Google Classroom Code: mhxgl24

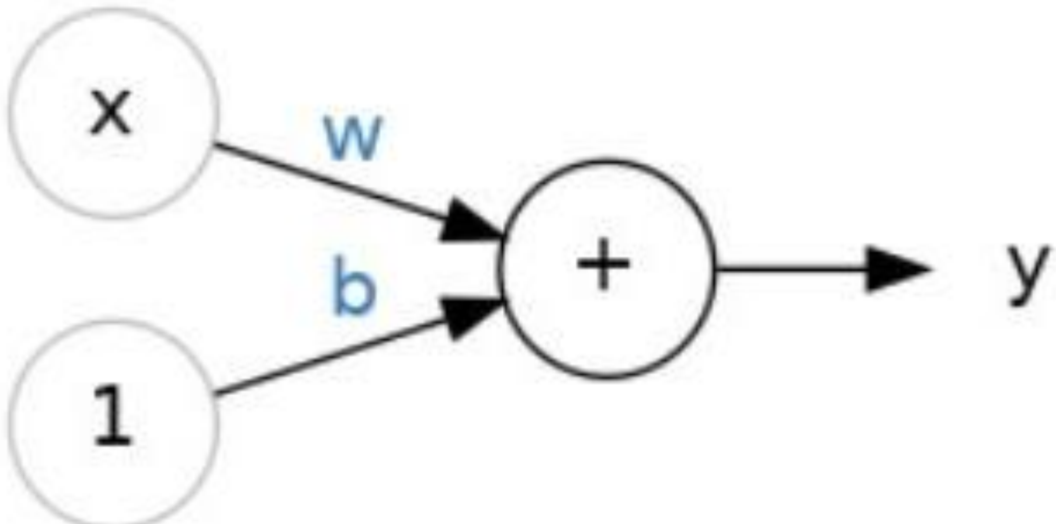
PyTorch

Deep Learning (DS-5006)
Dr. Adeel Mumtaz
Lecture 4
Fall, 2022

National University
Of Computer and Emerging Sciences

Contents

Simple Linear Regression



The Linear Unit: $y = wx + b$

- **NN=Architecture + Parameters**
- **Training NN = Given data learn best Parameters which gives minimum loss (usually an iterative process/algorithm)**

Simple Linear Regression

$$\mathcal{L}(x, b, w) = \frac{1}{N}\sum_{i=1}^{N}(wx_i + b - y_i)^2$$

$$dw = \frac{\partial \mathcal{L}(x, b, w)}{\partial w} = 2 * \frac{1}{N}\sum_{i=1}^{N}(wx_i + b - y_i)(x_i)$$

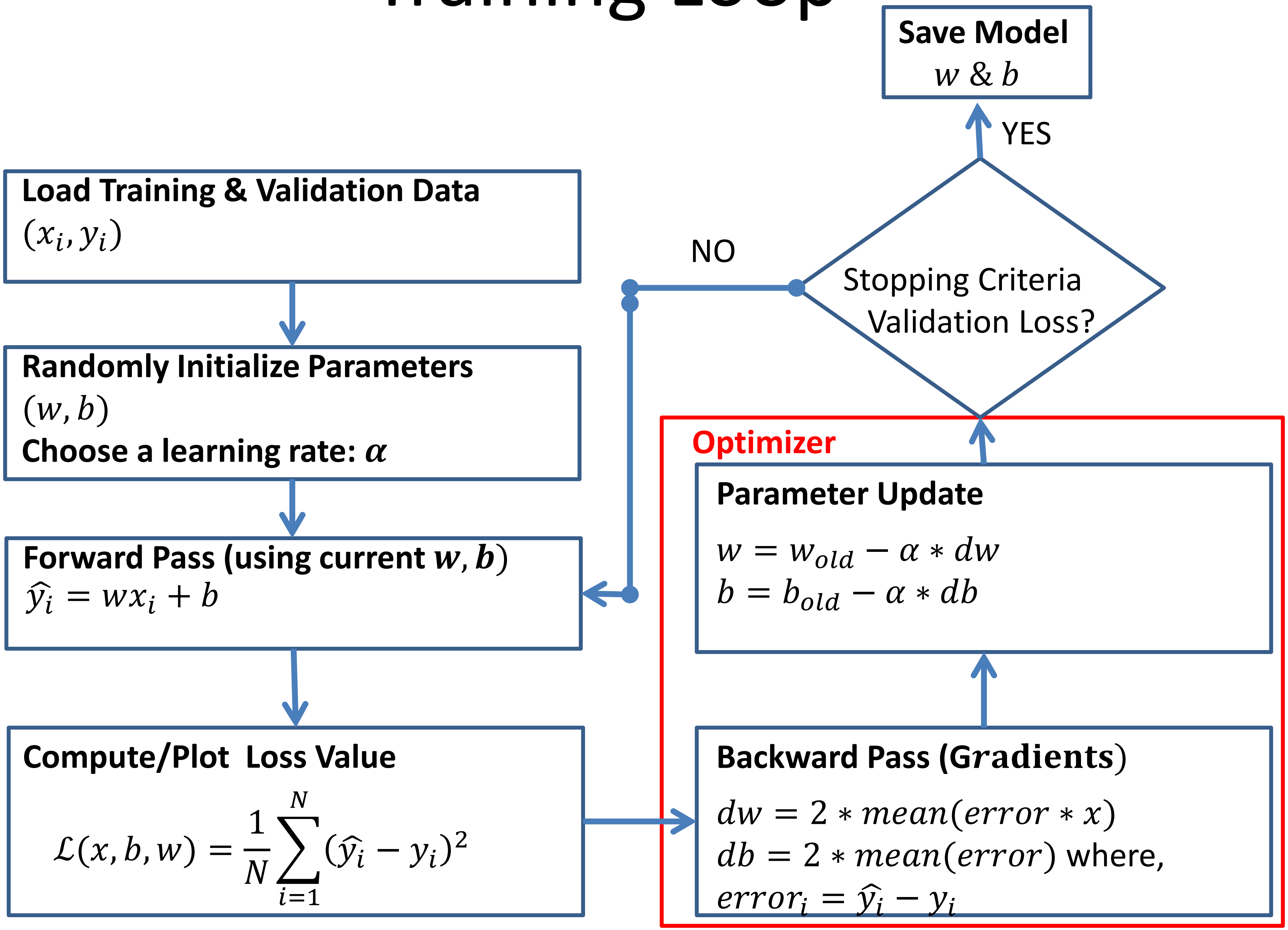
$$= 2 * \frac{1}{N}\sum_{i=1}^{N}(error_i)(x_i) = 2 * mean(error * x)$$

$$db = \frac{\partial \mathcal{L}(x, b, w)}{\partial b} = 2 * \frac{1}{N}\sum_{i=1}^{N}(error_i) = 2 * mean(error)$$

$$w_{new} = w_{old} - \alpha * dw$$

$$b_{new} = b_{old} - \alpha * db$$

Training Loop



Training Loop

```python
#training loop
#initializing parameters
trainLosses=[]
valLosses=[]
lr=0.1
w=np.random.randn(1)
b=np.random.randn(1)
for i in range(100):
    #forward pass
    yhat=w*x_train+b #note vectorized operation
    #MSE loss
    error=yhat-y_train
    loss= (error**2).mean()
    trainLosses.append(loss)
    #computing gradients
    db=2*error.mean()
    dw=2*(x_train*error).mean()
    #weight update
    b=b-lr*db
    w=w-lr*dw
```

NN Summary Review

- Data Set, Training, Validation, Test
- NN as function Approximators
- Cost/Loss Function
  - MSE Loss for regression
- Architecture of NN
- Parameters
- Training Loop
- Optimizer
- Learning Rate
- Types of Gradient Descent
  - Epoch
  - Batch
- Loading/Saving Model

Home Task

- Compare learning curves for different values of learning rate
- Convert code from Batch Gradient Descent to Stochastic Gradient Descent and compare learning curves

Quiz-1 (10min)

- Consider a regression problem with one input variable $\boldsymbol{x}$ and one output variable $\boldsymbol{y}$.
- Following regression model/architecture having unknown parameter $\boldsymbol{w}$ is assumed:
- $\boldsymbol{y_i = \log(w^2 x_i)}$
- Derive the **batch gradient descent update equation** for parameter w assuming MSE Loss

