RLChina Reinforcement Learning Summer School

RLChina 2022

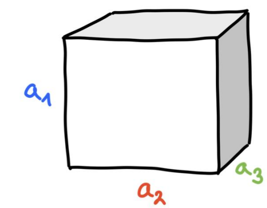
PyTorch Introduction

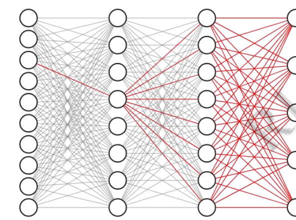
YanSong
CASIA

August 15, 2022

Outline

- Why Pytorch?
- Introduction to Pytorch tensor operations
- BackPropogation
- Training
- Examples and Homeworks
- Jidi plateform





What is PyTorch?

- Replacement for Numpy to use the power of GPUs
- Deep learning research platform that provides flexibility and speed
- Installation: *https://pytorch.org/get-started/locally/*
- Many alternatives based on similar principles

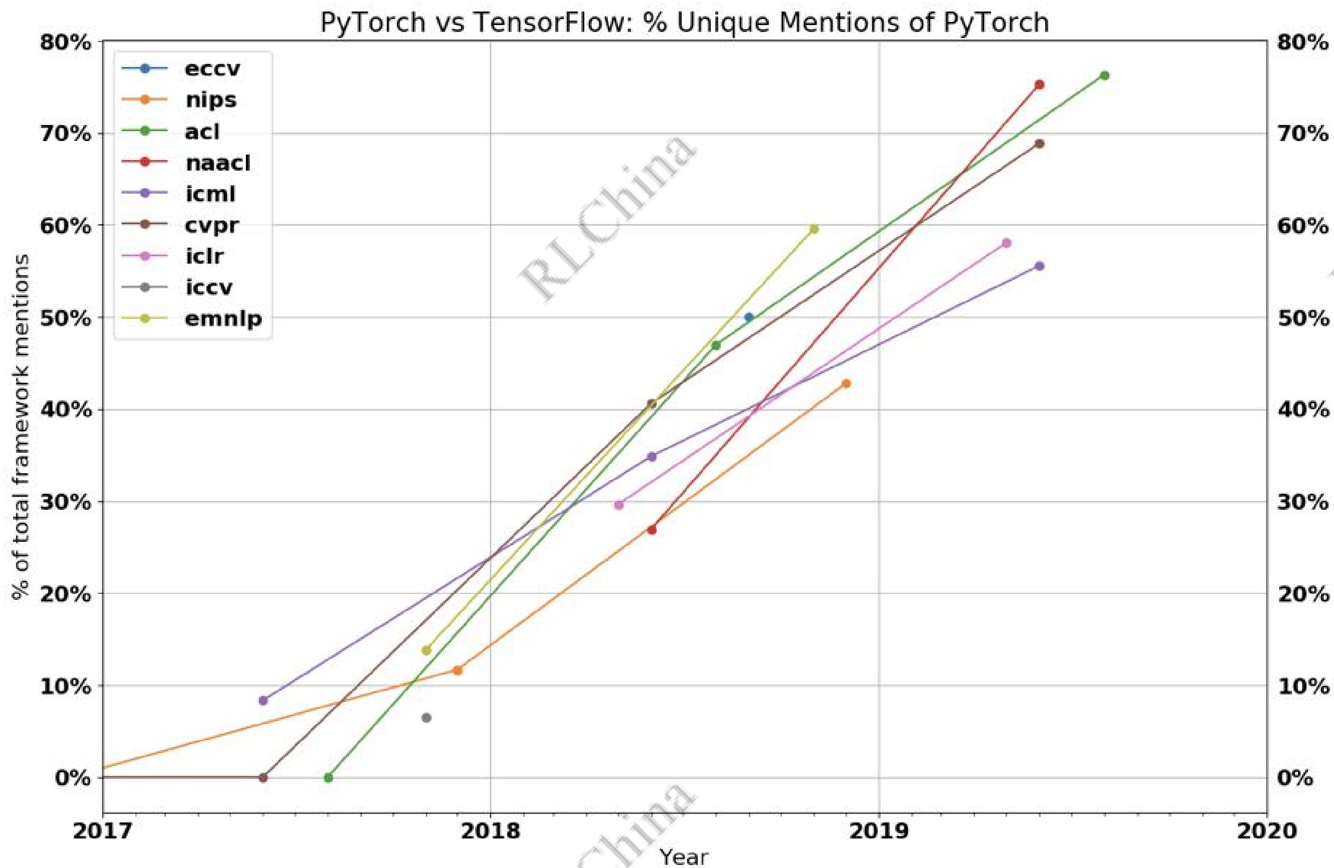
