Seminar Series Neural Networks for Finance

Lecture 4

Aim

Implementing neural networks with Pytorch

29-02-2024

Zhipeng Huang (z.huang1@uu.nl)

Outline

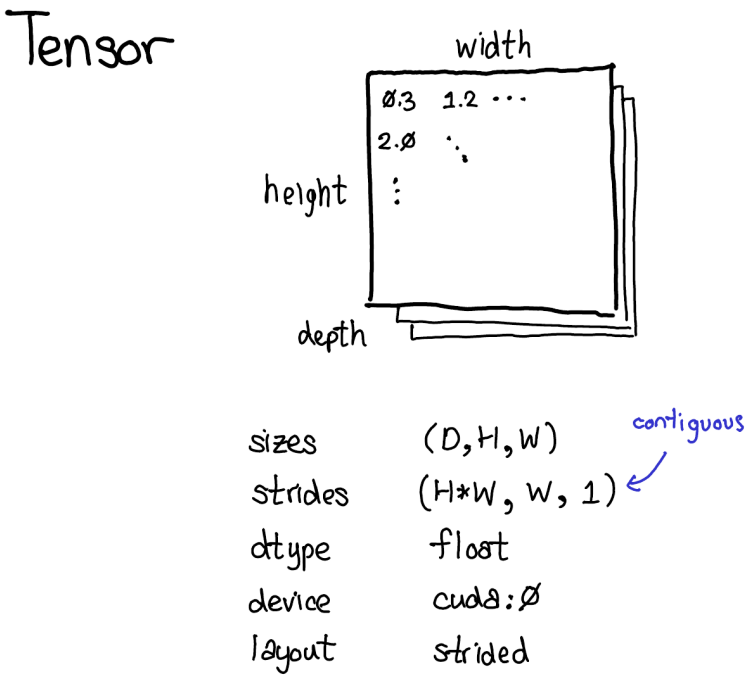
1. Basics of Pytorch
   - Pytorch tensors
   - Tensors operations and broadcast
   - Computational graph and automatic differentiation
   - Example: polynomial approximation
2. Building a neural network in Pytorch
   - Construct networks: low level APIs
   - Model training and optimization
   - Construct networks: high level APIs
   - Further issues
3. Training Techniques
   - Hyperparameter tuning
   - Early stopping
   - Regularization

Pytoch

Pytroch is an open-source machine learning framework, based on the programming language Python.
Visit the website https://pytorch.org/ for tutorials, documents, and source codes.

What is a Tensor in Pytorch?

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



Examples of Tensors

elements of a Tensor:

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

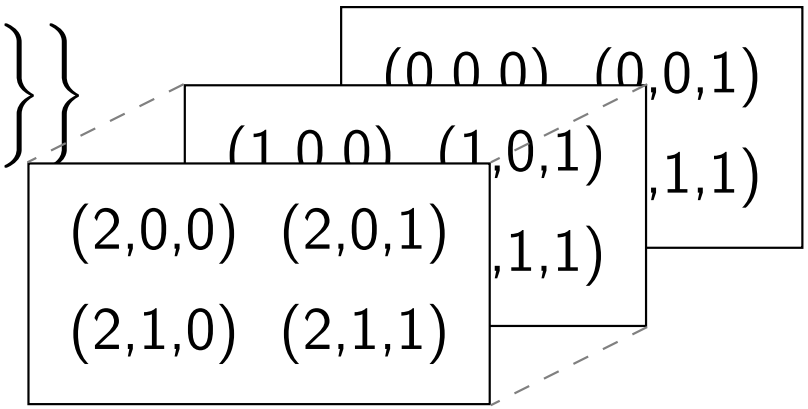
$$B = \begin{Bmatrix} a & b \\ c & d \end{Bmatrix}$$

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

index of a Tensor:

(0,1,2)

(0,0) (0,1)
(1,0) (1,1)



Creation of Tensors

Common ways to create tensors:

- from a list;
- from a Numpy array;
- generate from a given size/shape.

```
# from a list
data = [[0.0, 1.0], [2.0, 3.0]]
tensor_data_0 = torch.tensor(data)

# from a Numpy array
np_array = np.array(data)
tensor_data_1 = torch.tensor(np_array)
tensor_data_2 = torch.from_numpy(np_array)

# generate from a given size/shape
size = [3, 2]
ones_tensor = torch.ones(size)
rand_tensor = torch.rand(size)
x_ones = torch.ones_like(tensor_data)
x_rand = torch.rand_like(tensor_data)
```

Attributes of Tensors

Each Pytorch Tensor has attributes **torch.dtype**, **torch.device**, and **torch.layout**. Some common attributes given below:

