Seminar Series Neural Networks for Finance

Lecture 3

Aim

Implementing neural networks with Pytorch

03-03-2022

Shuaiqiang Liu (shuaiqiang.liu@cwi.nl)

Introduction to Pytorch

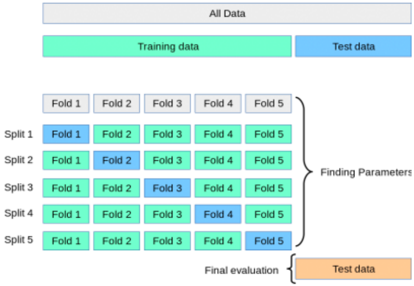
Training techniques(15 minutes), Pytorch basics (60 minutes) and code snippets (15 minutes).

- Training techniques
  - Hyperparemeter tuning
  - Early stopping
  - Regularization
- Basics of Pytorch
  - Data structure
  - Tensor operations
  - Computational graph
  - Optimization
- Common Modules in Pytorch
- Neural networks with Pytorch
  - Building a neural network
  - Training
  - Save and load the model

Hyperparameter tuning

The K-fold cross validation for model selection,

- Define the search space of hyperparameters, e.g., number of nodes and layers.
- Split the training data set into K different subsets.
- Select one subset as validation.
- Train a model on the remaining K-1 subsets.
- Calculate the model's metric on the validation part.
- Continue the above steps by exploring all subsets.
- Calculate the final metric by averaging over K cases.
- Explore the next set of hyper-parameters (e.g., different number of layers).
- Rank the model candidates using their final metric.

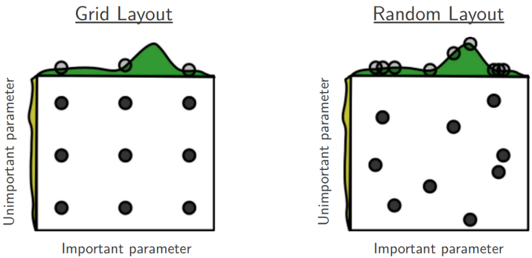


5-fold cross validation, @https://towardsdatascience.com/

Hyperparameter tuning

Methods of exploring the search space of hyperparameters,

- Grid search or random search,

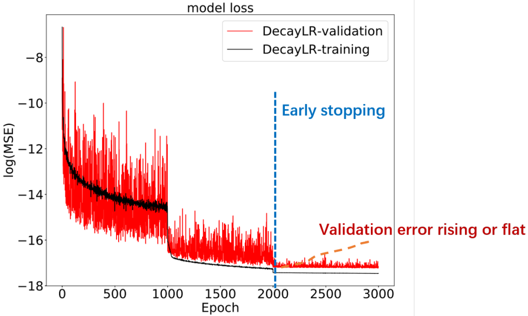


James & Yoshua: Random Search for Hyper-Parameter Optimization (2012)

- Hyperparameter Optimization, for instance, Bayesian optimization.

Early stopping

There are some training techniques to reduce the overfitting, e.g., Stopping training when a monitored metric fails to improve.



Validation-based early stopping.

Regularization

Adding a penalty term to the original loss function,

- L1 Regularization,

$$loss = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 + \lambda \sum_{j=1}^{m} |w_j|.$$

- L2 Regularization,

$$loss = \sum_{i=1}^{n} (\hat{y}_i - y_i)^2 + \lambda \sum_{j=1}^{m} w_j^2.$$

Pytorch

Pytroch is an open source machine learning framework, based on the programming language Python. Refer to `https://pytorch.org/` for the source code and tutorial.

A toy example

- The aim is to approximate the function $y = \cos(x)$, $\frac{1}{2}\pi \leq x \leq \frac{3}{2}\pi$, given some discrete data points (x_i, y_i), using a polynomial function,

$$\hat{y} = a + b * x + c * x^2 + d * x^3.$$

- First, defining a polynomial function,

$$\hat{y} = f(x; \boldsymbol{\theta}) = f(x; a, b, c, d).$$

- Second, defining the loss function

$$L = \sum_i (\hat{y}_i - y_i)^2 = \sum_i (f(x_i; a, b, c, d) - y_i)^2,$$

given input-output pairs $(x_i, y_i), i = 1, \ldots, n,$

Polynomial regression

- Third, solving the optimization problem

$$arg\min_{\boldsymbol{\theta}} L(\boldsymbol{\theta}|(\mathbf{x},\mathbf{y})),$$

a gradient descent algorithm is used as follows

$$\begin{cases} a \leftarrow a - \eta\dfrac{\partial L}{\partial a}, \\ b \leftarrow b - \eta\dfrac{\partial L}{\partial b}, \\ c \leftarrow c - \eta\dfrac{\partial L}{\partial c}, \\ d \leftarrow d - \eta\dfrac{\partial L}{\partial d}, \end{cases}$$

where the gradient of the loss w.r.t coefficients $\boldsymbol{\theta}$ is needed.