TensorFlow

theano

K Keras

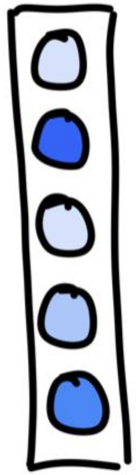
Why PyTorch?

- Dynamic computation graphs (by now also supported in TF Eager) are from our experience more intuitive for newcomers
- Easy debugging
- Data Parallelism

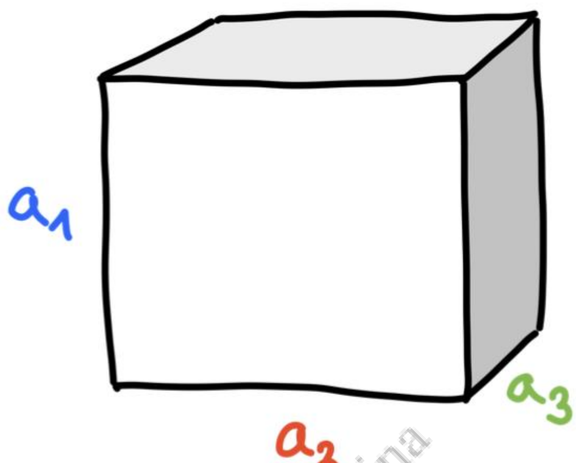


Scalars, Vectors, Matrices, Tensors and Basic operations

$$\vec{x} = \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$



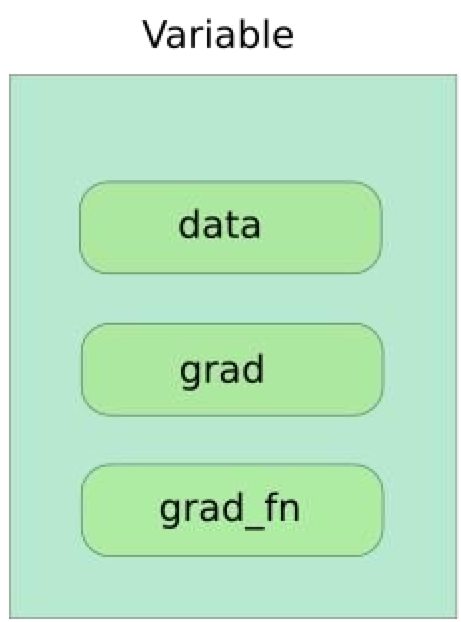
$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & \ddots & & x_{2n} \\ \vdots & & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