1. torch.dtype, (float, int, . . .).
2. torch.device, ('cpu', 'cuda', 'mps').
3. torch.layout (strided, sparse,. . .).
4. torch.shape returns the shape/size of the tensor, the same as torch.size()

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

Operations on Tensors

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

Operations on a single tensor:

Pytorch code
.transpose(x, dim0, dim1)
.sum(x, dim=None, keepdim=False)
.mean(x, dim=None, keepdim=False)
.std(x, dim=None, keepdim=False)
.reshape(x, shape)
.squeeze(x, dim=None)
.unsqueeze(x, dim)

```
# Swaps the 1st and 2nd dimensions
x1 = torch.randn(1, 2, 3)
x1_a = torch.transpose(x1, 1, 2)
print(x1_a.shape)

# torch.transpose shares data
x1_a[0, 0, 1] = 1.
print(x1_a)
print(x1)
```

Operations on Tensors

Suppose the two tensors x and y here are of the same shape/size. Then the following functions carry out element-wise operations.

Operations on multiple tensors:

	PyTorch	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

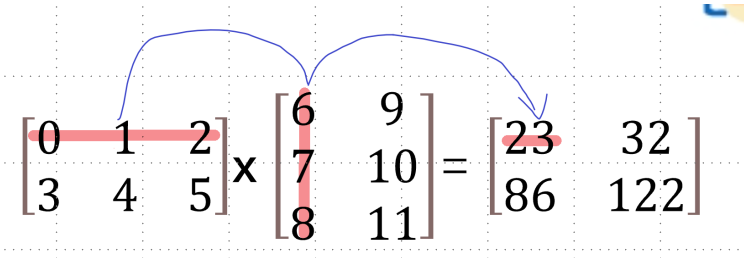
```
x = torch.tensor([[1.0, 1.0],
                  [2.0, 3.0]])
y = torch.tensor([[0.1, 0.1],
                  [0.2, 0.3]])

# elementwise operations
z_add = x + y
z_sub = x - y
z_mul = x * y
z_div = x / y
```

Operations on Tensors

Suppose the tensors X and Y here are of dimensions not greater than 2, with compatible sizes such that usual matrix multiplication holds.

Usage of torch.matmul :

- Matrix-matrix multiplication



- Matrix-vector multiplication
- Vector-vector multiplication
- These are NOT element-wise operations.

```
X = torch.tensor([[0., 1., 2.],
                  [3., 4., 5.]])
Y = torch.tensor([[6., 9.],
                  [7., 10.],
                  [8., 11.]])
Z_matmul = torch.matmul(X, Y)
Z = X @ Y    # same as .matmul
print(Z_matmul)
print(Z)

y = torch.tensor([6., 7., 8.])
z_1 = X @ y
z_2 = y @ torch.transpose(X, 0, 1)
print(z_1)
print(z_2)

y_dotprod = y @ y
print(y_dotprod)
```

If the sizes do not match: Broadcasting

Broadcasting allows the performing of operations (if they support broadcast) on tensors that are not of the same size.
Two tensors x and y are "broadcastable" if the following rules hold:

- Each tensor has at least one dimension (with exceptions);
- Starting from the last dimension, the dimension sizes must either be equal, one of them is 1, or one of them does not exist.

If they are "broadcastable", then:

- If their dimensions are not equal, prepend 1 to the dimensions of the tensor with fewer dimensions to make them equal length.
- Then, for each dimension size, the resulting dimension size is the max of the sizes of x and y along that dimension.

If the sizes do not match: Broadcasting

Broadcastable and non-broadcastable examples:

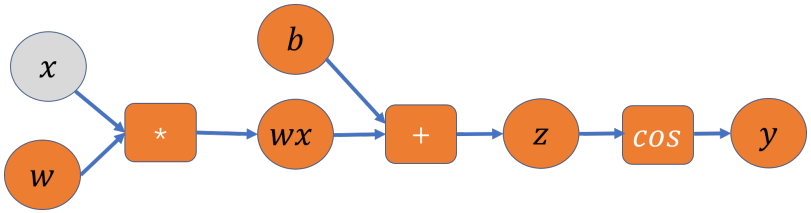
Question: What is the sizes of z?

```
#### Example 1
x = torch.tensor([[1.0], [0.1]])
y = torch.tensor([[4.0, 5.0]])
z = x * y
print(x * 2.)
print(x * torch.tensor(2.))

#### Example 2
x = torch.tensor([[1.0, 0.0], [0.0, 1.0]])
y = torch.randn([3.0, 2.0, 2.0])
z = x @ y
print(y)
print(z)
print(x @ torch.tensor(2.0))   # report errors
print(x @ 2.0)             # report errors
```