Step1: initialize

```python
import torch
import math

dtype  = torch.float
device = torch.device("cpu")
torch.manual_seed(0)

x = np.linspace(-0.5*math.pi, 1.5*math.pi, 2000)
y = np.cos(x)
x = torch.from_numpy(x)
y = torch.from_numpy(y)

# Create random Tensors for coefficients.
a = torch.randn((), device=device, dtype=dtype, requires_grad=True)
b = torch.randn((), device=device, dtype=dtype, requires_grad=True)
c = torch.randn((), device=device, dtype=dtype, requires_grad=True)
d = torch.randn((), device=device, dtype=dtype, requires_grad=True)
```

- input output
- coefficients

Step 2: Forward and backward pass

- define a polynomial function
- compute the loss function
- implement automatic differentiation

```
for t in range(25000):
    # compute yhat using operations on Tensors.
    yhat = a + b * x + c * x ** 2 + d * x ** 3

    # here loss is a Tensor of shape (1,)
    loss = (yhat - y).pow(2).sum()

    # Use autograd to compute the backward pass.
    loss.backward()
```

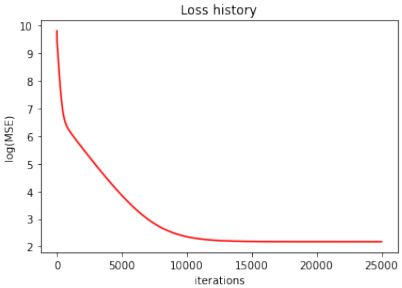
Step 3: Gradient descent

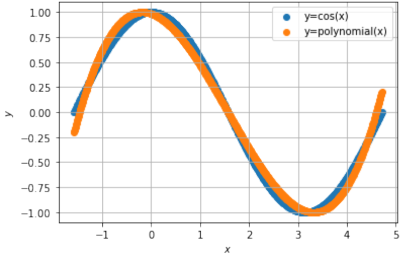
In order to minimize the loss function, the gradient descent algorithm is used as follows

$$\begin{cases} a \leftarrow a - \eta \dfrac{\partial L}{\partial a}, \\ b \leftarrow b - \eta \dfrac{\partial L}{\partial b}, \\ c \leftarrow c - \eta \dfrac{\partial L}{\partial c}, \\ d \leftarrow d - \eta \dfrac{\partial L}{\partial d}, \end{cases}$$

```
# Manually update coefficients
with torch.no_grad():
    a -= learning_rate * a.grad
    b -= learning_rate * b.grad
    c -= learning_rate * c.grad
    d -= learning_rate * d.grad
    # assign zeros to the gradients after updating weights
    a.grad = None
    b.grad = None
    c.grad = None
    d.grad = None
```

Results

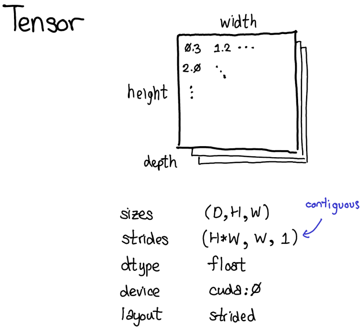




$$y = 0.985 - 0.166x - 0.440x^2 + 0.093x^3$$

What is a Tensor in Pytorch?

A Pytorch tensor has **two components, data and properties.**



http://blog.ezyang.com/2019/05/pytorch-internals/

elements in a Tensor:

$$A = \{a_1, a_2, a_3\}$$

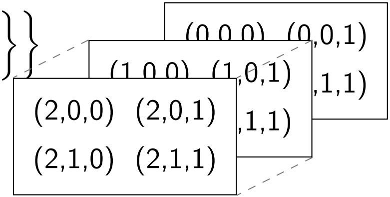
$$B = \begin{Bmatrix} a & b \\ c & d \end{Bmatrix}$$

$$C = \left\{ \begin{Bmatrix} a_1 & b_1 \\ c_1 & d_1 \end{Bmatrix}, \begin{Bmatrix} a_2 & b_2 \\ c2 & d_2 \end{Bmatrix}, \begin{Bmatrix} a_3 & b_3 \\ c_3 & d_3 \end{Bmatrix} \right\}$$