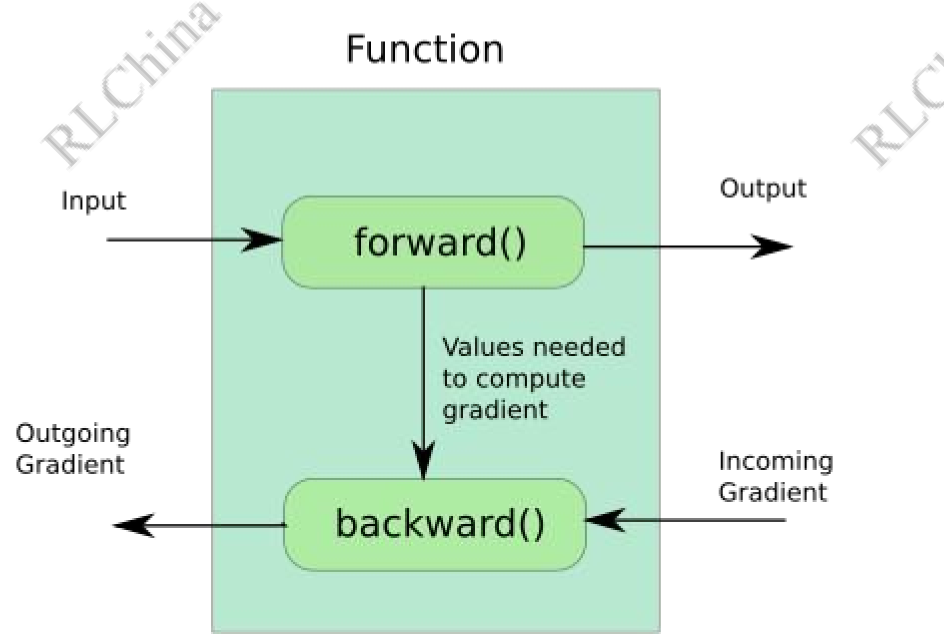


code on supplymentary notebook

PyTorch Variables

- A PyTorch Variable is a wrapper around a PyTorch Tensor, and represents a node in a computational graph.
- If x is a Variable then x.data is a Tensor giving its value
- x.grad is another Variable holding the gradient of x with respect to some scalar value

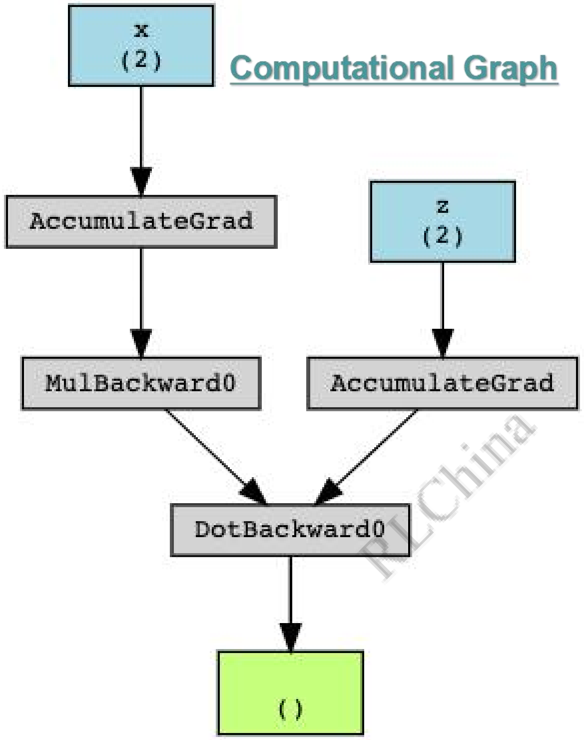




code on supplymentary notebook

PyTorch Autograd----Gradient

```
x = torch.tensor([-1.5, 1.2], requires_grad=True)
y = torch.tensor([1.0, -1.3])
z = torch.tensor([-2.0, 0.2], requires_grad=True)
r = x * y @ z
r.backward()
x.grad
> tensor([-2.0000, -0.2600])
```



Computational Graph

$$
\begin{aligned}
\frac{\partial}{\partial \mathbf{x}} r &= \frac{\partial}{\partial \mathbf{x}} (\mathbf{x} \odot \mathbf{y})^\top \mathbf{z} \\
&= \mathbf{z} \left[\frac{\partial}{\partial \mathbf{x}} (\mathbf{x} \odot \mathbf{y}) \right] + (\mathbf{x} \odot \mathbf{y}) \left[\frac{\partial}{\partial \mathbf{x}} \mathbf{z} \right] \\
&= \mathbf{z} \left[\mathbf{y} \left[\frac{\partial}{\partial \mathbf{x}} \mathbf{x} \right] + \mathbf{x} \left[\frac{\partial}{\partial \mathbf{x}} \mathbf{y} \right] \right] \\
&= \mathbf{z} \odot \mathbf{y} = \begin{bmatrix} -2.0 \\ 0.2 \end{bmatrix} \odot \begin{bmatrix} 1.0 \\ -1.3 \end{bmatrix} = \begin{bmatrix} -2.00 \\ -0.26 \end{bmatrix}
\end{aligned}
$$