Computational Graph

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

- The forward pass:

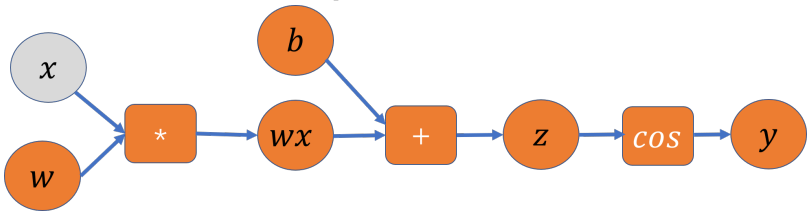



```
x=torch.tensor(2.0,requires_grad=False)
w=torch.tensor(1.0,requires_grad=True)
b=torch.tensor(3.0,requires_grad=True)

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

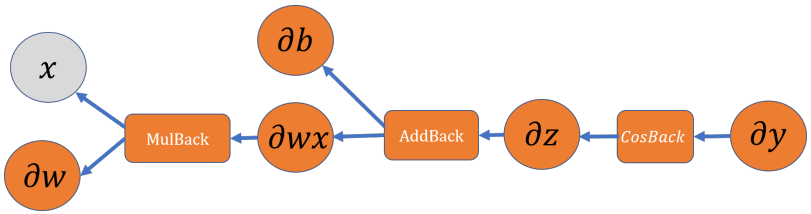
Automatic differentiation

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

- The forward pass:



- The backward pass:

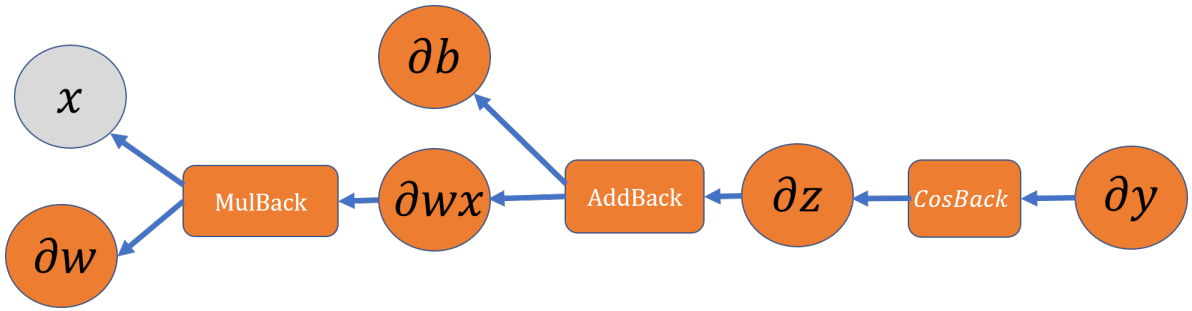



```
x=torch.tensor(2.0,requires_grad=False)
w=torch.tensor(1.0,requires_grad=True)
b=torch.tensor(3.0,requires_grad=True)

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

# backward pass for the gradients
y.backward()

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

To get the gradients of y w.r.t w and z, we use .backward() for the automatic differentiation.

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`




```
1 import torch
2
3 x = torch.tensor(0.5, requires_grad = False)
4 w = torch.tensor(1.0, requires_grad = True)
5 b = torch.tensor(0.5*torch.pi-0.5,requires_grad=True)
6 wx = w * x
7 z = wx + b
8 y = torch.cos(z)
```

```
1 ### fetch gradients backward using grad_fn
2 grad_cos_back = y.grad_fn
3 dy_dy = torch.tensor(1.0)
4 dy_dz = grad_cos_back(dy_dy)
5 print('grad_cos_back',grad_cos_back,'\n dy/dz=',dy_dz)
```

```
grad_cos_back  <CosBackward0 object at 0x789d40ca1660>
 dy/dz= tensor(-1., grad_fn=<MulBackward0>)
```

