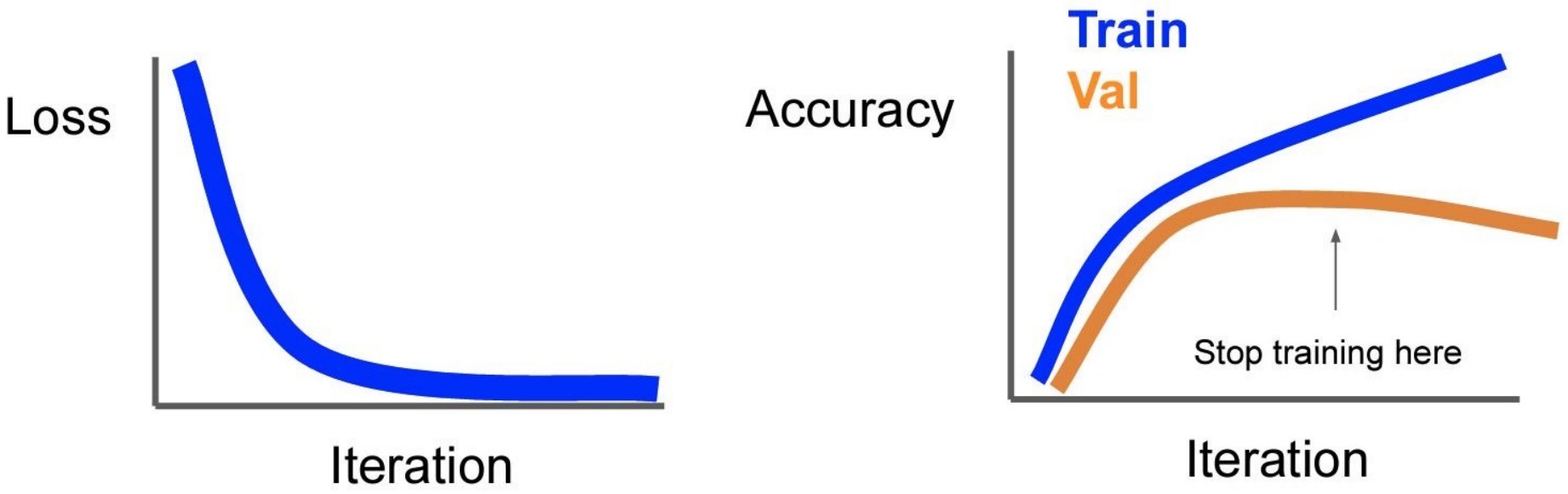
$$\widehat{\boldsymbol{\theta}} = \arg\min_{\boldsymbol{\theta}} \underbrace{J(\boldsymbol{\theta}; \mathbf{X}, \mathbf{y})}_{\text{(i)}} + \underbrace{\lambda}_{\text{(iii)}} \underbrace{R(\boldsymbol{\theta})}_{\text{(ii)}} .$$

(i) fit the training data

(ii) the regularization term, such as L1 or L2 norm.