BackPropogation derivation (Chain Rule)

$$
\mathbf{x}_{t+1} = \mathbf{x}_t - \alpha \frac{\partial}{\partial \mathbf{x}_t} r_t
$$

$$
\theta_{t+1} = \theta_t - \alpha \frac{\partial}{\partial \theta_t} L(X, Y; \theta_t)
$$

Gradient update

code on supplymentary notebook

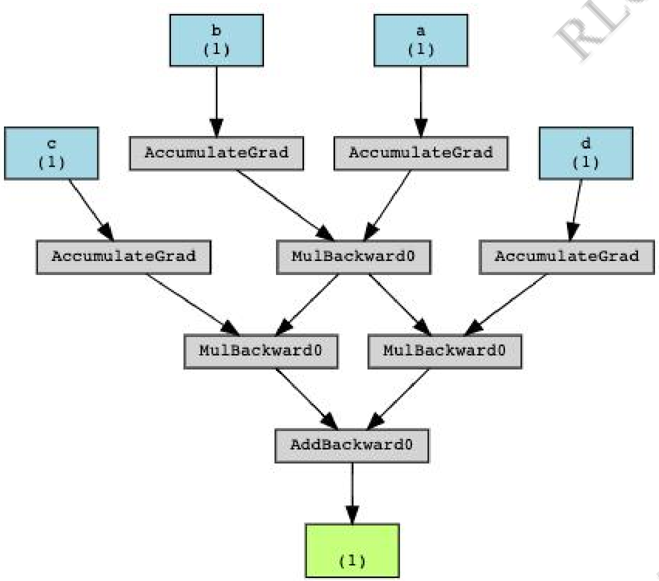
PyTorch Autograd----BackPropagation

- Backpropagation = Efficient Application of Chain Rule
- Chain Rule:

$$y = g(x)$$
$$z = f(y)$$
$$\frac{\partial z}{\partial x} = \frac{\partial z}{\partial y}\frac{\partial y}{\partial x} = f'(g(x))g'(x)$$

- Backprop


```
a = torch.rand(1, requires_grad=True)
b = torch.rand(1, requires_grad=True)
c = torch.rand(1, requires_grad=True)
d = torch.rand(1, requires_grad=True)

e = a * b
# compare to: f = c * a * b + d * a * b
f = c * e + d * e

f
> tensor([0.0994], grad_fn=<AddBackward0>)
```

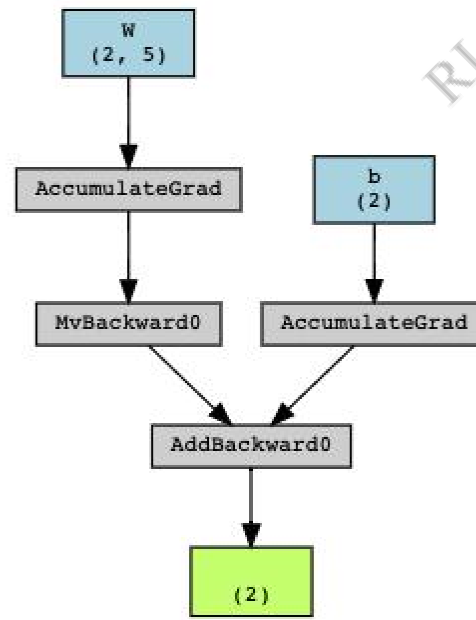


code on supplymentary notebook

PyTorch Modules

- All network components should inherit from nn.Module and override the forward method
- Using a module provides functionality:
  - Keeps track of trainable parameters
  - Allows you to easily swap between CPU and GPU (see .to(device))
- To register a variable tensor to the parameters of a module you need to wrap it using nn.Parameter

```python
class LinearModule(torch.nn.Module):
    def __init__(self, x_dim, y_dim):
        super(LinearModule, self).__init__()
        self.W = nn.Parameter(torch.randn(y_dim, x_dim, requires_grad=True))
        self.b = nn.Parameter(torch.randn(y_dim), requires_grad=True)
    def forward(self, x):
        return self.W @ x + self.b

# Some random input and output data
x = torch.randn(5)
y = torch.randn(2)

model = LinearModule(5, 2)

for param in model.parameters():
    print(param.size())

pred = model(x)
pred
```