index in a Tensor:

$$\boxed{(0,1,2)}$$

$$\boxed{\begin{matrix} (0,0) & (0,1) \\ (1,0) & (1,1) \end{matrix}}$$



Create a Tensor within Pytorch

- from an existing array.
- from Numpy.
- from an existing tensor.
- given a specific size.

```
#from an existing array
data        = [[0.0, 1.0], [2.0, 3.0]]
tensor_data = torch.tensor(data)

# from Numpy
np_array    = np.array(data)
tensor_data = torch.from_numpy(np_array)

#from an existing tensor
x_ones = torch.ones_like(tensor_data)
x_rand = torch.rand_like(tensor_data)

#given a specific size
shape = (3,2)
ones_tensor = torch.ones(shape)
rand_tensor = torch.rand(shape)
```

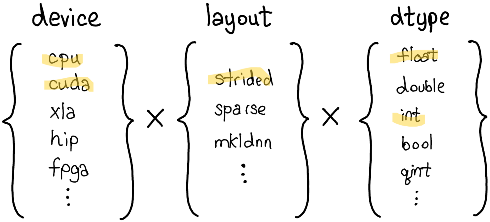
Properties of a Tensor

A Tensor has multiple properties,

1. torch.dtype,
   (float, int, . . .).
2. torch.device,
   ('cpu', 'cuda', . . .).
3. torch.layout
   (strided, sparse,. . .).

```
shape= (2,3)
x1 = torch.randn(shape,
                 dtype=torch.float32,
                 device=torch.device('cpu')
                 )
print(f"x1: \n {x1.dtype},
{x1.device},{x1.layout}")
```

tensor wrapper



Operations of Tensor

There are more than 100 tensor operations, including mathematical operations, linear algebra, slicing, and etc.
https://pytorch.org/docs/stable/torch.html

For example, on a single tensor

	Torch	Tensor
Transpose	.transpose(x)	x.transpose()
Reshape	.reshape(x)	x.reshape()
Mean	.mean(x)	x.mean()
Deviation	.std(x)	x.std()
Sum	.sum(x)	x.sum()

```python
a = torch.randn(1, 2, 3, 4)
print(a.size())

# Swaps 2nd and 3rd dimension
b = a.transpose(1, 2)
print(b.size())

c = torch.transpose(a, 1, 2)
print(c.size())
```

Math operations of Tensor

On multiple tensors

	Torch	symbol
Add	.add(x,y)	x+y
Subtract	.sub(x,y)	x-y
Multiply	.mul(x,y)	x*y
Divide	.div(x,y)	x/y
Concatenate	.cat(x,y)	-

```
m,n =3,3
x = torch.zeros((m,n)) +2
y = torch.zeros((m,n)) +1

## element wise
z_add    = x+y
z_minus  = x-y
z_times  = x*y
z_divide = x/y
```

Math operations of Tensor

Two types of multiplication

- Matrix multiplication

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \end{bmatrix} \mathbf{x} \begin{bmatrix} 6 & 9 \\ 7 & 10 \\ 8 & 11 \end{bmatrix} = \begin{bmatrix} 23 & 32 \\ 86 & 122 \end{bmatrix}$$

- Element-wise multiplication.

```
m,n =2,2
x = torch.zeros((m,n)) +2
y = torch.zeros((m,n)) +1

## matrix multiplication
z   = torch.matmul(x,y)
zmm = torch.mm(x,y)

## elementwise multiplication
zz  = x*y
zzem=torch.mul(x,y)
```

Element-wise multiplication

Broadcasting if the shapes of two tensors do not match.
$z(i) = x(i) \times y(i), i = 1, \ldots, \max(\text{x.size(dim=0),y.size(dim=0)})$

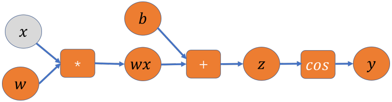
Question: the size of z_1 and z_2?

```
m,n =1,2
x = torch.rand((m,n))
y = torch.rand((n,m))

## elementwise multiplication
z1 = x*y
z2 = y*x
```

Computational Graph

$y = \cos(wx + b)$

- The forward pass:

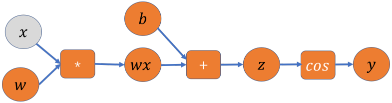



```
x=torch.tensor(2.0,requires_grad=False)
w=torch.tensor(1.0,requires_grad=True)
b=torch.tensor(3.0,requires_grad=True)

# forward
wx   = w*x
z    = wx +b
y    = torch.cos(z)
```