Solution

$$\mathcal{L}(x, b, w) = \frac{1}{N}\sum_{i=1}^{N}(\log(w^2 x_i) - y_i)^2$$

$$dw = \frac{\partial \mathcal{L}(x, b, w)}{\partial w} = 2 * \frac{1}{N}\sum_{i=1}^{N}(\log(w^2 x_i) - y_i) * \frac{1}{w^2 x_i} * 2 w x_i$$

$$dw = 4 * \frac{1}{N}\sum_{i=1}^{N}(\log(w^2 x_i) - y_i) * \frac{1}{w}$$

$$\mathrm{d}w = \frac{4}{N * w}\sum_{i=1}^{N}(\log(w^2 x_i) - y_i)$$

$$w_{new} = w_{old} - \alpha * dw$$

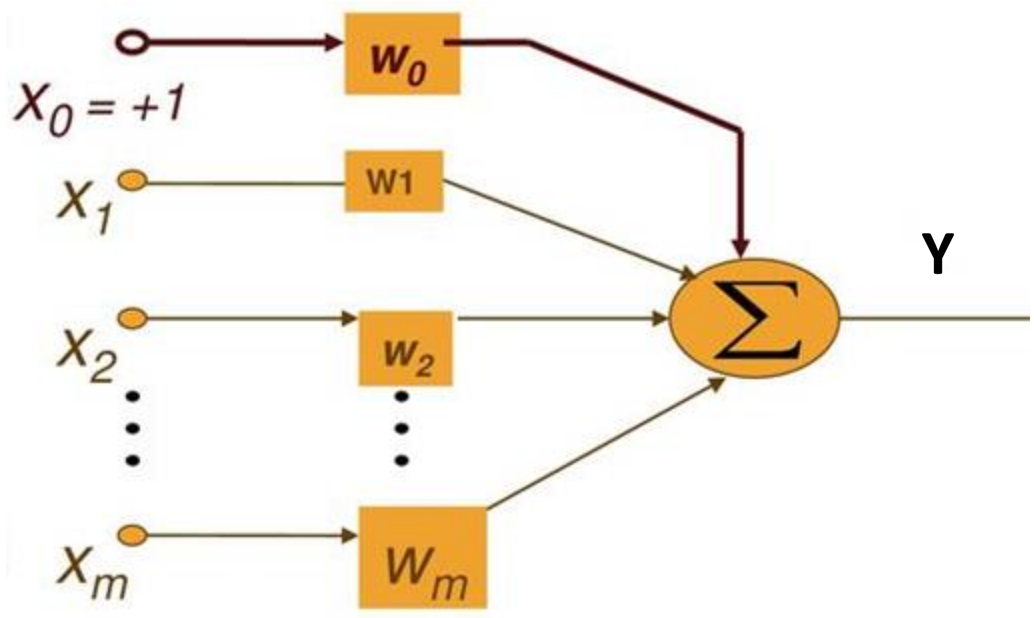
MULTIPLE REGRESSION

Multiple regression

- Multiple regression is a technique that can be used to analyze the relationship between a single dependent variable and several independent variables

Size (feet^2) x_1	Number of bedrooms x_2	Number of floors x_3	Age of home (years) x_4	Price ($1000) y
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

Model



Vectorization

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \cdots & x_n^{(1)} \\ 1 & x_1^{(2)} & x_2^{(2)} & \cdots & x_n^{(2)} \\ 1 & x_1^{(3)} & x_2^{(3)} & \cdots & x_n^{(3)} \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ 1 & x_1^{(m)} & x_2^{(m)} & \cdots & x_n^{(m)} \end{bmatrix}_{m\times(n+1)}$$

$$\hat{Y} = XW$$

$$W = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix}_{n+1}$$

$$Y = \begin{bmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{bmatrix}_{m}$$

Gradient Descent (Vectorization)

$$\mathcal{L}(X, W) = \frac{1}{m}\sum_{i=1}^{m}\left(W^T x^{(i)} - y^{(i)}\right)^2$$

$$dW = \frac{\partial \mathcal{L}(X, W)}{\partial W} = \frac{2}{m}\sum_{i=1}^{m}(x^{(i)}W - y^{(i)})\, x^{(i)}$$

$$W_{new} = W_{old} - \alpha * dW$$

$$b_{new} = ?$$

Scalar derivative			Vector derivative		
$f(x)$	$\rightarrow$	$\frac{df}{dx}$	$f(\mathbf{x})$	$\rightarrow$	$\frac{df}{d\mathbf{x}}$
bx	$\rightarrow$	b	$\mathbf{x}^T\mathbf{B}$	$\rightarrow$	$\mathbf{B}$
bx	$\rightarrow$	b	$\mathbf{x}^T\mathbf{b}$	$\rightarrow$	$\mathbf{b}$
x^2	$\rightarrow$	$2x$	$\mathbf{x}^T\mathbf{x}$	$\rightarrow$	$2\mathbf{x}$
bx^2	$\rightarrow$	$2bx$	$\mathbf{x}^T\mathbf{B}\mathbf{x}$	$\rightarrow$	$2\mathbf{B}\mathbf{x}$

Gradient Descent (Matrixization)

$$\mathcal{L}(X,W) = \frac{1}{m}\sum_{i=1}^{m}\left(W^T x^{(i)} - y^{(i)}\right)^2$$

$$dW = \frac{\partial \mathcal{L}(X,W)}{\partial W} = \frac{2}{m} X^T(XW - Y)$$

$$W_{new} = W_{old} - \alpha * dW$$

Scalar derivative $f(x)$	$\rightarrow$	$\frac{\mathrm{d}f}{\mathrm{d}x}$	Vector derivative $f(\mathbf{x})$	$\rightarrow$	$\frac{\mathrm{d}f}{\mathrm{d}\mathbf{x}}$
bx	$\rightarrow$	b	$\mathbf{x}^T\mathbf{B}$	$\rightarrow$	$\mathbf{B}$
bx	$\rightarrow$	b	$\mathbf{x}^T\mathbf{b}$	$\rightarrow$	$\mathbf{b}$
x^2	$\rightarrow$	$2x$	$\mathbf{x}^T\mathbf{x}$	$\rightarrow$	$2\mathbf{x}$
bx^2	$\rightarrow$	$2bx$	$\mathbf{x}^T\mathbf{B}\mathbf{x}$	$\rightarrow$	$2\mathbf{B}\mathbf{x}$

$$\mathcal{L}(X,W) = \frac{1}{m}(XW - Y)^T(XW - Y)$$

$$\mathcal{L}(X,W) = \frac{1}{m}(\,(XW)^T - Y^T)(XW - Y)$$

$$\mathcal{L}(X,W) = \frac{1}{m}(W^T X^T - Y^T)(XW - Y)$$

$$\mathcal{L}(X,W) = \frac{1}{m}(W^T X^T XW - W^T X^T Y - Y^T XW + Y^T Y)$$

$$dW = \frac{\partial \mathcal{L}(X,W)}{\partial W} = \frac{1}{m}(2X^T XW - X^T Y - X^T Y)$$

$$dW = \frac{\partial \mathcal{L}(X,W)}{\partial W} = \frac{2}{m}(X^T XW - X^T Y)$$

TIME TO TORCH IT :-)

PyTorch

- https://pytorch.org/
- PyTorch is a deep learning framework and scientific computing package based on Python that uses the power of graphics processing units (GPU)
- FROM RESEARCH TO PRODUCTION
  - An open source machine learning framework that accelerates the path from research prototyping to production deployment

PyTorch

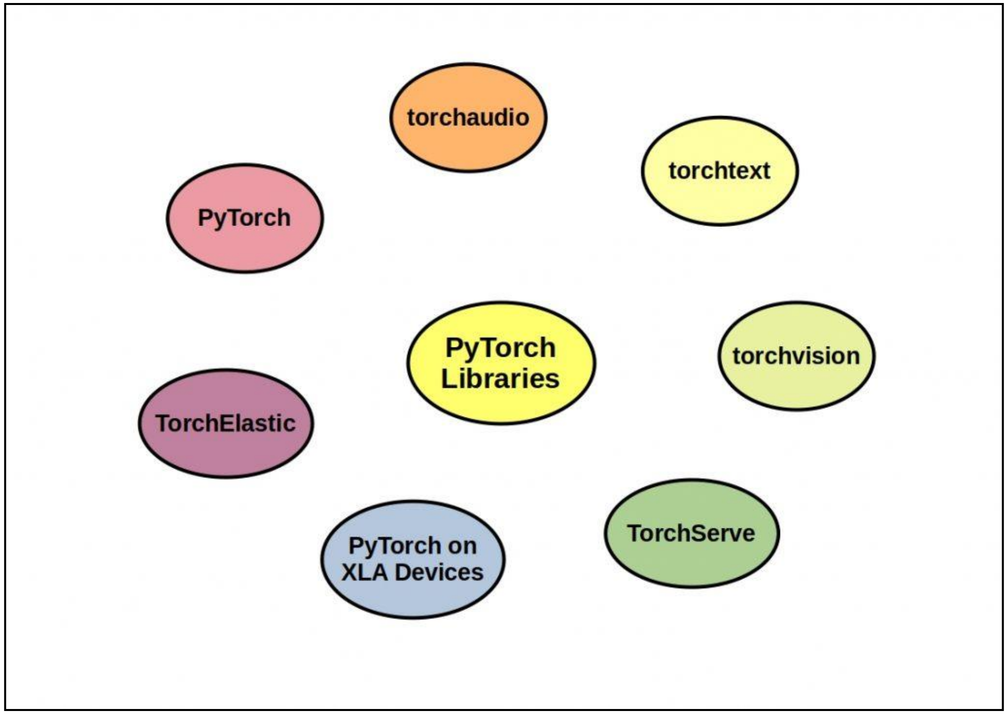
- Primarily developed by Facebook's AI Research lab (FAIR)
- PyTorch was launched in October of 2016 as Torch
- Caffe2 was merged into PyTorch at the end of March 2018
- A number of pieces of Deep Learning software are built on top of PyTorch, including:
  - Tesla Autopilot.
  - Uber's Pyro.
  - HuggingFace's Transformers.
  - PyTorch Lightning.
  - Catalyst.