code on supplymentary notebook

Loss Function

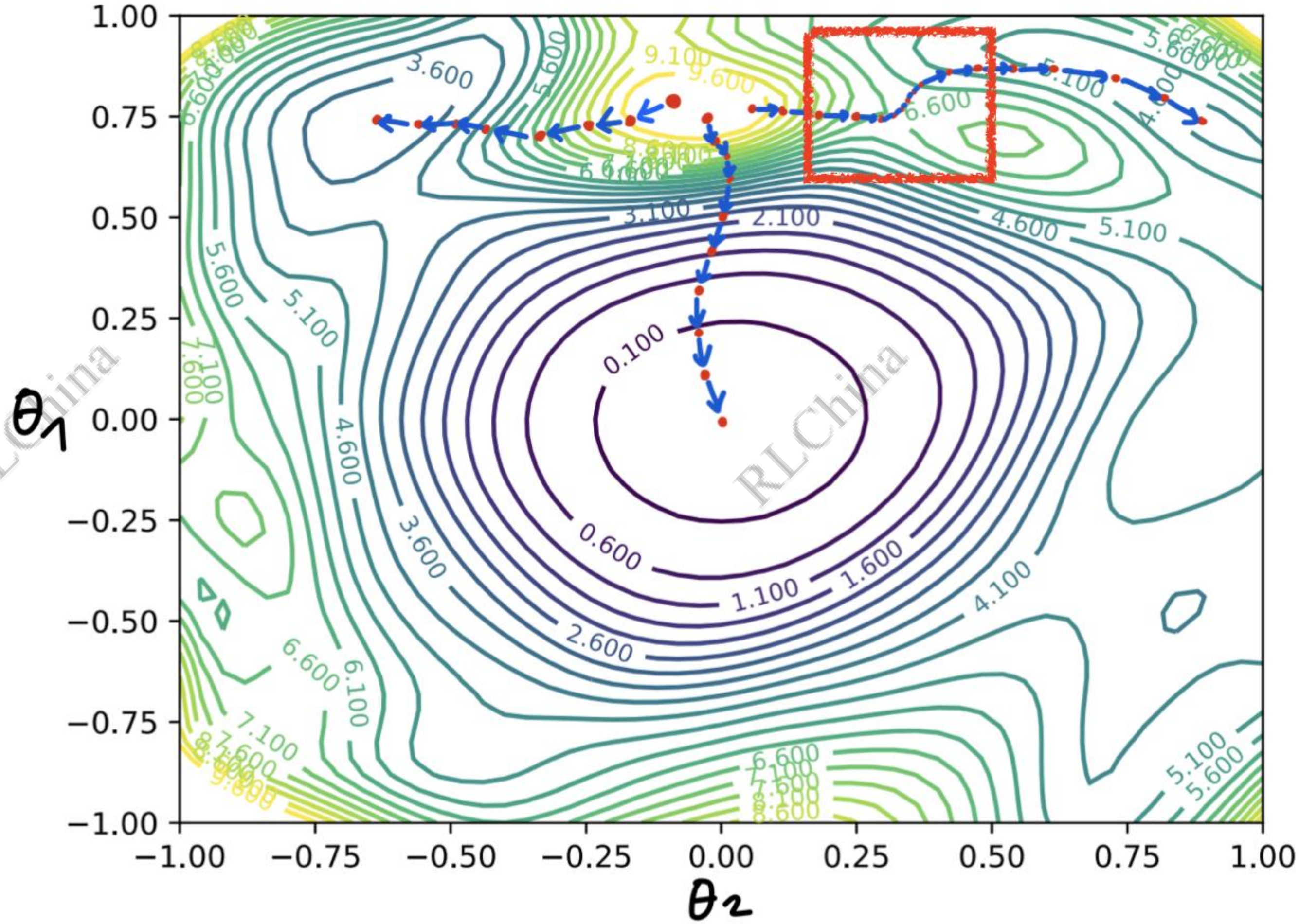
- Least squares [`nn.MSELoss`] $L(f_\theta(x), y) = \frac{1}{2}\left(f_\theta(x) - y\right)^2$
- Logistic [`nn.SoftMarginLoss`] $L(f_\theta(x), y) = \log(1 + exp(-yf_\theta(x)))$
- Hinge loss [`nn.MultiMarginLoss` / `nn.MultiLabelMarginLoss`]
$$L(f_\theta(x), y) = \max(0,\ 1 - yf_\theta(x))$$
- Cross-entropy [`nn.CrossEntropyLoss`]
$$L(f_\theta(x), y) = -\left[y\log(f_\theta(x)) - (1-y)\log(1 - f_\theta(x))\right]$$