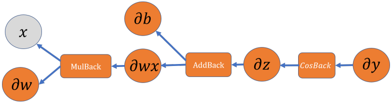
Automatic differentiation

$$y = \cos(wx + b)$$

- The forward pass:



- The backward pass:

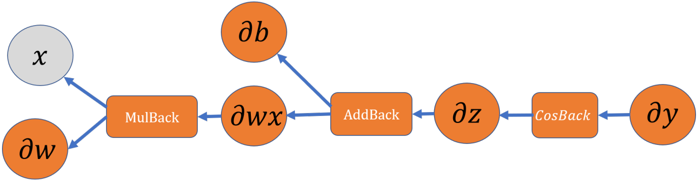



```python
x=torch.tensor(2.0,requires_grad=False)
w=torch.tensor(1.0,requires_grad=True)
b=torch.tensor(3.0,requires_grad=True)

# forward
wx   = w*x
z    = wx +b
y    = torch.cos(z)

# backward
y.backward()
# Note that y should be a scalar.

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

Automatic differentiation using .backward().

A step-by-step way of back propagation

Computing intermediate partial derivatives, $(\partial z, \partial wx, \partial b, \partial w)$, using lower-level functions, `.grad_fn` and `.next_functions`



In [1]:
```
import torch
x=torch.tensor(0.5,requires_grad=False)
w=torch.tensor(1.0,requires_grad=True)
b=torch.tensor(0.5*torch.pi-0.5,requires_grad=True)
wx  = w*x
z   = wx+b
y   = torch.cos(z)
```

In [2]:
```
# fetch the backward function
CosBack = y.grad_fn
dy = torch.tensor(1.)  # dy=1.0
dz = CosBack(dy)
print('CosBack function:',CosBack, '\n dz=',dz)
```

```
CosBack function: <CosBackward0 object at 0x00000255C4B4CB20>
 dz= tensor(-1., grad_fn=<MulBackward0>)
```

A step-by-step way of back propagation

In [3]:
```
# next_functions of back_cos
AddBack = CosBack.next_functions[0][0]
dwx, db  = AddBack(dz)
print('AddBack function:',AddBack)
print('dwx=',dwx,'\n db=',db)
```

```
AddBack function: <AddBackward0 object at 0x00000255C4B4C19
dwx= tensor(-1., grad_fn=<MulBackward0>)
 db= tensor(-1., grad_fn=<MulBackward0>)
```

In [4]:
```
MulBack= CosBack.next_functions[0][0].next_functions[0][0]
dw = MulBack(dwx)
print(' dw=',dw)
```

```
 dw= (tensor(-0.5000, grad_fn=<MulBackward0>), None)
```

Gradients associated with a Tensor

Two kinds of gradient properties:

- .grad, *a value*
- .grad_fn, *a function*

```
x=torch.tensor(0.5,requires_grad=False)
w=torch.tensor(1.0,requires_grad=True)
b=torch.tensor(0.5*torch.pi-0.5,requires_grad=True)

wx  = w*x
z   = wx+b
y   = torch.cos(z)

print(x.grad)
print(y.grad_fn)
```

"Each grad_fn stored with the tensors allows one to walk the computation all the way back to its inputs with its next_functions property. "

Recap the polynomial regression

With these basic Tensor operations, we can create machine learning models, for example, a polynomial regression.

```
import torch
import math
dtype  = torch.float
device = torch.device("cpu")
torch.manual_seed(0)

x = torch.from_numpy(x)
y = torch.from_numpy(y)
a = torch.randn((), device=device, dtype=dtype, requires_grad=True)
b = torch.randn((), device=device, dtype=dtype, requires_grad=True)
c = torch.randn((), device=device, dtype=dtype, requires_grad=True)
d = torch.randn((), device=device, dtype=dtype, requires_grad=True)

