Lec3 神经网络与深度学习

·AIPKU·

人工智能引论实践课 计算机视觉小班

主讲人：刘家瑛

部分讲义来源与致谢

1. PyTorch Documentation. CONV2D.

   https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html

2. Convolution arithmetic.

   https://github.com/vdumoulin/conv_arithmetic

3. Grace Zhang. What is the kernel trick? Why is it important?

4. PyTorch Tutorial. Training a Classifier.

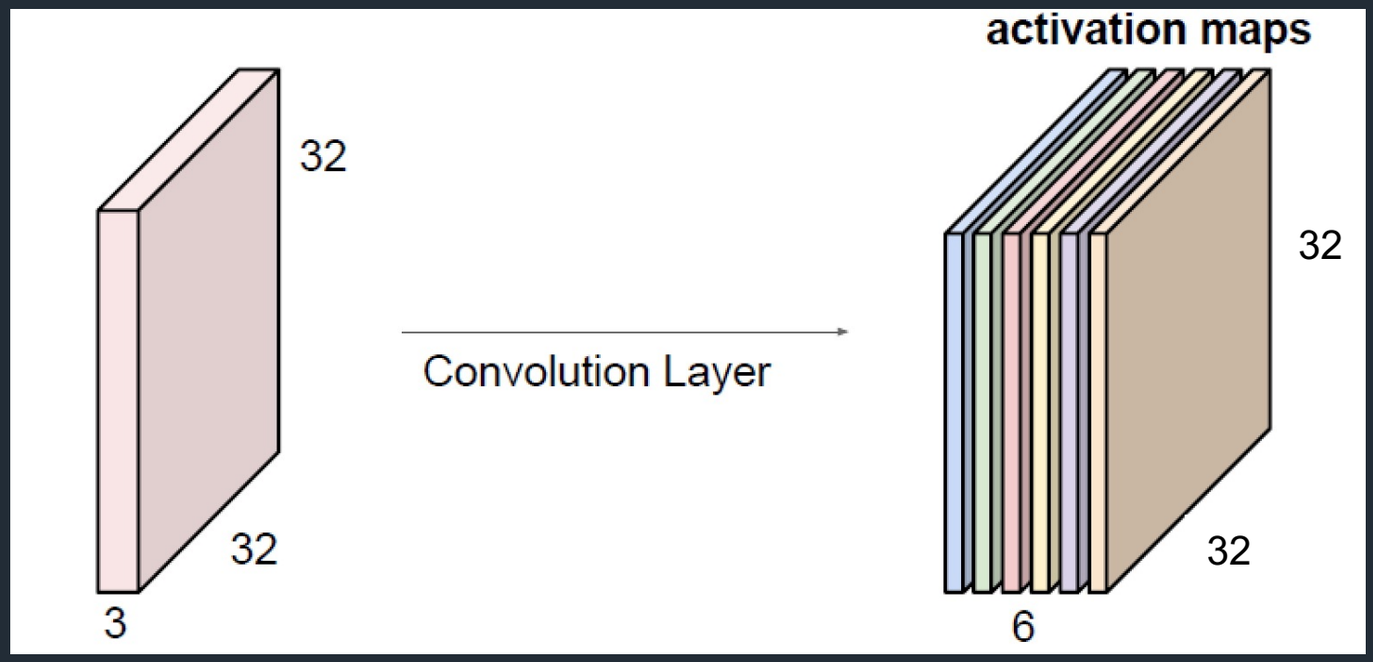
   https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html