For more see https://pytorch.org/docs/stable/nn.html#loss-functions

Training Loop

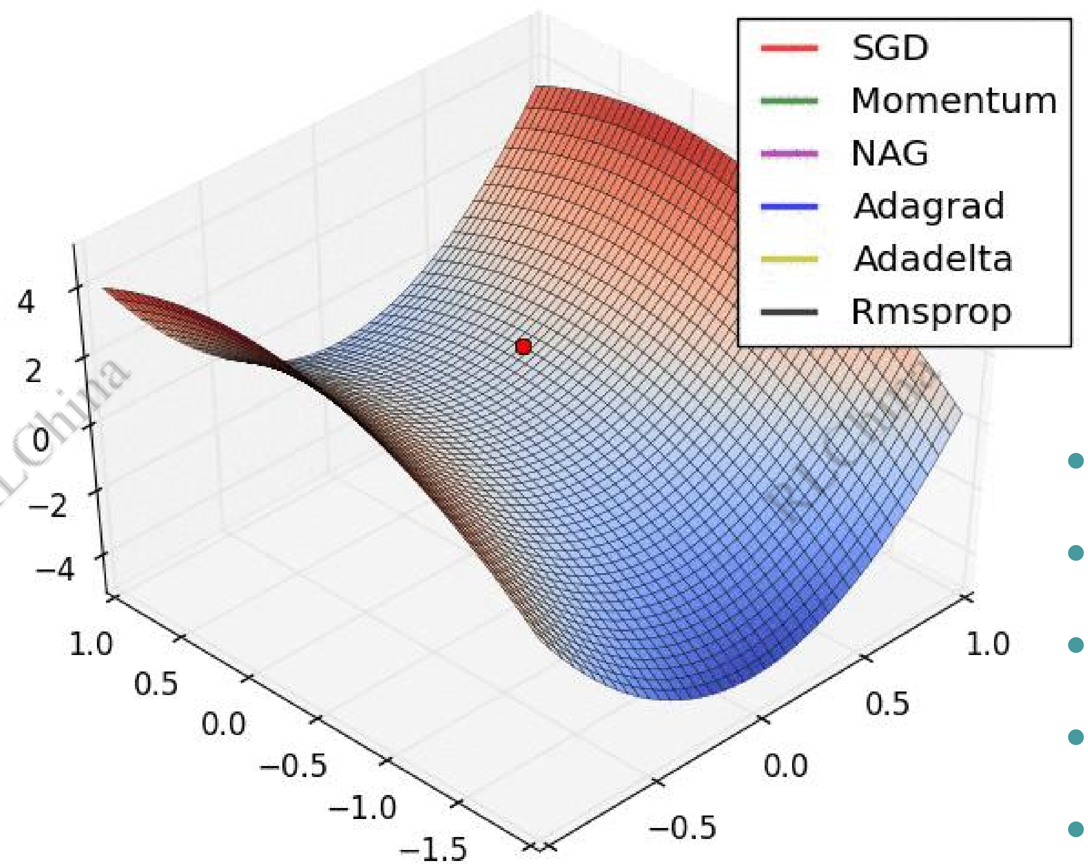
- Given model f_θ, initialize parameters θ (e.g. randomly)
- For number of epochs:
  - for number of iterations (i.e. number of batches in data):
    - sample batch of data $\boldsymbol{x}, \boldsymbol{y} \sim \boldsymbol{T}$
      - $\boldsymbol{x}$: input, $\boldsymbol{y}$: target output
    - run model forward $\boldsymbol{y}^* = \boldsymbol{f}_{\boldsymbol{\theta}}(\boldsymbol{x})$ and calculate loss $\boldsymbol{L}(\boldsymbol{y}^*, \boldsymbol{y})$
    - Calculate gradient of loss $\boldsymbol{\nabla}_\theta \boldsymbol{L}$ w.r.t parameter using backprop
    - Update parameters using optimiser, e.g. $\theta_{t+1} = \theta_t - \alpha \boldsymbol{\nabla} \theta_t$