PyTorch

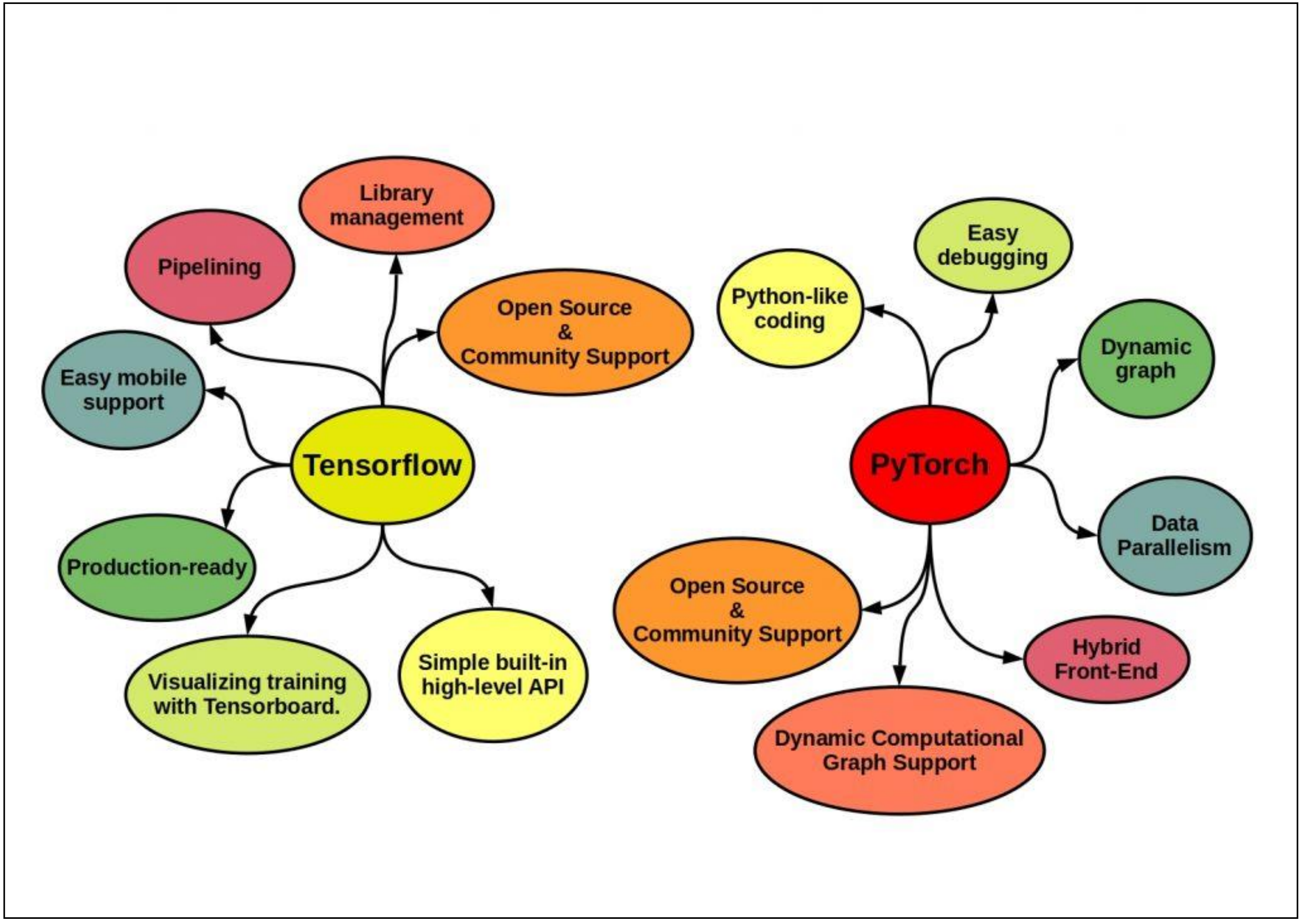
- Distributed Training
- ONNX (Open Neural Network Exchange) Support
- Autograd module
- Optim module
  - Most of the commonly used methods are already supported,
- nn module
  - layers and tools to easily create a neural networks by just defining the layers of the network.

PyTorch

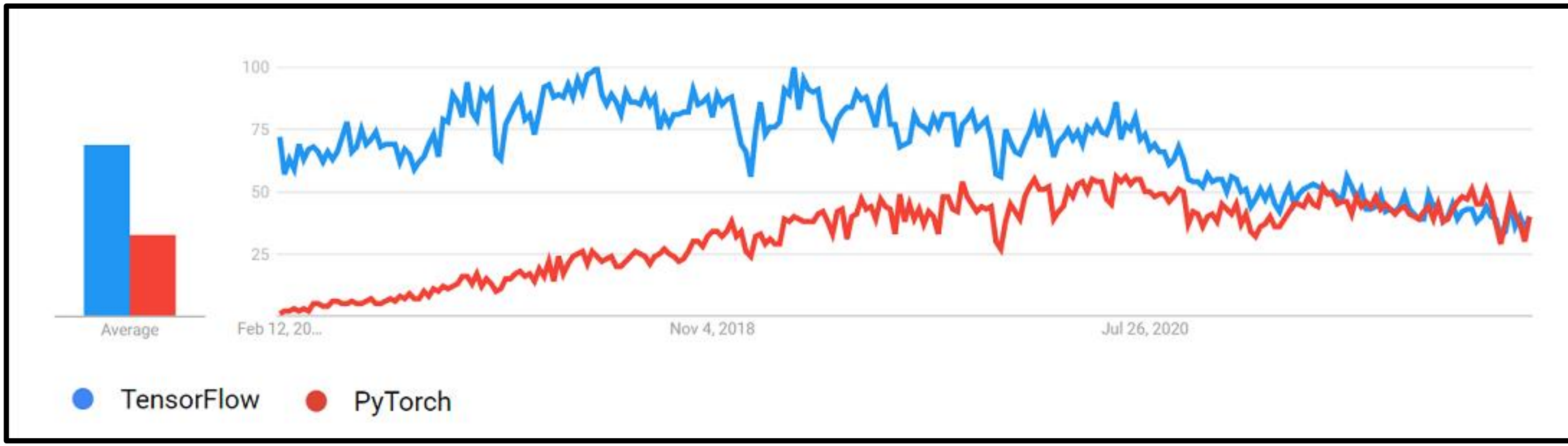
- **PyTorch rules research**
- It has become the framework of choice at CVPR, ICLR, and ICML, among others