A step-by-step way of back propagation

```
# use next_functions of grad_fn to walk backward
grad_add_back = y.grad_fn.next_functions[0][0]
dy_dwx, dy_db = grad_add_back(dy_dz)
print('grad_add_back ',grad_add_back)
print('dy/dwx=',dy_dwx,'\n dy/db=',dy_db)
```

```
grad_add_back  <AddBackward0 object at 0x789d40ca27a0>
dy/dwx= tensor(-1., grad_fn=<MulBackward0>)
 dy/db= tensor(-1., grad_fn=<MulBackward0>)
dy/dw= (tensor(-0.5000, grad_fn=<MulBackward0>), None)
```

```
grad_mul_back=y.grad_fn.next_functions[0][0].next_functions[0][0]
dy_dw = grad_mul_back(dy_dwx)
print('dy/dw=',dy_dw)
```

```
dy/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. "

Example: polynomial approximation of $\cos(x)$

- Goal: Approximating $y = \cos(x)$ for $\frac{1}{2}\pi \leq x \leq \frac{3}{2}\pi$ by the following polynomial function

$$\hat{y} = a + bx + cx^2 + dx^3.$$

  where the samples (x_i, y_i) of $y = \cos(x)$ is given, and coefficients a, b, c and d are unknown.

- First, we define the model f with parameter $\boldsymbol{\theta} \equiv \{a, b, c, d\}$,

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

- Second, we define and compute the loss L for the samples (x_i, y_i), $i = 1, \ldots, n$,

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

Example: polynomial approximation of $\cos(x)$

- Third, we solve the optimization problem

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

by a gradient descent algorithm, which has the following update rules

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

for a specified learning rate η.
The gradient descent algorithm (and any gradient type optimization algorithms) requires the gradient of the loss w.r.t coefficients $\boldsymbol{\theta}$.

Step1: initialize

```
import numpy as np
import torch
import matplotlib.pyplot as plt

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

# generate and visualize data for y=cos(x)
x = np.linspace(-0.5*np.pi, 1.5*np.pi, 2000)
y = np.cos(x)
x = torch.from_numpy(x)
y = torch.from_numpy(y)

# create tensors for the data and initiliaze the 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)
```

- generate data;
- initialized the coefficients.

Step 2: forward and backward pass

- defines a polynomial function
- computes the loss function
- implements automatic differentiation

```
# hyperparameters
num_iterations = 25000
learning_rate = 1e-7
loss_values = []

# model training
for k in range(num_iterations):
    yhat = a + b * x + c * x**2 + d * x**3
    loss = torch.sum((yhat - y)**2)
    loss_values.append(loss.item())

    if k % 1000 == 99:
        print('Iteration',k,'Loss=',loss.item())

    loss.backward()
```

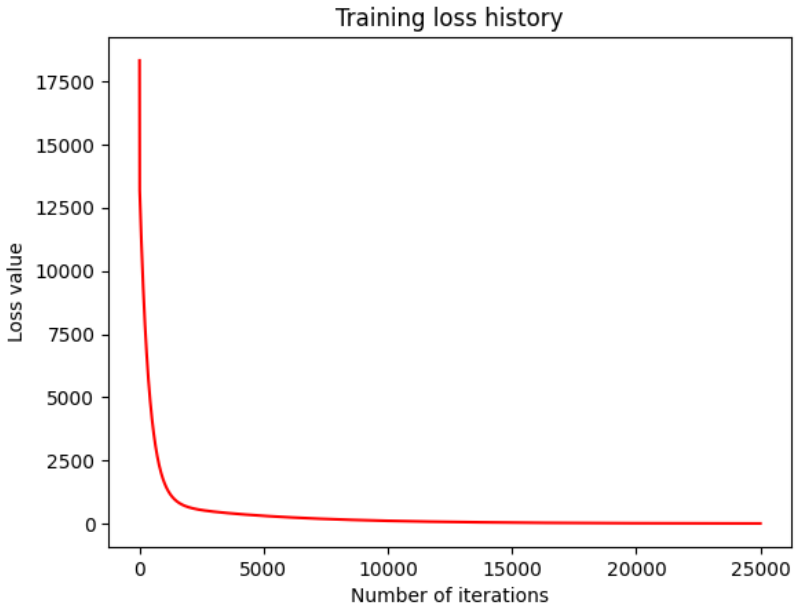
Step 3: apply gradient descent

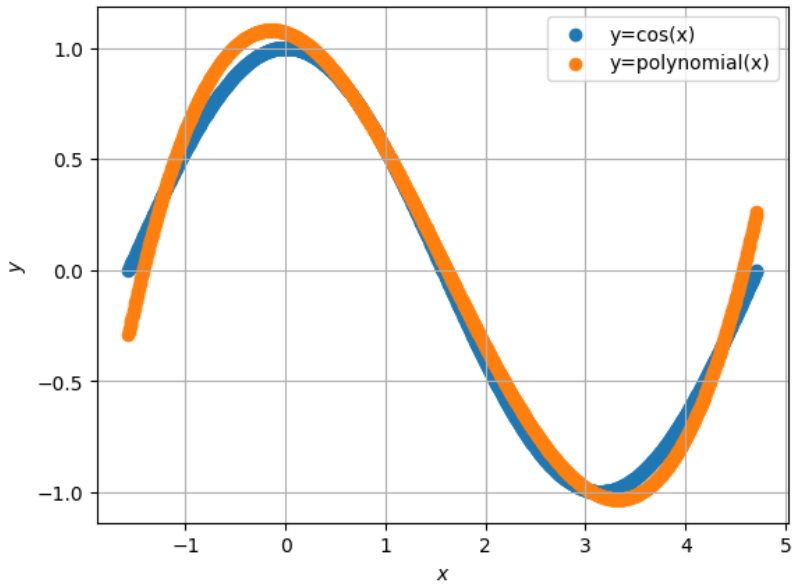
To minimize the loss function, we apply the gradient descent algorithm as follows

$$a \leftarrow a - \eta \frac{\partial L}{\partial a},$$
$$b \leftarrow b - \eta \frac{\partial L}{\partial b},$$
$$c \leftarrow c - \eta \frac{\partial L}{\partial c},$$
$$d \leftarrow d - \eta \frac{\partial L}{\partial d}.$$