5. Wikipedia. CUDA.

   https://en.wikipedia.org/wiki/CUDA

基础网络结构：卷积层

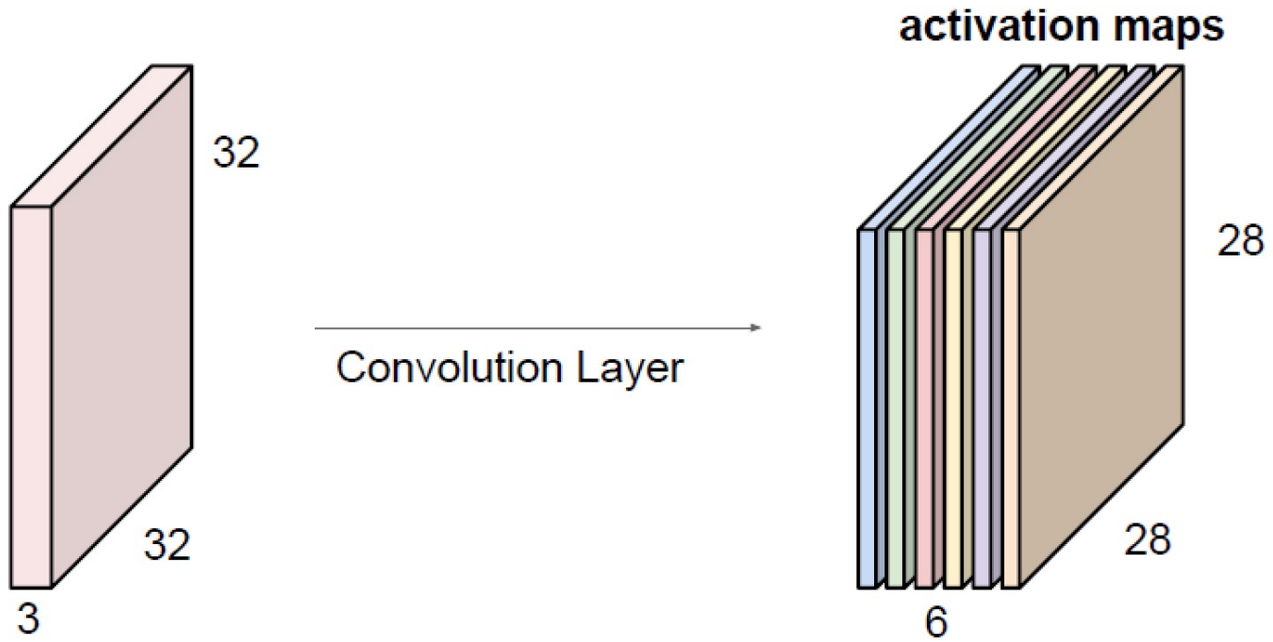
- 单层卷积 `self.conv1 = nn.Conv2d(3, 6, 3, 1, 1)`




```
import torch
torch.nn.Conv2d(in_channels, out_channels, kernel_size,
                stride=1, padding=0, dilation=1, groups=1,
                bias=True, padding_mode='zeros')
```

基础网络结构：卷积层

- 单层卷积：在通道上可以看作全连接层，在空间上看作卷积

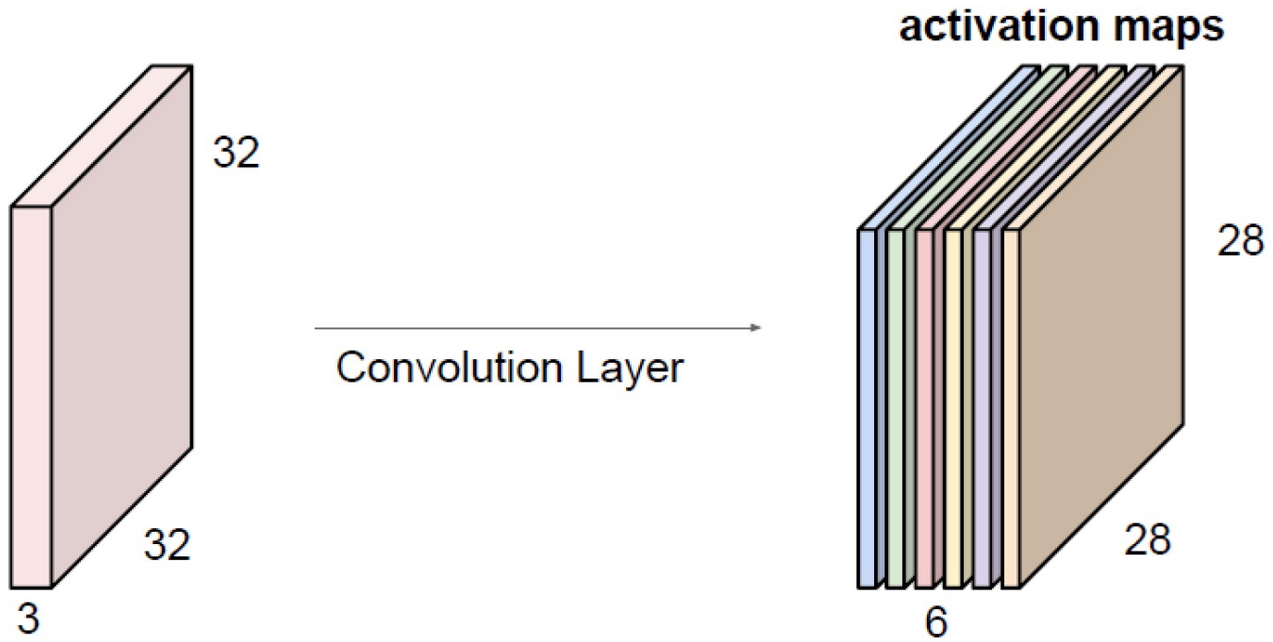



```
self.conv1 = nn.Conv2d(3, 6, 3, 1, 1)
```