Stochastics Gradient Descent



Li, Hao, et al. "Visualizing the loss landscape of neural nets." *NeurIPS*. 2018.

Optimiser



- torch.optim.SGD
- torch.optim.Adagrad
- torch.optim.Adadelta
- torch.optim.Adam
- torch.optim.RMSprop
- … and many more

PyTorch Training Loop Scaffold

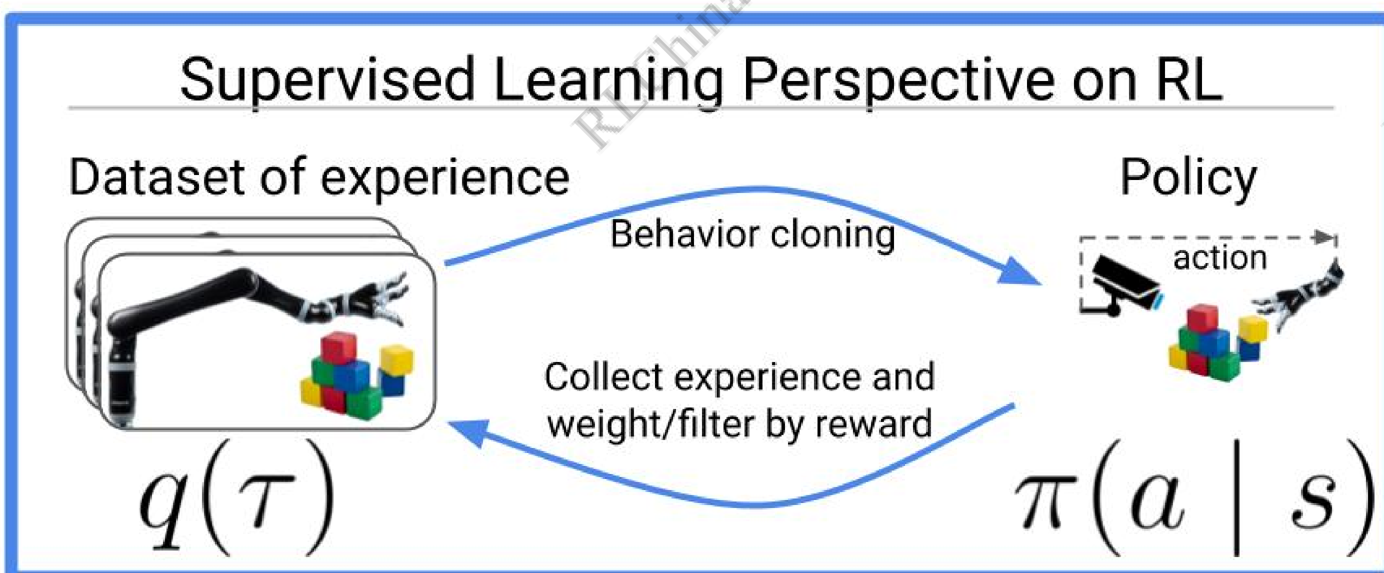
```
import torch.nn as nn
import torch.optim as optim

# Set a seed; your experiments should be reproducible!
torch.manual_seed(1)
# Load data
train, dev, test = ...
# Instantiate model
model = MyModel(...)
# Define loss function
loss_fn = nn.CrossEntropyLoss()
# Instantiate optimizer with a learning rate (lr)
optimizer = optim.SGD(model.parameters(), lr=0.1)

for epoch in range(10):  # 10 epochs in this example
  for i, batch in enumerate(train):  # assuming train is a generator that reshuffles data
    # Set gradients to zero
    optimizer.zero_grad()
    # Run forward
    y = model(batch)
    # Calculate loss
    loss = loss_fn(y, y_target)
    # Run backward to compute gradient of loss w.r.t. to model parameters
    loss.backward()
    # Perform one step of optimization
    optimizer.step()
    # Print diagnostics (e.g. loss or dev set performance)
    ...

# Evaluate on test
...
```