(iii) hyperparameter for controlling the trade-off

Early stopping



Tools for observing learning curves: tensorboard, wandb

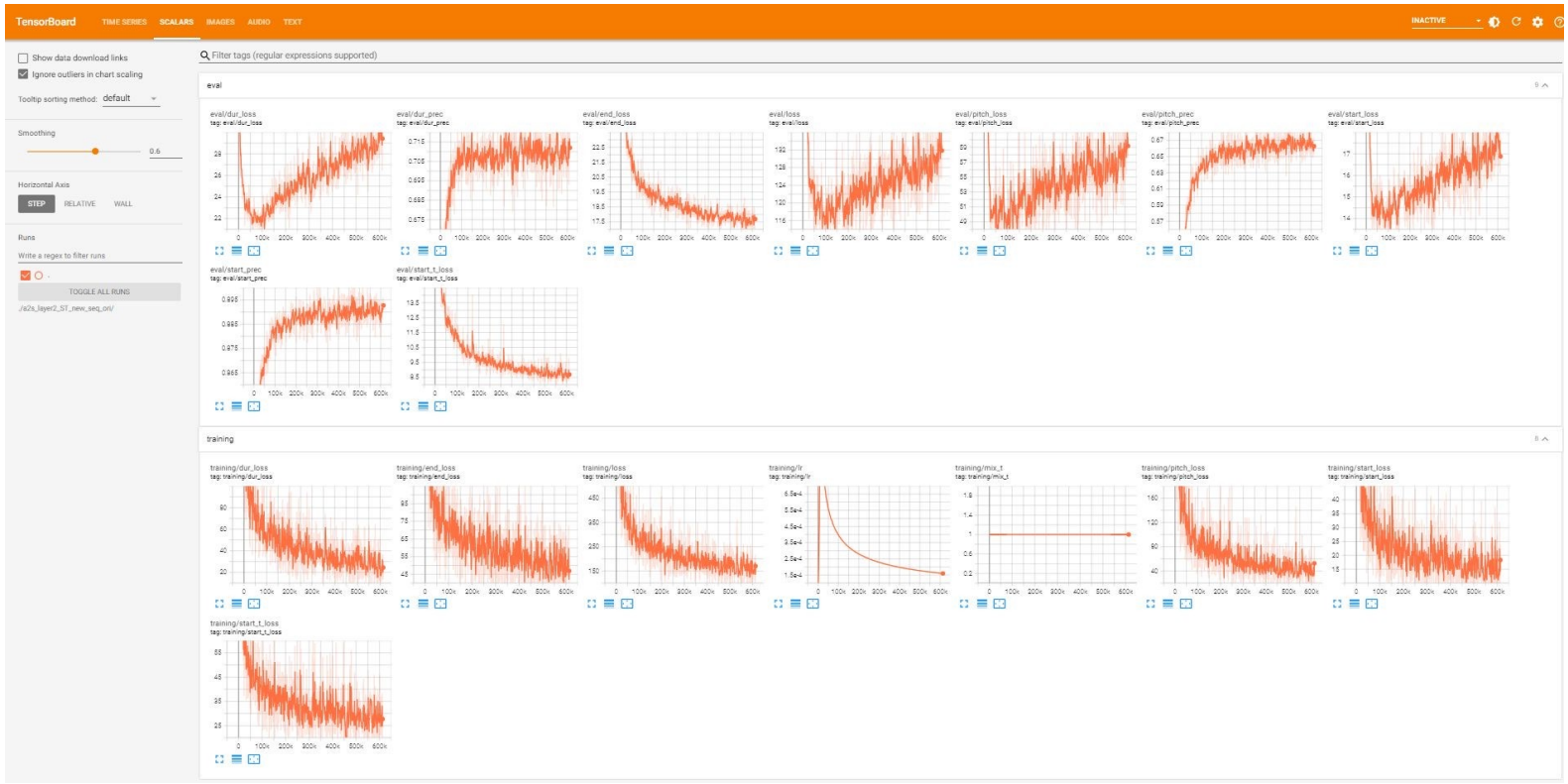
Tensorboard

- Install tensorboard with pip or conda
- Add code in your training script:

```python
from torch.utils.tensorboard import SummaryWriter

writer = SummaryWriter("path/to/logdir")

# record scaler
writer.add_scalar("tag/title", x, steps)

# record picture
add_image("tag/title", img_tensor, steps)
# shape of img_tensor: (3, H, W)

# More examples at: https://pytorch.org/docs/stable/tensorboard.html
```

- Run “tensorboard --logdir path/to/logdir” with command line and it will provide you a link to access the tensorboard web application. You can also launch it in the jupyter notebook if you need.