```
# manually update the parameters using gradient descend
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 clear gradients after the updates
    a.grad = None
    b.grad = None
    c.grad = None
    d.grad = None
```

Results

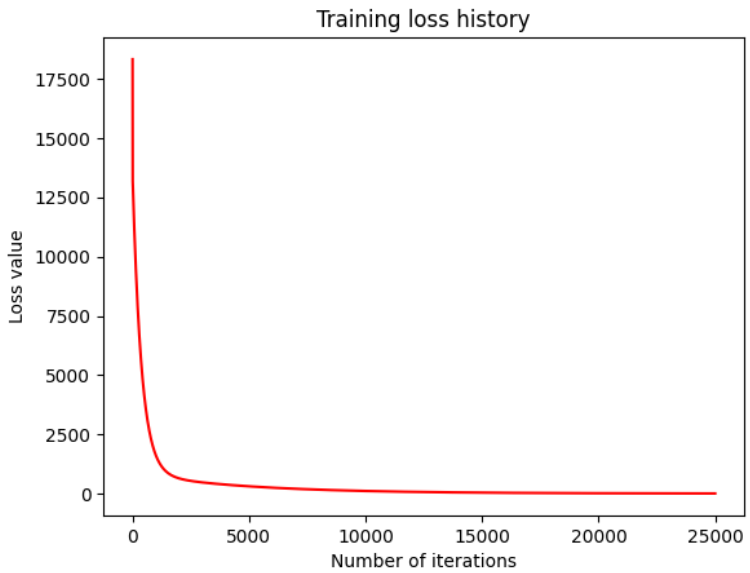




$$y = 1.068 - 0.146x - 0.484x^2 + 0.102x^3$$

Results

It is also beneficial to plot the history of training loss in the log scale so that we can visualize the smaller changes when the loss values are close to zero, e.g. the loss values after 10000 iterations. Moreover, the figure in the log scale below indicates that the loss decays exponentially after around 3000 iterations.



Artificial Neural Networks (ANN)

Goal: build a fully connected neural network to approximate data generated from $\cos(x)$.

- Define our neural network model f as

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

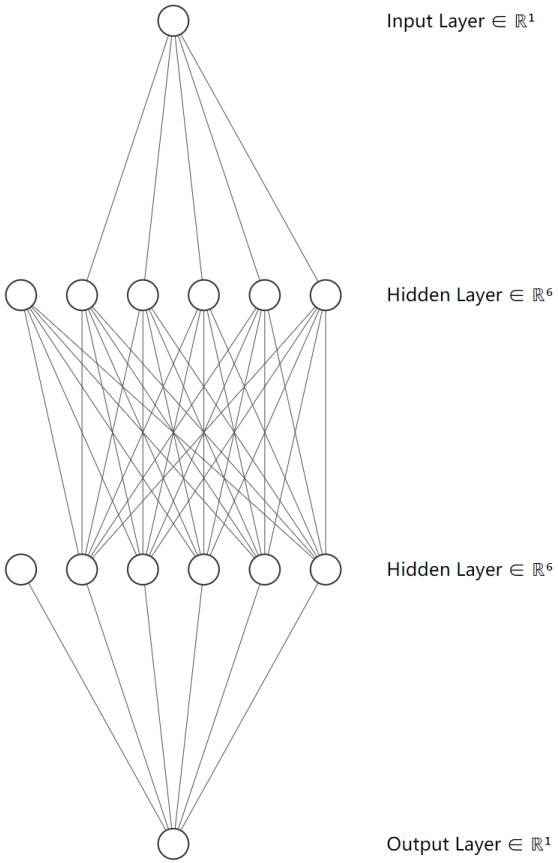
  where $\mathbf{W}, \mathbf{b}$ are weights and biases respectively.
- Compute the loss function

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

  for the given samples $(x_i, y_i), i = 1, \ldots, n$.
- 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 in Pytorch: low level APIs




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

class ANN_low(nn.Module):
    def __init__(self, in_dim=1, out_dim=1, neurons=[6,6],
                 act_fn=sigmoid):
        super(ANN_low, self).__init__()
        self.in_dim = in_dim
        self.out_dim = out_dim
        self.neurons = neurons
        self.act_fn = act_fn
        self.w1 = torch.rand(self.in_dim, self.neurons[0],
                  requires_grad=True)
        self.w1 = nn.Parameter(self.w1) # register as nn.Parameter
        self.b1 = nn.Parameter(torch.zeros(self.neurons[0],
                  requires_grad=True))
        self.w2 = nn.Parameter(torch.rand(self.neurons[0],
                  self.neurons[1], requires_grad=True))
        self.b2 = nn.Parameter(torch.zeros(self.neurons[1],
                  requires_grad=True))
        self.w3 = nn.Parameter(torch.rand(self.neurons[1],
                  self.out_dim, requires_grad=True))
        self.b3 = nn.Parameter(torch.zeros(1, requires_grad=True))
    def forward(self, x):
        z = torch.matmul(x, self.w1) + self.b1
        z = self.act_fn(z)
        z = torch.matmul(z, self.w2) + self.b2
        z = self.act_fn(z)
        z = torch.matmul(z, self.w3) + self.b3
        return z
```