- 输入特征图形状：(8, 3, 32, 32) [b, c, h, w]
- 输出特征图形状：(8, 6, 32, 32)
- 卷积核大小: 3x3，stride=1，padding=1, bias=True
- 该卷积层有多少参数?

基础网络结构：卷积层

- 单层卷积：在通道上可以看作全连接层，在空间上看作卷积

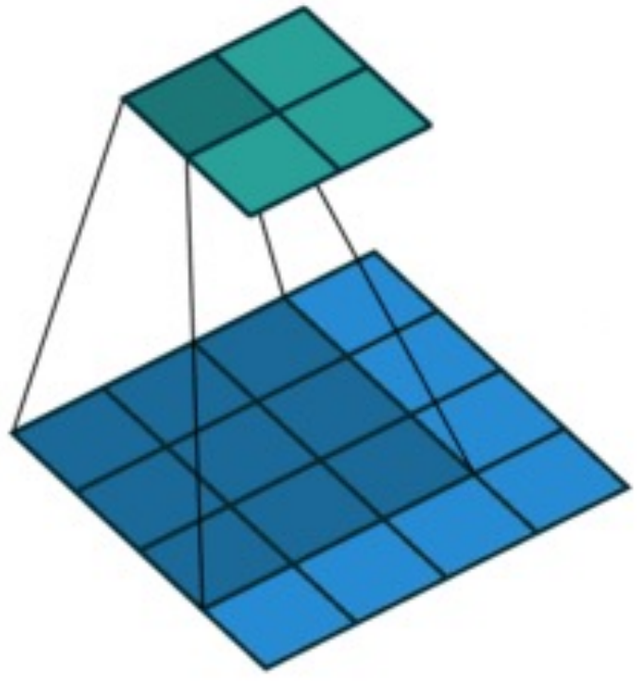



```
self.conv1 = nn.Conv2d(3, 6, 3, 1, 1)
```

- 输入特征图形状：(8, 3, 32, 32) [b, c, h, w]
- 输出特征图形状：(8, 6, 32, 32)
- 卷积核大小: 3x3，stride=1，padding=1, bias=True
- 该卷积层有多少参数？3x6x3x3 + 6