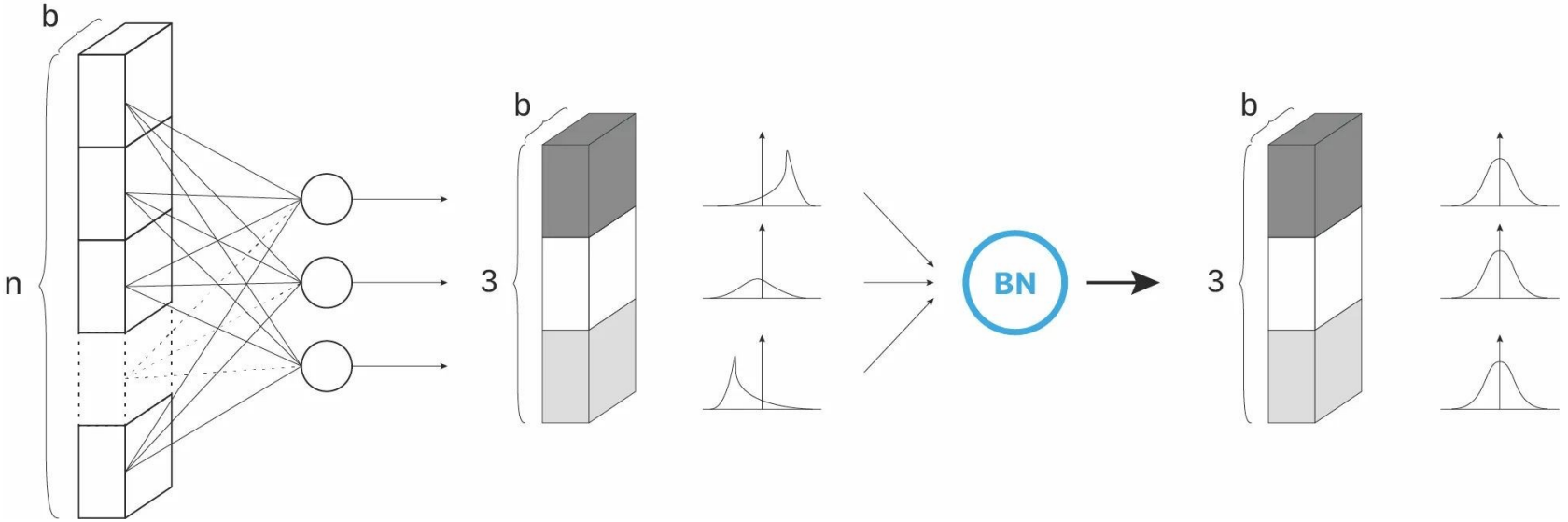
Tensorboard



Batch normalization

During training, it normalizes the activation values across the batch.

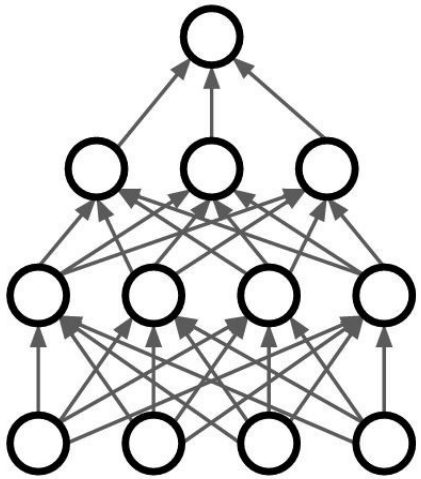
During testing, it uses the mean and variance values determined in the training.

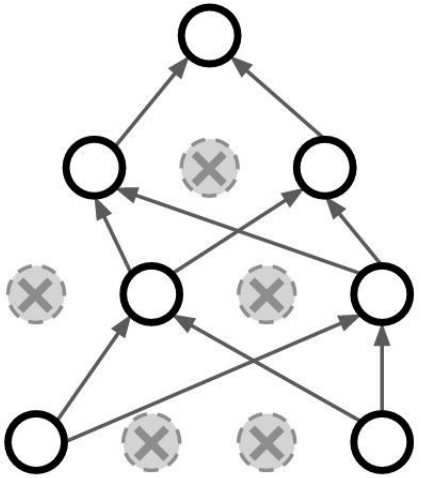


https://towardsdatascience.com/batch-normalization-in-3-levels-of-understanding-14c2da90a338

Dropout

During training, in each forward pass, randomly set some neurons to zero. Probability of dropping is a hyperparameter; 0.5 is common





At test time, all y dropout probability.