Training using low level APIs

Suppose you have the samples and set up all hyperparameters, then we can build the model and train it as follows.

```
########## initialize the ANN model
in_dim = 1
out_dim = 1
neurons = [6, 6]
model = ANN_low(in_dim, out_dim, neurons, sigmoid)

########## train the model
for k in range(num_iterations):
    current_index = np.random.choice(N_train, batch_size,
          replace = False)
    batchX = x_train[current_index]
    batchY = y_train[current_index].view(batch_size,1)
    batchY_pred = model.forward(batchX)   # forward pass
    loss_value = loss_fn(batchY_pred, batchY)   # compute the loss value
    loss_value.backward()   # automatic differentiate to get the gradients
    with torch.no_grad():   # manually update parameters and clear gradients
        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
        model.w1.grad = None
        model.b1.grad = None
        model.w2.grad = None
        model.b2.grad = None
        model.w3.grad = None
        model.b3.grad = None
```

Optimizers in Pytorch

Instead of updating parameters manually, we can use the module "torch.optim", which provides multiple optimizers (SGD, Adam, etc).
When using low level APIs, we need to use `nn.parameters()` to register a Tensor as trainable model parameters, before setting an optimizer and using it.

```
class ANN_low(nn.Module):
    def __init__(self,in_dim=1,out_dim=1,neurons=[6,6],
                 act_fn=sigmoid):
        super(NeuralNetworkLow, self).__init__()
        ......
        self.w1 = torch.rand(self.in_dim, self.neurons[0],
                    requires_grad=True)
        self.w1 = nn.Parameter(self.w1)    # register it as nn.Parameter
        self.b1 = nn.Parameter(torch.zeros(self.neurons[0],
                    requires_grad=True))
        ......
model = ANN_low(in_dim, out_dim, neurons, sigmoid)
optimizer = torch.optim.SGD(model.parameters(),lr=0.01)
......
```

Implementing optimziers

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

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

```
model= ANN_low()
loss_fn = torch.nn.MSELoss()
for k in range(num_iterations):
    ......
    optimizer.zero_grad()
    batchY_pred = model.forward(batchX)
    loss_value = loss_fn(batchY_pred, batchY)
    loss_value.backward()
    # use optimizer to 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)
```