learning_rate = 3e-7
for t in range(25000):
    yhat = a + b * x + c * x ** 2 + d * x ** 3
    loss = (yhat - y).pow(2).sum()
    loss.backward()
    with torch.no_grad():
        a -= learning_rate * a.grad
        b -= learning_rate * b.grad
        c -= learning_rate * c.grad
        d -= learning_rate * d.grad
        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 ')
```

Next we turn to ANNs.

- Define an ANN function as follows,

$$\hat{y} = f(x; \boldsymbol{\theta}) := f(x; \mathbf{W}, \mathbf{b}),$$

where $\mathbf{W}, \mathbf{b}$ are weights and biases respectively.

- Compute the loss function

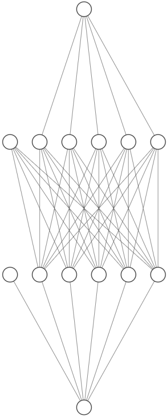
$$L = \sum_i (\hat{y}_i - y_i)^2 = \sum_i (f(x_i; \mathbf{W}, \mathbf{b}) - y_i)^2,$$

given input-output data pairs $(x_i, y_i), i = 1, \ldots, n$.

- Use gradient descent algorithms to minimize the loss function,

$$\begin{cases} \mathbf{W} \leftarrow \mathbf{W} - \eta \dfrac{\partial L}{\mathbf{W}}, \\ \mathbf{b} \leftarrow \mathbf{b} - \eta \dfrac{\partial L}{\mathbf{b}}. \end{cases}$$

Neural networks within Pytorch




```python
def sigmoid(x): return 1.0/(1.0+torch.exp(-x))

class NeuralNetworkLow(nn.Module):
    def __init__(self,in_features=1,out_features=1,neurons=5,
                actvfun=sigmoid):
        super(NeuralNetworkLow, self).__init__()
        ## input layer
        self.w1 = torch.rand(in_features, neurons, requires_grad=True)
        self.b1 = torch.zeros(neurons, requires_grad=True)
        ##  hidden layer
        self.w2 = torch.rand(neurons, neurons, requires_grad=True)
        self.b2 = torch.zeros(neurons, requires_grad=True)
        ## output layer
        self.w3 = torch.rand(neurons, out_features,requires_grad=True)
        self.b3 = torch.zeros(out_features, requires_grad=True)
        ## activation function
        self.act_fun = actvfun

    def forward(self,x):
        z = torch.matmul(x, self.w1)+self.b1
        z = self.act_fun(z)
        z = torch.matmul(z, self.w2)+self.b2
        z = self.act_fun(z)
        z = torch.matmul(z, self.w3)+self.b3
        return z   ## output layer
```

Training

```
model     = NeuralNetworkLow(NofIn,NofOut,NofNodes,sigmoid)

for b in range(max_batches):
    curr_bat = np.random.choice(n_items, bat_size,
           replace=False)
    batchX = x_train[curr_bat]
    batchY = y_train[curr_bat].view(bat_size,1)

    # evaluate the function
    batchy_pred = model.forward(batchX)
    # compute the loss function MSE
    loss_obj = (batchy_pred-batchY).pow(2).sum()
    # back propagation
    loss_obj.backward()

    with torch.no_grad():
         model.w1 -=learning_rate*model.w1.grad
         model.b1 -=learning_rate*model.b1.grad
         model.w2 -=learning_rate*model.w2.grad
         model.b2 -=learning_rate*model.b2.grad
         model.w3 -=learning_rate*model.w3.grad
         model.b3 -=learning_rate*model.b3.grad
         # manually clear the gradient
         model.w1.grad = None
         model.b1.grad = None
         model.w2.grad = None
         model.b2.grad = None
         model.w3.grad = None
         model.b3.grad = None
```

Higher level functions in Pytorch

The module "torch.nn" has higher level APIs,

- nn.Linear()
- nn.Sequential()
- nn.Sigmoid()
- nn.Conv2d()
- ...

```python
import torch.nn as nn
class NeuralNetworkHigh(nn.Module):

    def __init__(self, in_features=1,out_features=1,
                               neurons=2, actvfun = nn.Sigmoid):
        super(NeuralNetworkHigh, self).__init__()
        ## the input layer
        self.layer1=nn.Linear(in_features, neurons)
        ## hidden layer
        self.layer2=nn.Linear(neurons, neurons)
        ## hidden layer
        self.layer3=nn.Linear(neurons, out_features)
        ## activation function
        self.actvfun=actvfun()

        ## put all the layers together
        self.fnn = nn.Sequential(self.layer1,self.actvfun,
                        self.layer2,self.actvfun,self.layer3)
    def forward(self, x):
        y = self.fnn(x)
        return y
```

Optimizers in Pytorch

The module "torch.optim" provides multiple numerical optimizers (SGD, Adam, etc).
One needs to use `nn.parameters()` to register a Tensor as trainable model parameters before implementing an optimzier.