PyTorch vs Tensorflow



PyTorch vs Tensorflow

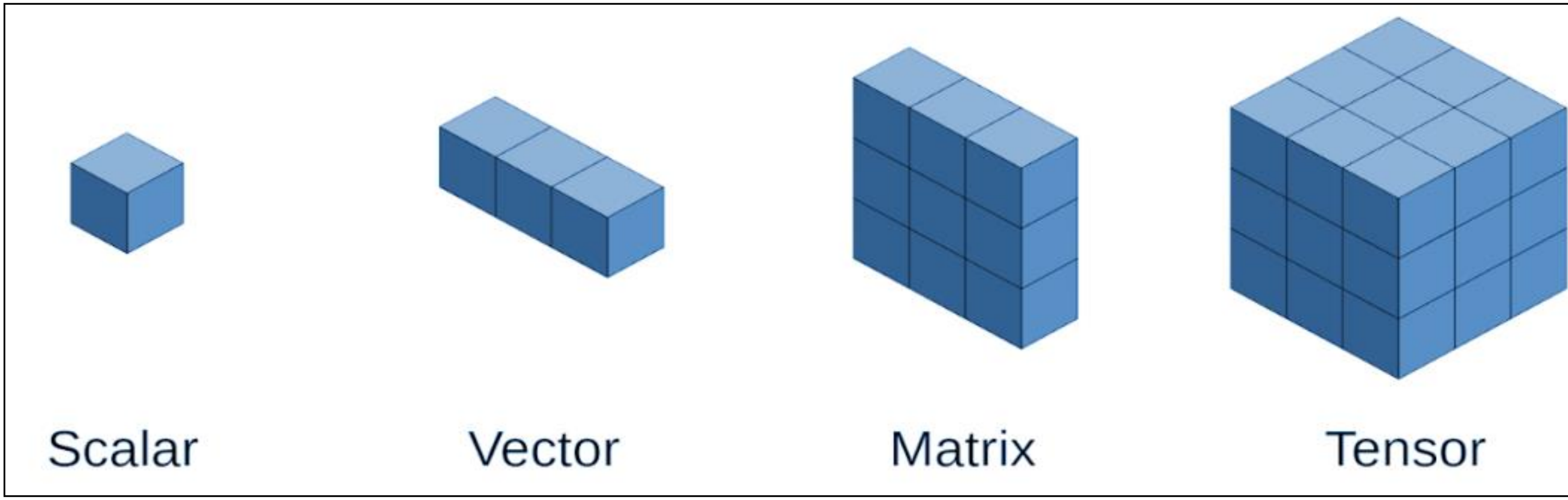


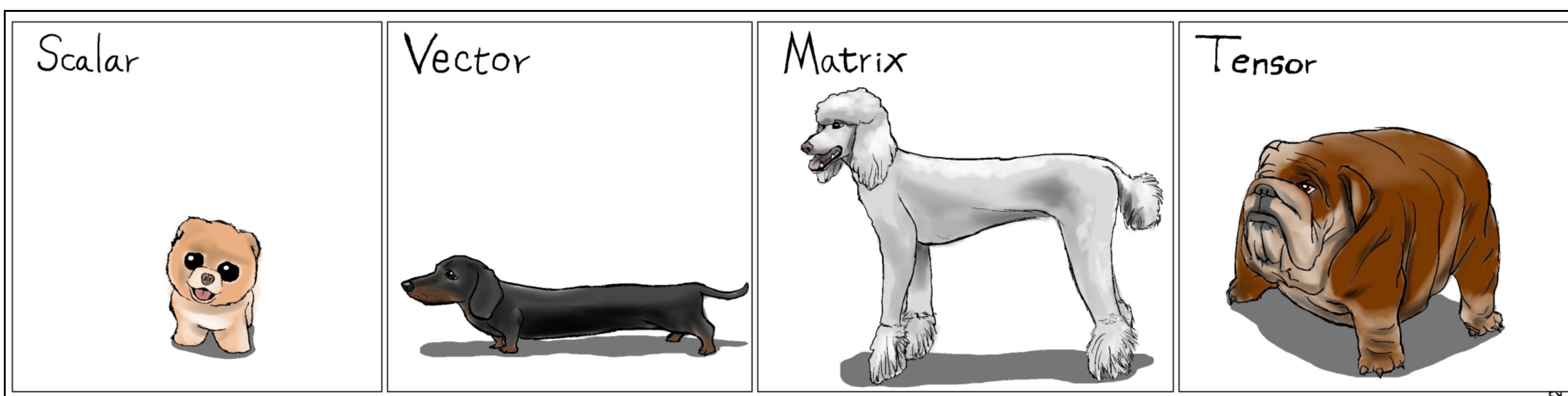
TENSORS

Tensor

- Scalar (a single number) has zero dimensions
- Vector has one dimension
- Matrix has two dimensions
- Tensor has three or more dimensions
- But, to keep things simple, it is commonplace to call vectors and matrices tensors as well
- so, from now on, **everything is either a scalar or a tensor**

Tensor





Creating Tensors

```python
import torch
```

```python
scalar = torch.tensor(3.14159)
vector = torch.tensor([1, 2, 3])
matrix = torch.ones((2, 3), dtype=torch.float)
tensor = torch.randn((2, 3, 4), dtype=torch.float)

print(scalar)
print(vector)
print(matrix)
print(tensor)
```

Creating Tensors

```
tensor(3.1416)
tensor([1, 2, 3])
tensor([[1., 1., 1.],
        [1., 1., 1.]])
tensor([[[-1.0658, -0.5675, -1.2903, -0.1136],
         [ 1.0344,  2.1910,  0.7926, -0.7065],
         [ 0.4552, -0.6728,  1.8786, -0.3248]],

        [[-0.7738,  1.3831,  1.4861, -0.7254],
         [ 0.1989, -1.0139,  1.5881, -1.2295],
         [-0.5338, -0.5548,  1.5385, -1.2971]]])
```

Getting Shape of Tensor

```python
print(tensor.size(), tensor.shape)
```

```
torch.Size([2, 3, 4]) torch.Size([2, 3, 4])
```

Reshaping a Tensor

- You can also reshape a tensor using its **view()** (preferred) or **reshape()** methods
- Beware it **DOES NOT create a new, independent, tensor**!

```
# We get a tensor with a different shape but it still is
# the SAME tensor
same_matrix = matrix.view(1, 6)
# If we change one of its elements...
same_matrix[0, 1] = 2.
# It changes both variables: matrix and same_matrix
print(matrix)
print(same_matrix)
```

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

Copying a Tensor

- If you want to copy all data for real, that is, duplicate the data in memory, you may
- use either its **new_tensor()** or clone() methods

```
# Lets follow PyTorch's suggestion and use "clone" method
another_matrix = matrix.view(1, 6).clone().detach()
# Again, if we change one of its elements...
another_matrix[0, 1] = 4.
# The original tensor (matrix) is left untouched!
print(matrix)
print(another_matrix)
```

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

Numpy to PyTorch

- **as_tensor() or from_numpy() is is used to convert Numpy arrays to PyTorch Tensors**
- **This operation preserves the type of the array**
- **both as_tensor() and from_numpy() return a tensor that shares the underlying data with the original Numpy array.**
- torch.tensor() always **makes a copy of Numpy array**

```
x_train_tensor = torch.as_tensor(x_train)
x_train.dtype, x_train_tensor.dtype
```

```
(dtype('float64'), torch.float64)
```

```
dummy_array = np.array([1, 2, 3])
dummy_tensor = torch.as_tensor(dummy_array)
# Modifies the numpy array
dummy_array[1] = 0
# Tensor gets modified too...
dummy_tensor
```

```
tensor([1, 0, 3])
```

PyTorch to Numpy

You can also perform the opposite operation, namely, transforming a PyTorch tensor back to a *Numpy* array. That's what numpy() is good for:

```
dummy_tensor.numpy()
```

```
array([1, 0, 3])
```

GPU Tensors

- **These tensors store their data in the graphics card's memory**
- **Operations on top of them are performed by the GPU**
- **If you have a graphics card from NVIDIA, you can use the power of its GPU to speed up model training.**
- **PyTorch supports the use of these GPUs for model training using CUDA (Compute Unified Device Architecture), which needs to be previously installed and configured**

PyTorch Build	Stable (1.12.1)		Preview (Nightly)	LTS (1.8.2)	
Your OS	Linux		Mac	Windows	
Package	Conda	Pip		LibTorch	Source
Language	Python			C++ / Java	
Compute Platform	CUDA 10.2	CUDA 11.3	CUDA 11.6	~~ROCm 5.1.1~~	CPU
Run this Command:	**NOTE:** 'conda-forge' channel is required for cudatoolkit 11.6 `conda install pytorch torchvision torchaudio cudatoolkit=11.6 -c pytorch -c conda-forge`				

GPU Tensors

- you should always make your code GPU-ready,
- that is, it should automatically run in a GPU, if one is available

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

"*Why* `cuda:0`*? Are there others, like* `cuda:1`, `cuda:2` *and so on?*"

```
n_cudas = torch.cuda.device_count()
for i in range(n_cudas):
    print(torch.cuda.get_device_name(i))
```

```
GeForce GTX 1060 6GB
```

GPU Tensors

- Moving tensor to GPU using .to()

```
gpu_tensor = torch.as_tensor(x_train).to(device)
gpu_tensor[0]
```

```
tensor([0.7713], device='cuda:0', dtype=torch.float64)
```

"*Should I use* `to(device)`, *even if I am using CPU only?*"

Putting it all together

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

# Our data was in Numpy arrays, but we need to transform them
# into PyTorch's Tensors and then we send them to the
# chosen device
x_train_tensor = torch.as_tensor(x_train).float().to(device)
y_train_tensor = torch.as_tensor(y_train).float().to(device)
```

```
# Here we can see the difference - notice that .type() is more
# useful since it also tells us WHERE the tensor is (device)
print(type(x_train), type(x_train_tensor), x_train_tensor.type())
```

```
<class 'numpy.ndarray'> <class 'torch.Tensor'>
torch.cuda.FloatTensor
```

GPU Tensors to Numpy

```
back_to_numpy = x_train_tensor.numpy()
```

```
TypeError: can't convert CUDA tensor to numpy. Use
Tensor.cpu() to copy the tensor to host memory first.
```

```
back_to_numpy = x_train_tensor.cpu().numpy()
```

Summary So Far

```
scalar = torch.tensor(3.14159)
vector = torch.tensor([1, 2, 3])
matrix = torch.ones((2, 3), dtype=torch.float)
tensor = torch.randn((2, 3, 4), dtype=torch.float)
```

```
print(tensor.size(), tensor.shape)
```

```
same_matrix = matrix.view(1, 6)
```

```
another_matrix = matrix.view(1, 6).clone().detach()
```

```
x_train_tensor = torch.as_tensor(x_train)
```

```
dummy_tensor.numpy()
```

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

```
x_train_tensor = torch.as_tensor(x_train).float().to(device)
```

```
back_to_numpy = x_train_tensor.cpu().numpy()
```

Creating Parameters

- What is difference between a tensor used for training (validation or test) data and a tensor used for the parameter/weight of NN?
- Parameters requires computation of gradients with respect to a loss function during training
- A tensor for a learnable parameter requires a gradient!
- Always assign tensors to a device at the moment of their creation to avoid unexpected behaviors!

Creating Parameters (best approach)

```
# FINAL
# We can specify the device at the moment of creation
# RECOMMENDED!

# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
print(b, w)
```

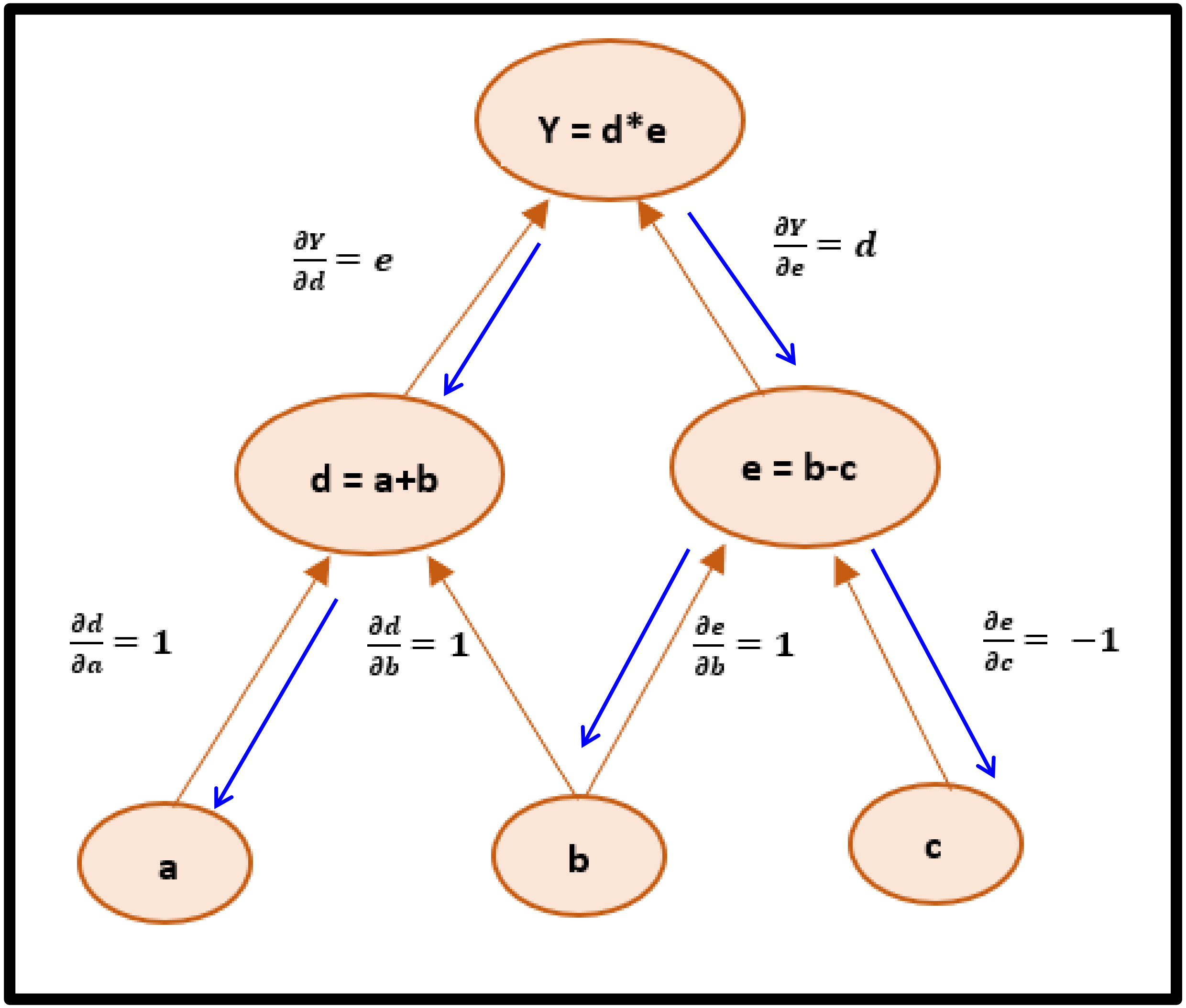
```
tensor([0.1940], device='cuda:0', requires_grad=True)
tensor([0.1391], device='cuda:0', requires_grad=True)
```

DYNAMIC COMPUTATION GRAPH

Computation Graph (Chain Rule)

- Computational graphs are a type of graph that can be used to represent mathematical expressions
- These can be used for two different types of calculations:
  - Forward computation
  - Backward computation
- Static Vs Dynamic Computation Graph

Example

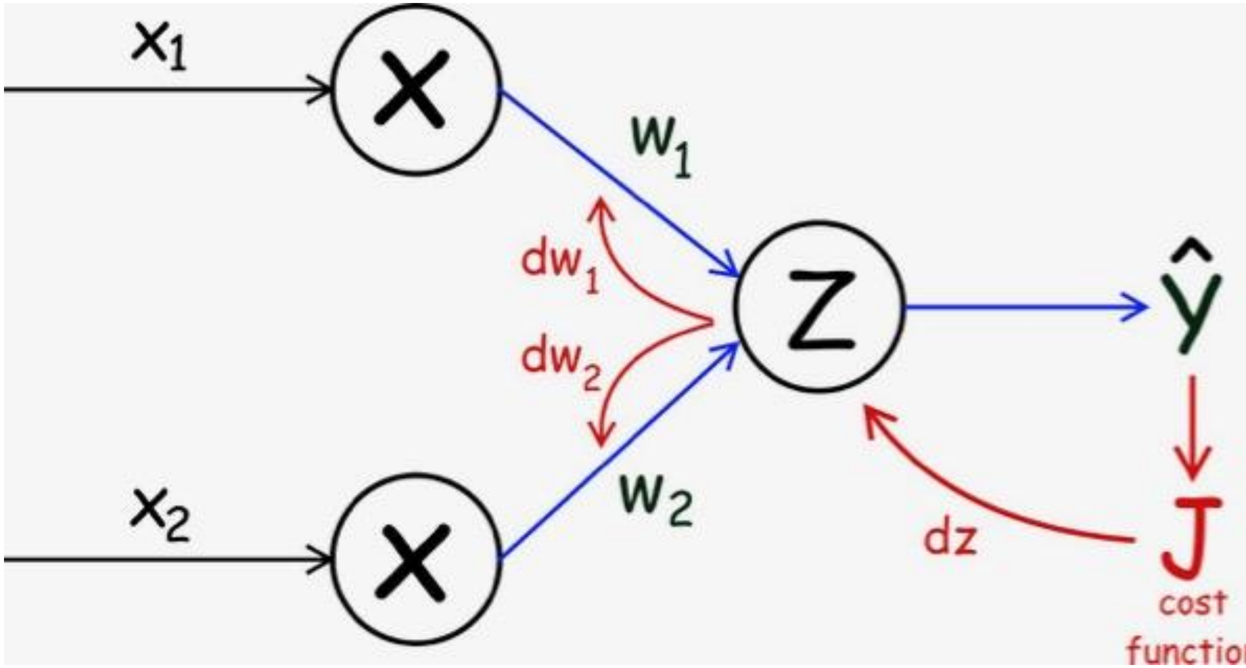


$$\frac{\partial Y}{\partial a} = \frac{\partial Y}{\partial d} \times \frac{\partial d}{\partial a} = e \times 1 = e$$

$$\frac{\partial Y}{\partial b} = \frac{\partial Y}{\partial d} \times \frac{\partial d}{\partial b} = e \times 1 = e$$

$$\frac{\partial Y}{\partial c} = \frac{\partial Y}{\partial e} \times \frac{\partial e}{\partial c} = d \times -1 = -d$$

Computation Graph (Chain Rule)

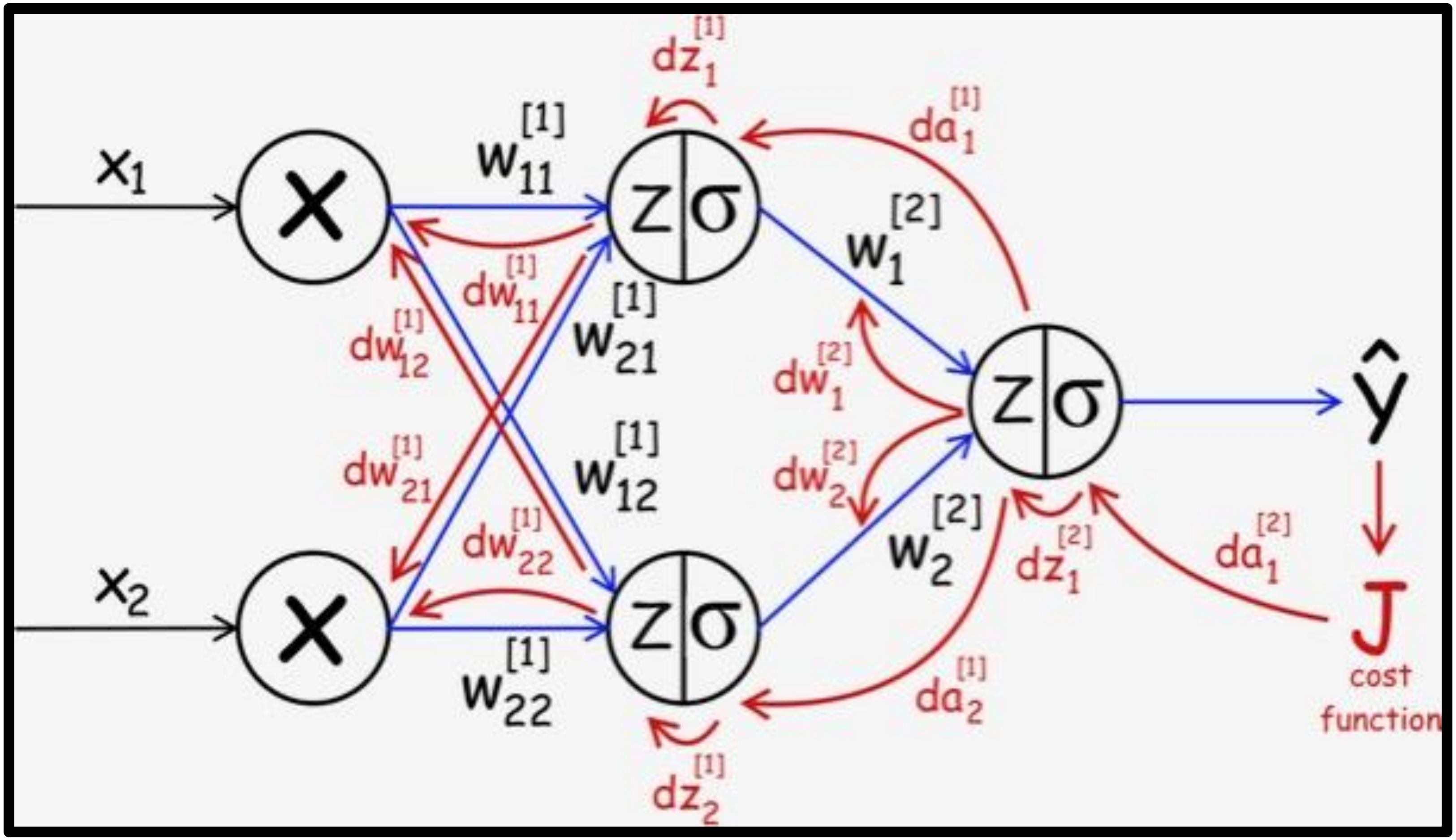


$$\hat{y} = z = x_1 w_1 + x_2 w_2$$

$$dw = \frac{\partial J}{\partial w} = \frac{\partial J}{\partial z} * \frac{\partial z}{\partial w}$$

$$J = f(z, y)$$

Computation Graph (Chain Rule)



PyTorch Computation Graph

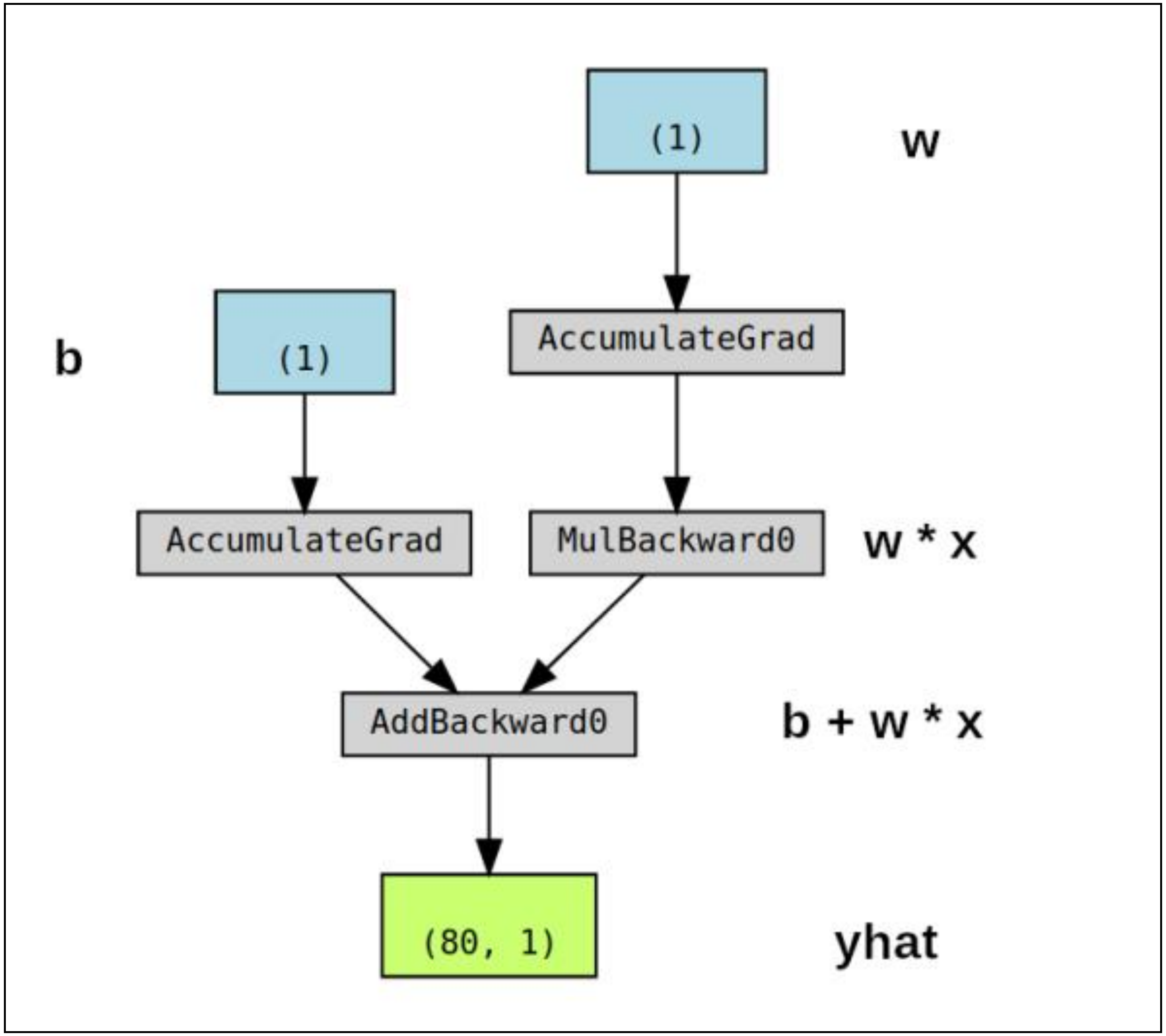
- You have to see it for yourself
- The **PyTorchViz** package and its make_dot(variable)