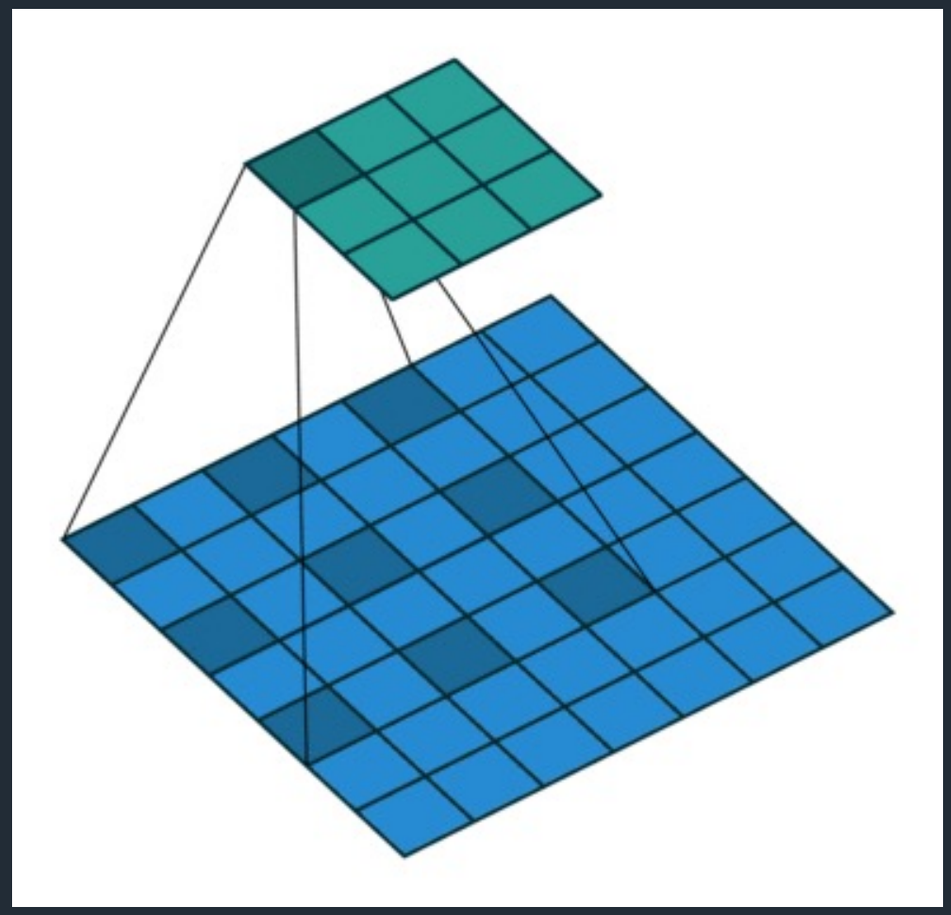
基础网络结构：卷积层

- 感受野：单个神经元可以“看见”的区域大小
- 影响感受野的因素：
  - 卷积核大小（大卷积核会导致参数过多）
  - 网络深度：越深的神经元感受野越大
  - 重采样，插空卷积...
- 我们一般希望感受野能尽量大



空洞卷积

- 空洞卷积 Dilated Convolution
  - 在卷积核中插空
  - 直接扩大感受野



```
torch.nn.Conv2d(in_channels, out_channels, kernel_size,
                stride=1, padding=0, dilation=2, groups=1,
                bias=True, padding_mode='zeros')
```