```
optimizer = torch.optim.SGD(model.parameters(),lr=0.01)

class NeuralNetworkLow(nn.Module):
    def __init__(self,in_features=1,out_features=1,neurons=5,
               actvfun=sigmoid):
        super(NeuralNetworkLow, self).__init__()
         #initialize a tensor
        self.w1 = torch.rand(in_features, neurons, requires_grad=True)
         #register it as trainable model  parameters
        self.w1 = torch.nn.parameters(self.w1)

         #initialize and register
        self.b1=torch.nn.parameters(torch.zeros(neurons, requires_grad=True))
        ......
```

Implementing optimziers

There are two possible ways to implement an optimizer within Pytorch.

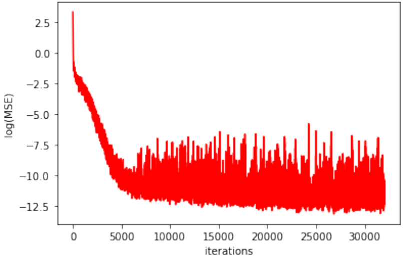
- a `step()` method to update the parameters once.

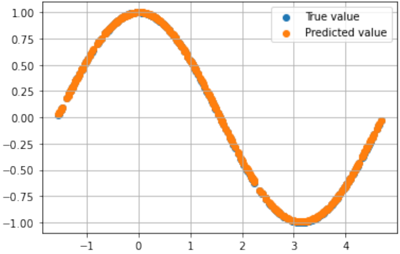
```
model= NeuralNetworkHigh()
loss = torch.nn.MSELoss()
for b in range(max_batches):
    ....
    optimizer.zero_grad()
    batchy_pred = model.forward(batchX)
    loss_obj = loss(batchy_pred, batchY)
    loss_obj.backward()
    # update the parameters
    optimizer.step()
```

- a closure method when reevaluating the function multiple times

```
for input, target in dataset:
    def closure():
        optimizer.zero_grad()
        output = model(input)
        loss = loss_fn(output, target)
        loss.backward()
        return loss
    optimizer.step(closure)
```

Results of training ANNs





Training a neural network to approximate $y = \cos(x)$.

View neurons and their gradients

- For low-level functions,

```
print(model.w2)
print(model.w2.grad)
```

- For high-level functions,

```
print(model.layer2.weight)
print(model.layer2.weight.grad)
```

Saving and loading models

Two ways of saving a Pytorch model,

- The model structure and learned parameters (e.g., model inference):

```
model_file = 'model.pth'
torch.save(model, model_file) # save model to a file
#...
model_loaded = torch.load(model_file) # load model from a file
```

- The learnable parameters and Optimizer objects (*torch.optim*):

```
PATH = 'model.pt'
# suppose the last training status as follows
Loss = 0.01
torch.save(model.state_dict(), PATH) # save the status to a dictionary
# or save more status
torch.save({'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'Loss': Loss}, PATH)
#...
model.load_state_dict(torch.load(PATH))
model.eval()
```

Question: Which one is preferred for training a large model?

Restart from a checkpoint

Resuming model training,

```
model = NeuralNetworkHigh()
optimizer = optim.SGD(net.parameters(), lr=0.001)
# load the status
checkpoint = torch.load(PATH)
Loss       = checkpoint['Loss']
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])

# evaluate the pre-trained model
model.eval()
# - continue training -
model.train()
```

Graphic Processing Unit (GPU)

A GPU contains hundreds to thousands of individual cores, suitable for accelerating neural networks (parallel computing).

- Check if the computer system has a CUDA enabled GPU.

```
if torch.cuda.is_available():
    print(f"CUDA version: {torch.version.cuda}")
```

- Transfer tensor data to a GPU.

```
x = x.to(device='cuda')
y = y.to(device='cuda')
```

- Transfer the model to a GPU.

```
model  = model.to(device='cuda')
y_pred = model(x)
```

The rest of the workflow on GPUs and CPUs is the same.

Regularization with Pytorch

- L1 Regularization,

$$loss = \sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda \sum_{j=1}^{m} |w_j|.$$

- L2 Regularization,

$$loss = \sum_{i=1}^{n}(\hat{y}_i - y_i)^2 + \lambda \sum_{j=1}^{m} w_j^2.$$

```
loss=loss_fun(y,y_pred)
lambda_coef = 0.001
l2_norm = sum(p.pow(2.0).sum() for p in model.parameters())
#l1_norm = sum(p.abs().sum() for p in model.parameters())
loss = loss + lambda_coef * l2_norm

optimizer.zero_grad()
loss.backward()
optimizer.step()
```