```
from torchviz import make_dot
```

```
# Step 0 - Initializes parameters "b" and "w" randomly
torch.manual_seed(42)
b = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
w = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
# Step 1 - Computes our model's predicted output - forward pass
yhat = b + w * x_train_tensor
# Step 2 - Computes the loss
error = (yhat - y_train_tensor)
loss = (error ** 2).mean()

# We can try plotting the graph for any python variable:
# yhat, error, loss...
make_dot(yhat)
```

PyTorch Computation Graph



Where is X?

AUTOGRAD

Autograd

- Autograd is PyTorch’s automatic differentiation package.
- Thanks to it, we don’t need to worry about partial derivatives, chain rule, or anything like it.
- Role of the **backward()** method
  - It will compute gradients for all (requiring gradient) tensors involved in the computation of a given variable
- Do you remember the **starting point** for computing the gradients?
  - ***loss.backward()***

Autograd in Action

```
yhat = b + w * x_train_tensor
```

```
error = (yhat - y_train_tensor)
```

```
loss = (error ** 2).mean()
```

```
# b_grad = 2 * error.mean()
# w_grad = 2 * (x_tensor * error).mean()
```

```
loss.backward()
```

Which tensors are going to be handled by the backward() method applied to the loss?

Autograd in Action

```
print(error.requires_grad, yhat.requires_grad, \
      b.requires_grad, w.requires_grad)
print(y_train_tensor.requires_grad, x_train_tensor.requires_grad)
```

```
True True True True
False False
```

grad

- What about the actual values of the gradients? We can inspect them by looking at the grad attribute of a tensor.