How to choose hyperparameters when training deep neural networks?

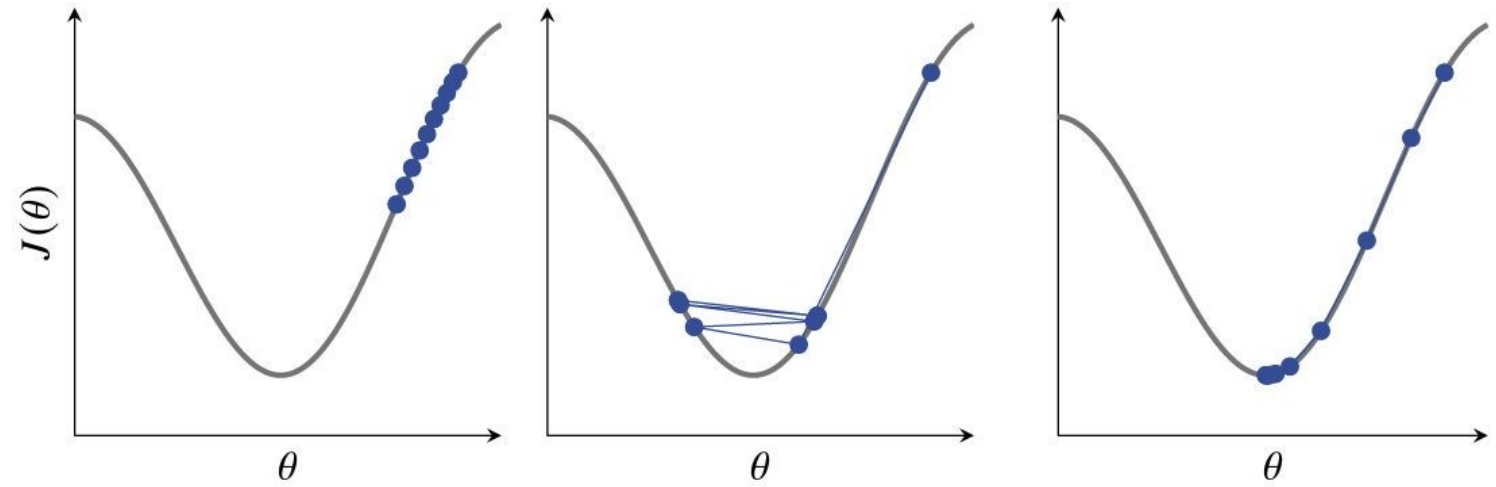
Choose hyperparameters

Choose the batch size according to your device.

Start with a learning rate that makes training loss go down. If not, overfit a small batch of samples to debug.

Look at the learning curves (loss and metrics).