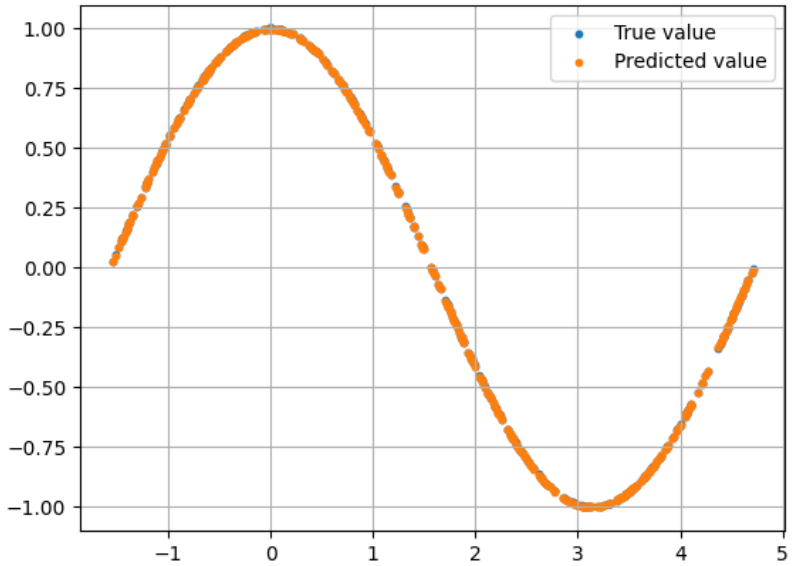
Higher level functions in Pytorch

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

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

```
import torch
import torch.nn as nn

class ANN_high(nn.Module):
    def __init__(self,in_dim=1,out_dim=1,neurons=[6,6],
                 act_fn=nn.Sigmoid):
        super(ANN_high, self).__init__()
        self.in_dim = in_dim
        self.out_dim = out_dim
        self.neurons = neurons
        self.act_fn = act_fn()
        self.layer1 = nn.Linear(self.in_dim,
                                self.neurons[0])
        self.layer2 = nn.Linear(self.neurons[0],
                                self.neurons[1])
        self.layer3 = nn.Linear(self.neurons[1],
                                self.out_dim)
        # put all the layers together in order
        self.forwardpass = nn.Sequential(self.layer1,
                 self.act_fn, self.layer2, self.act_fn,
                 self.layer3)
    def forward(self, x):
        y = self.forwardpass(x)
        return y
```

Results of training ANNs



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

View model parameters and their gradients

- For the low-level case,

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

- For the high-level case,

```
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 (for model inference):

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

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

```
PATH = 'model.pt'
# collect relevant information
EPOCH = 5
LOSS = 0.01
torch.save({
            'epoch': EPOCH,
            'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'loss': LOSS,
            }, PATH)
model.load_state_dict(torch.load(PATH))
model.eval()
```

Restart from a checkpoint

Load the saved model and resume the training:

```
# initialize a model and optimizer again
model_reloaded = ANN_high(in_dim, out_dim, neurons, torch.nn.Sigmoid)
optimizer = torch.optim.Adam(model_reloaded.parameters(), lr=learning_rate)

checkpoint = torch.load(PATH)
model_reloaded.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']
loss = checkpoint['loss']