```
print(b.grad, w.grad)
```

```
tensor([-3.3881], device='cuda:0')
tensor([-1.9439], device='cuda:0')
```

Gradient Accumulation

- If you check the method's documentation, it clearly states that **gradients are accumulated**
- **Why?**
- If we run **above** code twice and check the grad attribute afterward

```
tensor([-6.7762], device='cuda:0')
tensor([-3.8878], device='cuda:0')
```

Zeroing Accumulation

- Every time we use the gradients to update the parameters, we need to zero the gradients afterward. And that's what **zero_()** is good for

```
b.grad.zero_(), w.grad.zero_()
```

```
(tensor([0.], device='cuda:0'),
tensor([0.], device='cuda:0'))
```

no_grad

- Perform regular Python operations on tensors, without affecting PyTorch's computation graph
- Will be used during inference

```
with torch.no_grad():
    b -= lr * b.grad
    w -= lr * w.grad
```

Summary So Far

```
b = torch.randn(1, requires_grad=True, \
                dtype=torch.float, device=device)
```

```
loss = (error ** 2).mean()
```

```
loss.backward()
```

```
print(b.grad, w.grad)
```

```
b.grad.zero_(), w.grad.zero_()
```

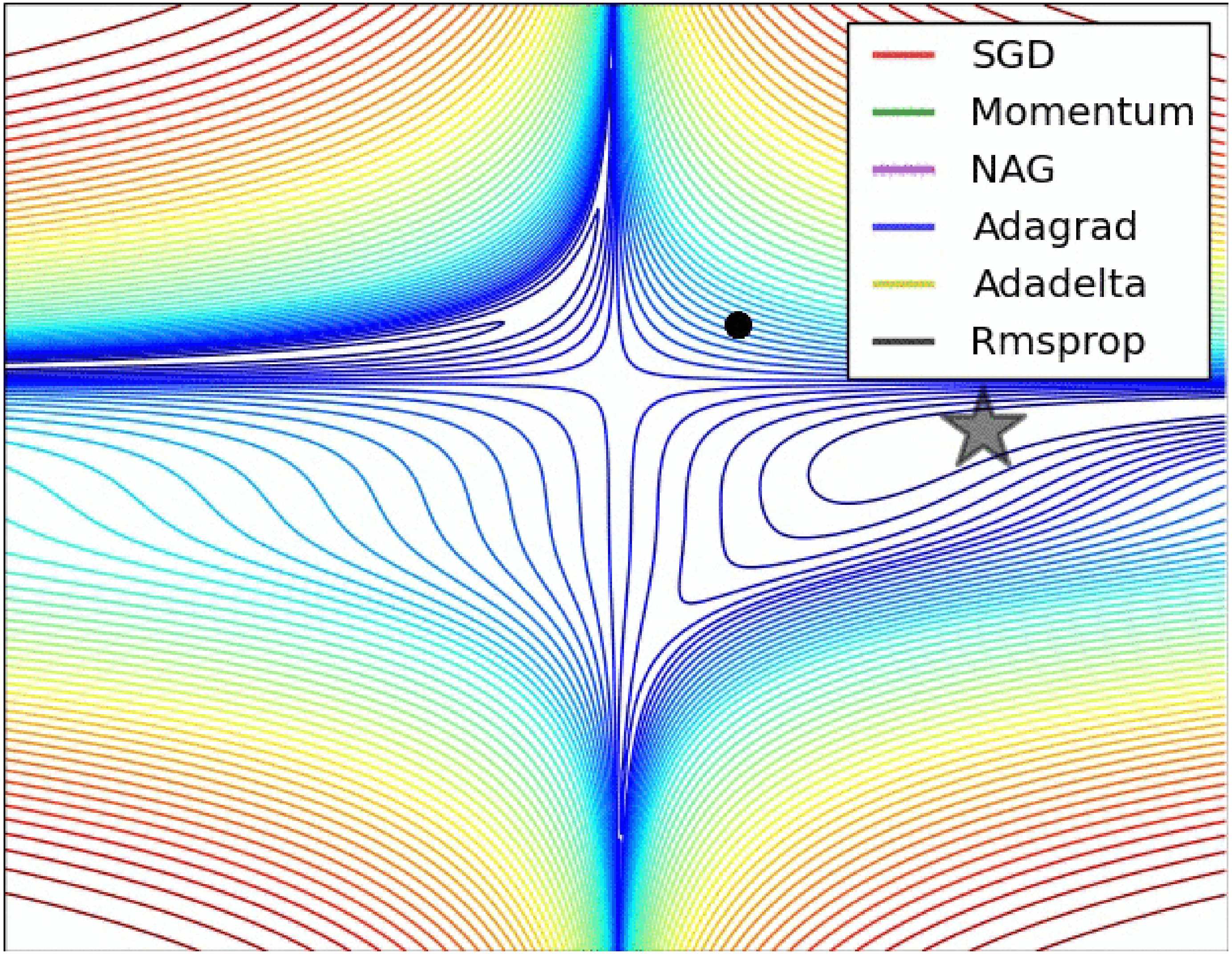
```
with torch.no_grad():
    b -= lr * b.grad
    w -= lr * w.grad
```

OPTIMIZERS

Optimizer

- We’ve been manually updating the parameters using the computed gradients so far
- What if we had a whole lot of parameters?!
- We need to use one of PyTorch’s optimizers, like **SGD**, RMSprop, or Adam.
- SGD is the most basic of them, and Adam is one of the most popular.
- Different optimizers use different mechanics for updating the parameters, but they all achieve the same goal through, literally, different paths

Optimizer



- **Remember, the choice of mini-batch size influences the path of gradient descent, and so does the choice of an optimizer**

Optimizer (**step / zero_grad**)

- An optimizer takes the parameters we want to update, the **learning rate** we want to use (and possibly many other hyper-parameters as well!), and performs the updates through its **step()** method

```python
import torch.optim as optim
```

```python
# Defines an SGD optimizer to update the parameters
optimizer = optim.SGD([b, w], lr=lr)
```

```python
#       b -= lr * b.grad
#       w -= lr * w.grad
optimizer.step()
```

```python
# b.grad.zero_()
# w.grad.zero_()
optimizer.zero_grad()
```

LOSS FUNCTIONS IN PYTORCH

Loss functions in PyTorch

- PyTorch got us covered with many loss functions to choose from, depending on the task at hand.
- Since ours is a regression, we are using the Mean Squared Error (MSE) as loss, and thus we need PyTorch's nn.MSELoss

```
# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')
```

```
predictions = torch.tensor(0.5, 1.0)
labels = torch.tensor(2.0, 1.3)
loss_fn(predictions, labels)
```

```
tensor(1.1700)
```

Loss functions in PyTorch

```
# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')
```

```
loss.cpu().numpy()
```

```
# error = (yhat - y_train_tensor)
# loss = (error ** 2).mean()
loss = loss_fn(yhat, y_train_tensor)
```

```
loss.detach().cpu().numpy()
```

```
# Step 3 - Computes gradients for both "b" and "w" parameters
loss.backward()
```

```
# Step 4 - Updates parameters using gradients and
# the learning rate
optimizer.step()
optimizer.zero_grad()
```

```
print(loss.item(), loss.tolist())
```

PYTORCH MODEL

Model Class

- Are you comfortable with object-oriented programming (OOP) concepts like classes, constructors, methods, instances, and attributes?
  - https://realpython.com/python3-object-oriented-programming/
  - Having a good understanding of OOP is key to benefit the most from PyTorch's capabilities.

Model Class Parts

- Must inherit from **nn.Module**
- **__init__(self)**:
  - it defines the parts that make up the model
  - In our case, two parameters, b and w.
  - You are **not** limited to defining **parameters**, though... **models can contain other models as their attributes** as well, so you can easily nest them. We'll see an example of this shortly as well
  - Do not forget to include **super().__init__()** to execute the __init__() method of the parent class (nn.Module) before your own

Model Class Parts

- forward(self, x):
  - It performs the actual computation, that is, it outputs a **prediction, given the input x**
  - We use model(x) instead of forward(x)

Model Class Parts

- forward(self, x):
  - It performs the actual computation, that is, it outputs a **prediction, given the input x**
  - We use model(x) instead of forward(x)

Our Regression Model