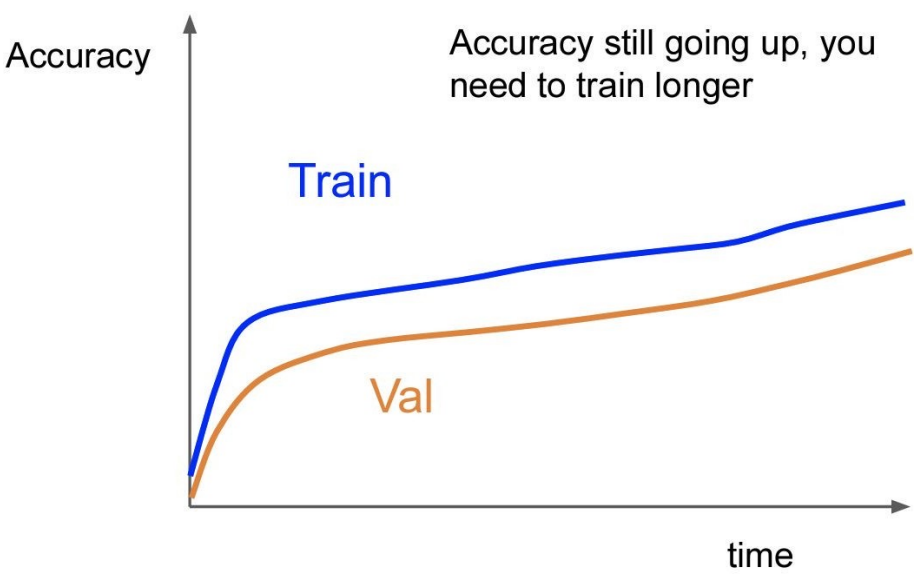
Learning rate

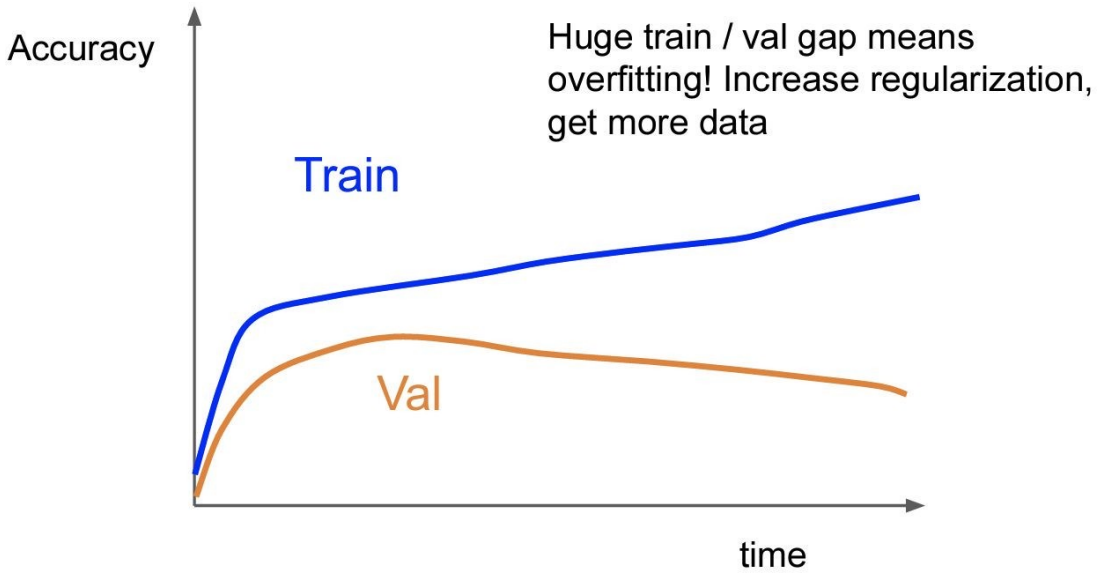


(a) Low learning rate $\gamma = 0.05$ **(b)** High learning rate $\gamma = 1.2$ **(c)** Good learning rate $\gamma = 0.3$