# model_reloaded.eval()  # use it for inference mode
model_reloaded.train()  # activate the training mode

# the remaining step is to use the train loop for model_reloaded
```

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_value = loss_fn(batchY_pred, batchY)
lambda_coeff = 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_value = loss_value + lambda_coeff  * l2_norm

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

Hyperparameter tuning

The K-fold cross-validation for models selection,

- Define the search space of hyperparameters, e.g., the number of nodes and layers.
- Split the training data set into K different subsets.
- Select one subset as a validation set and train the model on the remaining K-1 subsets.
- Calculate the model's metric on the validation set.
- Repeat the above steps by exploring all possible validation sets.
- Calculate the final metric by averaging over K cases.
- Repeat the above steps by exploring the space of hyperparameters.
- Rank the model candidates using their final metric.

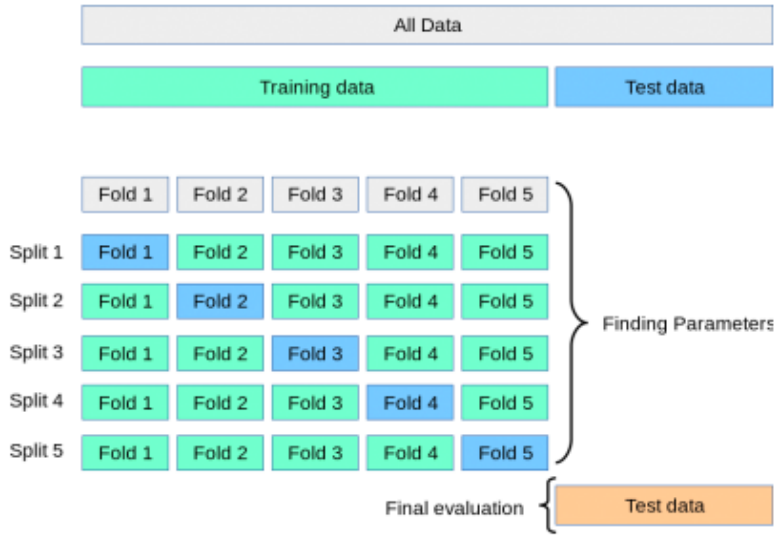
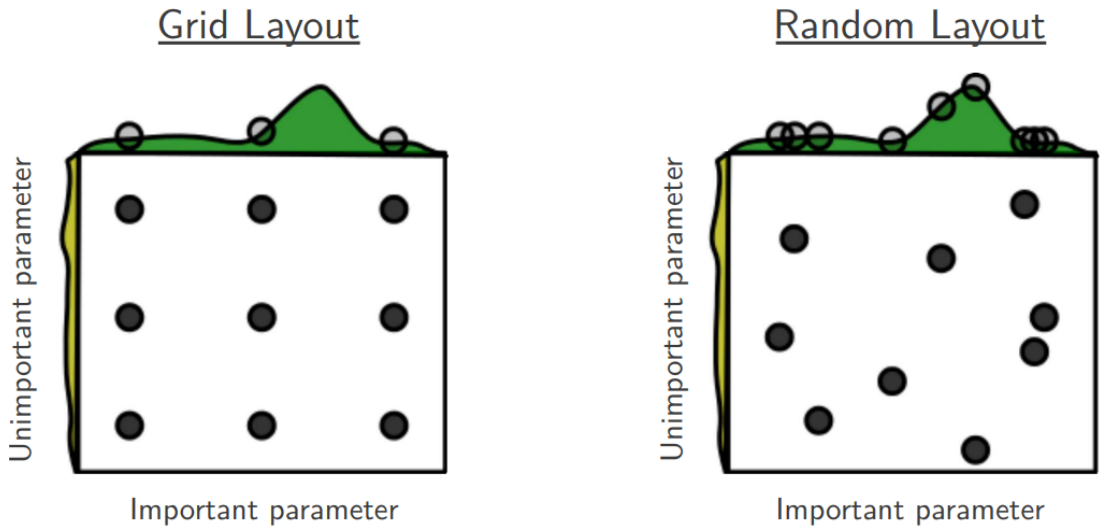


Figure: 5-fold cross validation

Hyperparameter tuning

Methods of exploring the search space of hyperparameters,

- Grid search or random search,

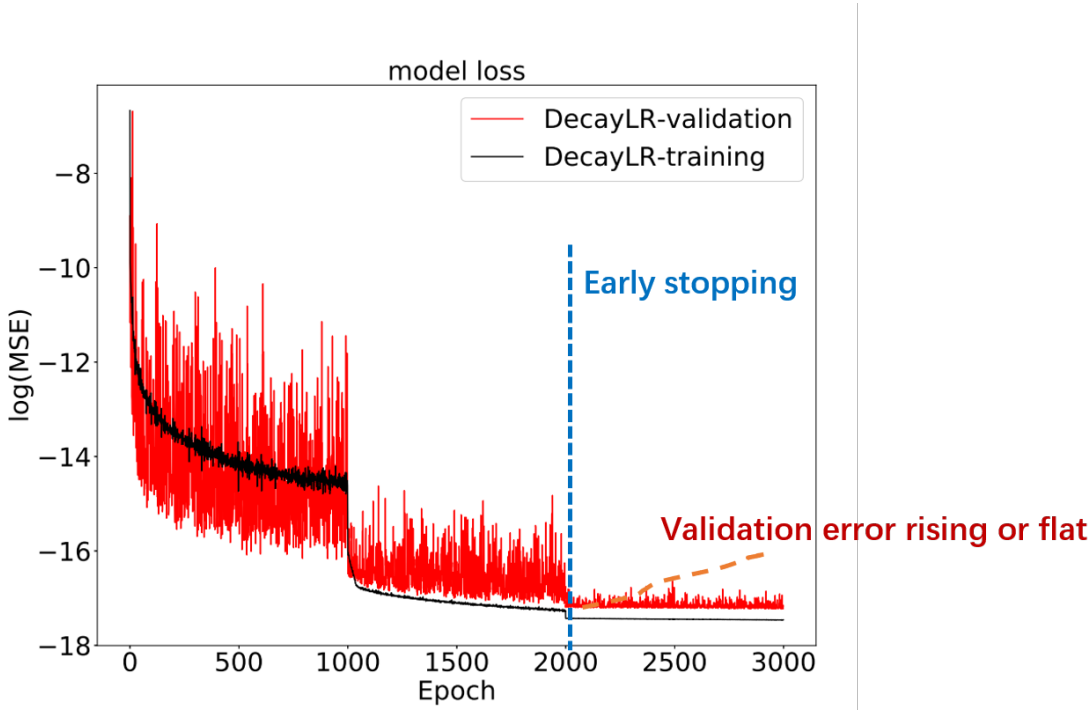


James & Yoshua: Random Search for Hyper-Parameter Optimization (2012)

- Hyperparameter Optimization, for instance, Bayesian optimization.

Early stopping

There are some training techniques to reduce 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,

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