Examples

- Regression
- Classification
- Recurrent Neural Network
- Convolution Neural Network

A Decision Making example using PyTorch

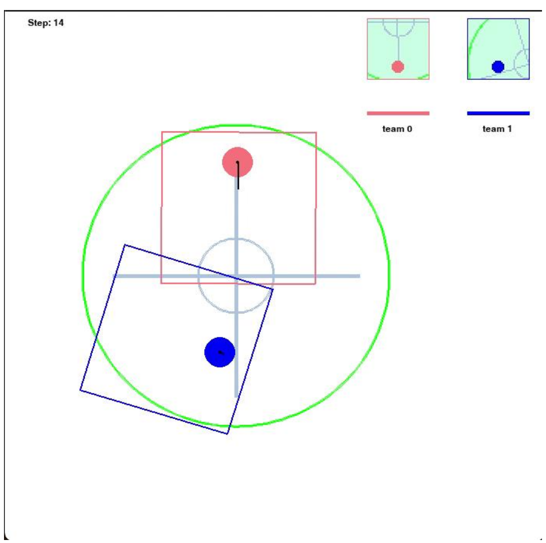
- Behaviour Cloning is a supervised learing way to tackle a decision making problem.
- Given expert experience, our goal is to imitate its behaviour as similar as possible with a parametrised policy
- therefore a good example to work with before get into the field of RL.



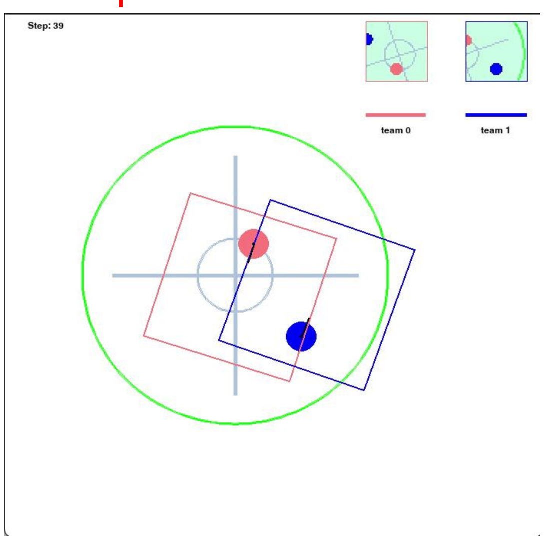
Homework: Behaviour Cloning on Olympics Wrestling

- two agent compete on stage and aim at pushing opponent out of bounds while avoid themselves falling out of the stage.
- Expert policy knows how to stay on the stage to survive.
- You are given with 10,000 collected expert data, battle begins!!!

random vs random



Expert vs random



code on supplymentary notebook

How to submit your policy ---- Jidi

1. Align your policy input and output with Jidi's evaluation, see run_log.py in ai_lib (*https://github.com/jidiai/ai_lib*) and /submission_example folder.
2. Create an accout on Jidi (*http://www.jidiai.cn/*)
3. Submit your policy

Thanks