```
class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # To make "b" and "w" real parameters of the model,
        # we need to wrap them with nn.Parameter
        self.b = nn.Parameter(torch.randn(1,
                                          requires_grad=True,
                                          dtype=torch.float))
        self.w = nn.Parameter(torch.randn(1,
                                          requires_grad=True,
                                          dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.b + self.w * x
```

Model Parameters

- We define our two parameters, b and w, using the **Parameter() class**
- This tells PyTorch that these **tensors**, which are **attributes** of the ManualLinearRegression **class**, should be considered **parameters of the model**
- we can use our model's **parameters()** method to retrieve an iterator over all model's parameters, including parameters of nested model
- we can use it to feed our optimizer (instead of building a list of parameters ourselves!)

Model **state_dict**

- We can get the current values of all parameters using our model's **state_dict()** method
- Only learnable parameters are included, as its purpose is to keep track of parameters

```
dummy.state_dict()
```

```
OrderedDict([('b', tensor([0.3367])), ('w', tensor([0.1288]))])
```

Model **Device**

- We need to send our model to the same device where data is
  - If our data is made of GPU tensors, our model must “live” inside the GPU as well

```
torch.manual_seed(42)
# Creates a "dummy" instance of our ManualLinearRegression model
# and sends it to the device
dummy = ManualLinearRegression().to(device)
```

Nested Models

- PyTorch’s Linear model/layer

```
linear = nn.Linear(1, 1)
linear
```

```
Linear(in_features=1, out_features=1, bias=True)
```

```
linear.state_dict()
```

```
OrderedDict([('weight', tensor([[-0.2191]])),
('bias', tensor([0.2018]))])
```

Nested Models

```python
class MyLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # Instead of our custom parameters, we use a Linear model
        # with a single input and a single output
        self.linear = nn.Linear(1, 1)

    def forward(self, x):
        # Now it only takes a call
        self.linear(x)
```

```python
dummy = MyLinearRegression().to(device)
list(dummy.parameters())
```

```python
dummy.state_dict()
```

```
OrderedDict([('linear.weight',
              tensor([[0.7645]], device='cuda:0')),
             ('linear.bias',
              tensor([0.8300], device='cuda:0'))])
```

Summary So Far

- Optimizer
  - opt=optim.SGD(parameters)
  - opt.step()
  - opt.zero_grad()
- Loss Fuction
  - nn.MSELoss()
  - loss.backward()
- Model Class
  - __init__(self)
  - forward(self, x)
  - Defining Model Parameters
  - Model state_dict()
  - Model device
  - Nested Model

A SIMPLE REGRESSION PROBLEM (PYTORCH IMPLEMENTATION)

Imports, Device, Seed

```python
import numpy as np
import matplotlib.pyplot as plt
import torch
from torchviz import make_dot
```

```python
device='cuda' if torch.cuda.is_available() else 'cpu'
torch.manual_seed(100)
```

Model Class

```python
class SimpleRGNet(torch.nn.Module):
    def __init__(self):
        super().__init__()
        self.linear = torch.nn.Linear(1,1,bias=True)
    def forward(self,x):
        return self.linear(x)

```

Model, optimizer and loss initialization

```
model=SimpleRGNet().to(device)
paramList=list(model.parameters())
stateDict=model.state_dict()
print(paramList)
print(stateDict)

lr=0.1
optimizer=torch.optim.SGD(model.parameters(),lr=lr)
lossfnc = torch.nn.MSELoss(reduce="mean")
```

Data Preparation

```
true_w=2
true_b=1
N=100
#data generation
np.random.seed(100)
x=np.random.rand(N,1)
epsilon=0.1*np.random.randn(N,1)
y=true_w*x+true_b+epsilon
#data split
idx=np.arange(N)
np.random.shuffle(idx)
idx_train=idx[:int(0.8*N)]
idx_test=idx[int(0.8*N):]
x_train, y_train = x[idx_train],y[idx_train]
x_val, y_val = x[idx_test],y[idx_test]
x_train_tensor=torch.as_tensor(x_train).float().to(device)
y_train_tensor=torch.as_tensor(y_train).float().to(device)
x_val_tensor=torch.as_tensor(x_val).float().to(device)
y_val_tensor=torch.as_tensor(y_val).float().to(device)
```

Training Loop

```
trainLosses=[]
valLosses=[]
for i in range(1000):
    model.train()
    #forward pass
    yhat=model(x_train_tensor)
    loss=lossfnc(yhat,y_train_tensor)
    trainLosses.append(loss.item())
    #make_dot(loss).view()
    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
    stateDict=model.state_dict()
    w=stateDict['linear.weight']
    b=stateDict['linear.bias']
    w=w.item()
    b=b.item()

    model.eval()
    with torch.no_grad():
        #val MSE loss
        yhatval=model(x_val_tensor)
        valLoss=lossfnc(yhatval,y_val_tensor)
        valLosses.append(valLoss.item())
    #stopping condition
    if(valLoss.item()<0.0001):
        break
    print(f'train loss={loss.item()}, val loss={valLoss.item()}, w={w}, b={b}')
```

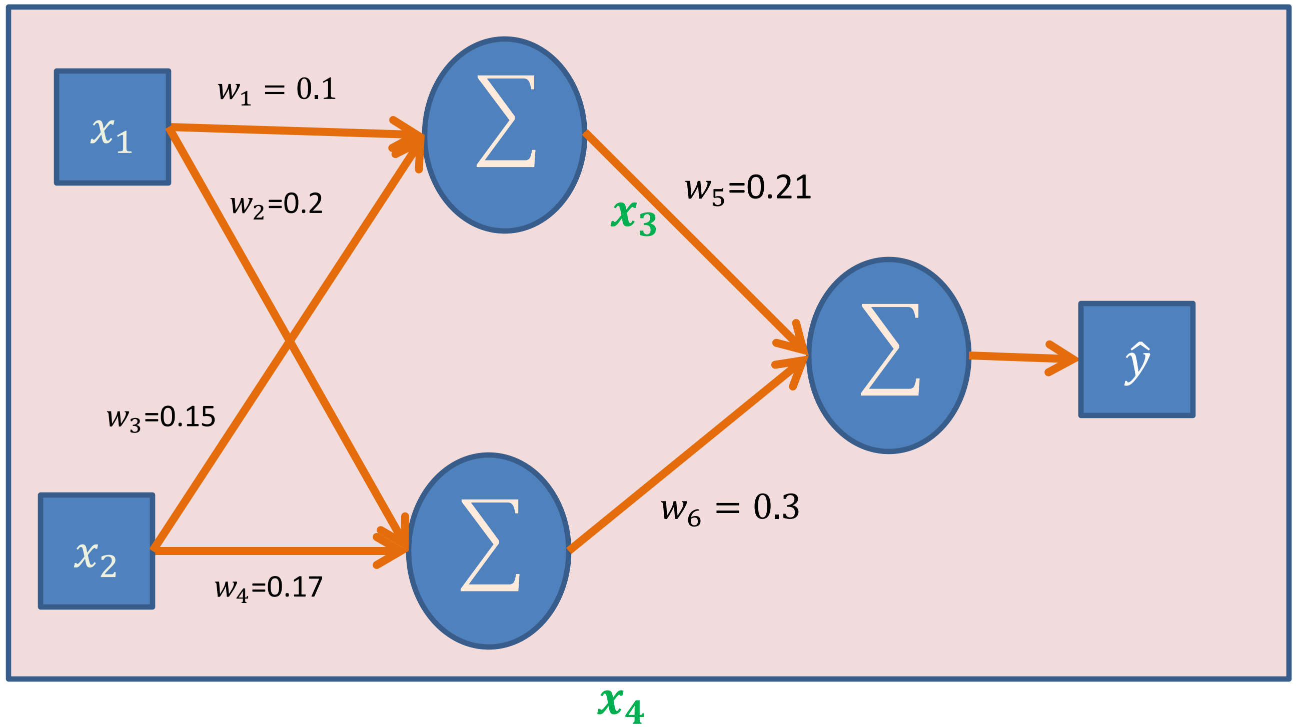
PyTorch Summary

- Data & Parameters as Tensors
- Moving from numpy to Tensor tp GPU
- PyTorch Model Class
- Autograd using backward()
- Built-in Loss Functions
- Built-in optimizers
- Much Simplified Training Loop
- Getting Ready for deep models training using large datasets?

Graded Home Task (backprop)

Training Data

x_1	x_2	y
-2	3	5
1	-2	-3



- **Perform weight update for one epoch using batch gradients and lr=0.1**
- **Do all steps manually first**
- **Write a PyTorch code using Autograd to verify your results**
- **Submit both manual calculation sheet & PyTorch code**