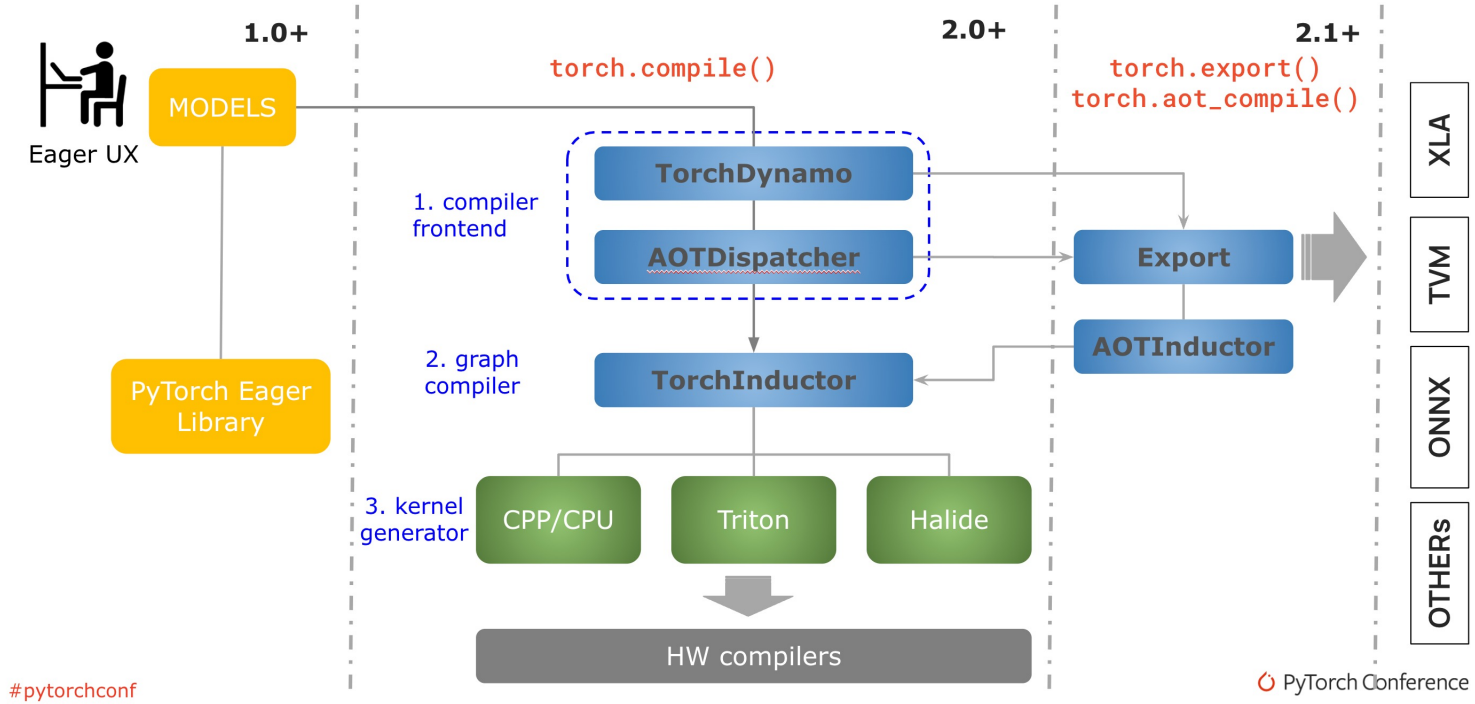
LWLS Figure 5-7

Learning curves





Training Acceleration: torch.compile

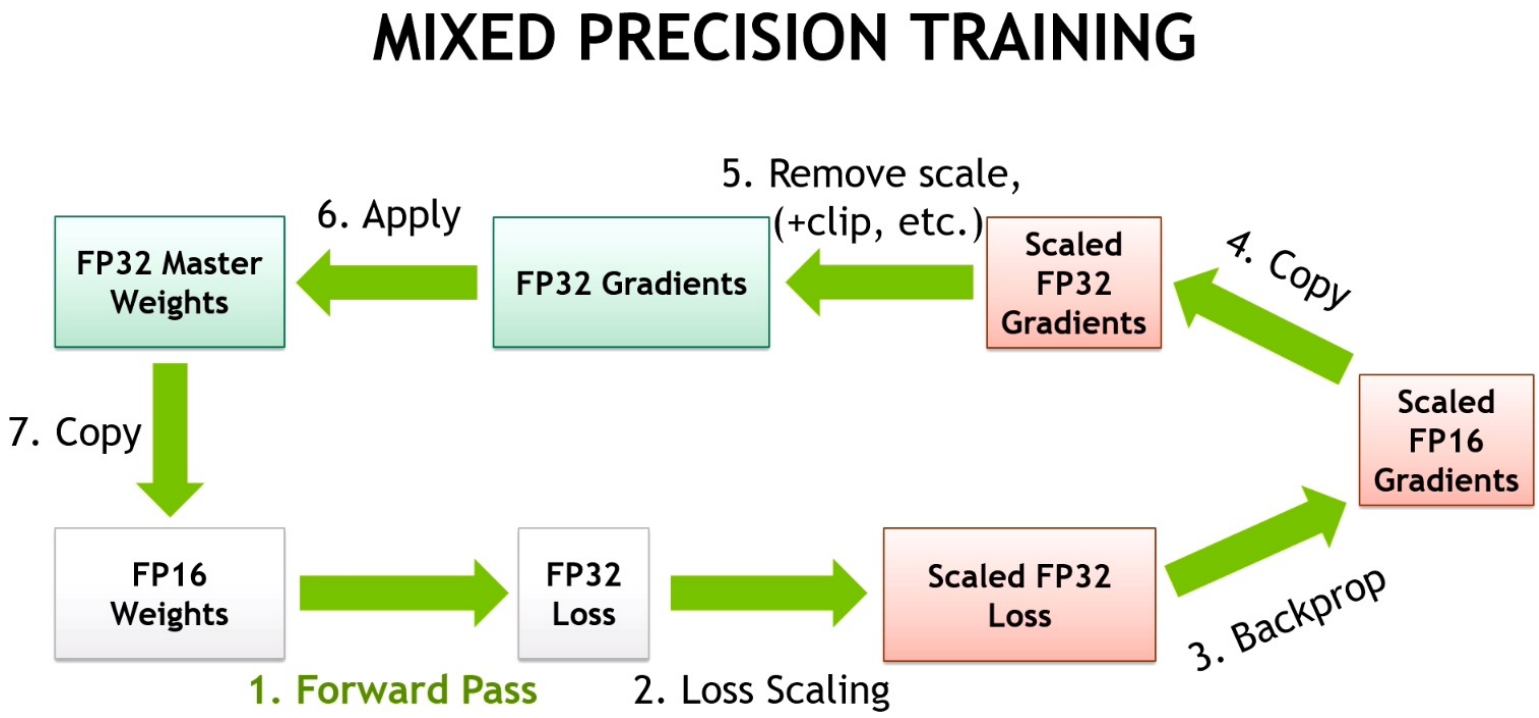


Eager to Static

Training Acceleration: torch.compile

```
def foo(x, y):
    a = torch.sin(x)
    b = torch.cos(y)
    return a + b
opt_foo1 = torch.compile(foo)
print(opt_foo1(torch.randn(10, 10), torch.randn(10, 10)))
```

Training Acceleration: Mixed Precision TRAINING



Training Acceleration: Mixed Precision

```
# Creates model and optimizer in default precision
model = Net().cuda()
optimizer = optim.SGD(model.parameters(), ...)

# Creates a GradScaler once at the beginning of training.
scaler = GradScaler()

for epoch in epochs:
    for input, target in data:
        optimizer.zero_grad()

        # Runs the forward pass with autocasting.
        with autocast(device_type='cuda', dtype=torch.float16):
            output = model(input)
            loss = loss_fn(output, target)

        # Scales loss.  Calls backward() on scaled loss to create scaled gradients.
        # Backward passes under autocast are not recommended.
        # Backward ops run in the same dtype autocast chose for corresponding forward ops.
        scaler.scale(loss).backward()

        # scaler.step() first unscales the gradients of the optimizer's assigned params.
        # If these gradients do not contain infs or NaNs, optimizer.step() is then called,
        # otherwise, optimizer.step() is skipped.
        scaler.step(optimizer)

        # Updates the scale for next iteration.
        scaler.update()
```

Lecture wrap up

We covered **PyTorch** basics. Practice with homework 6.

Preventing **overfitting** in deep neural networks requires a combination of techniques, including using more data, regularization, early stopping, reducing model complexity, dropout, batch normalization, etc.

Use learning curves to tune **hyperparameters**.