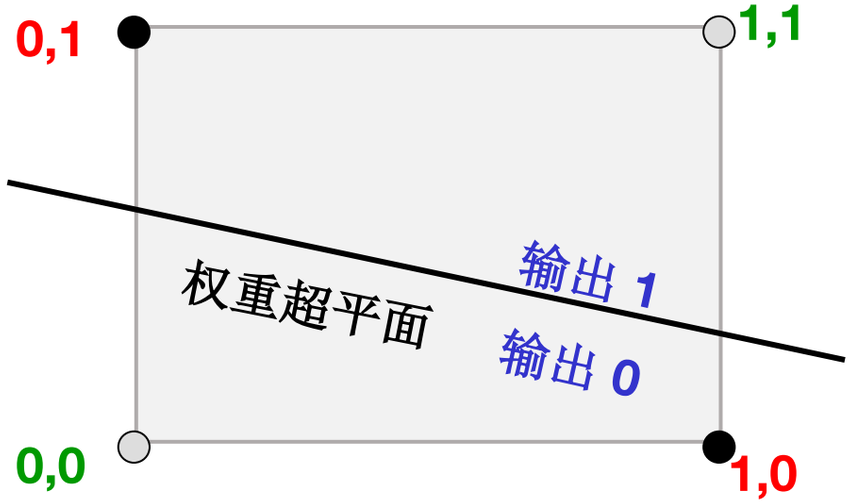
神经网络发展中遇到的挑战

- 异或难题

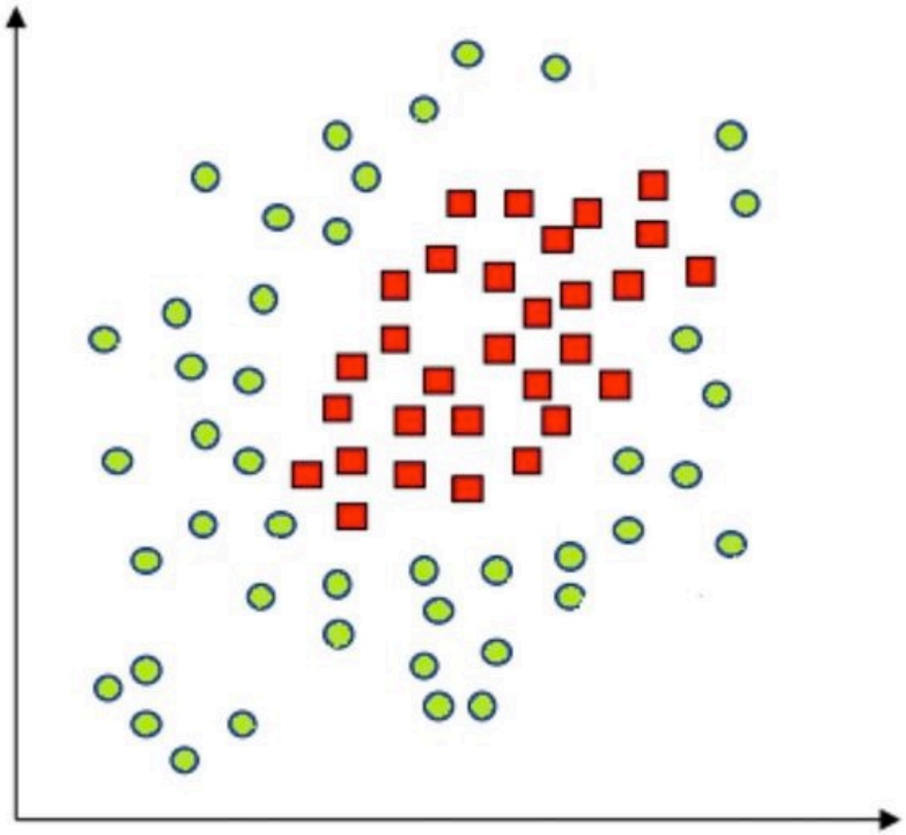
  异或运算真值表:　(1,1) → 1;　(0,0) → 1

  (1,0) → 0;　(0,1) → 0

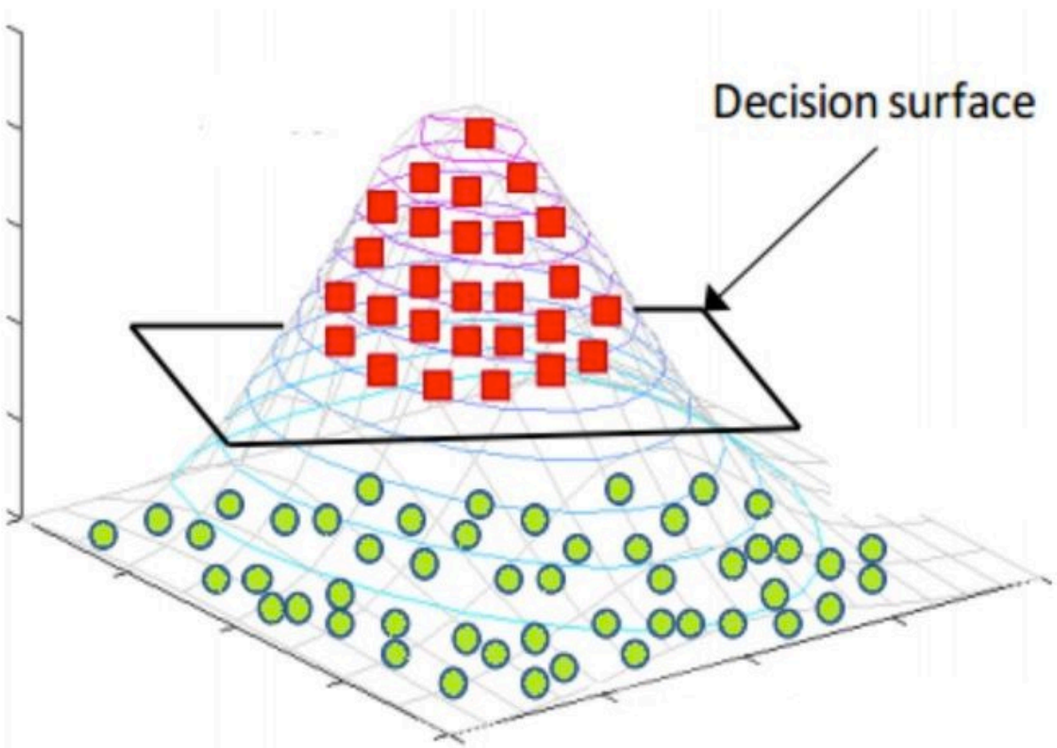
  无法用卷积模块拟合



支持向量机 —— 核函数



kernel



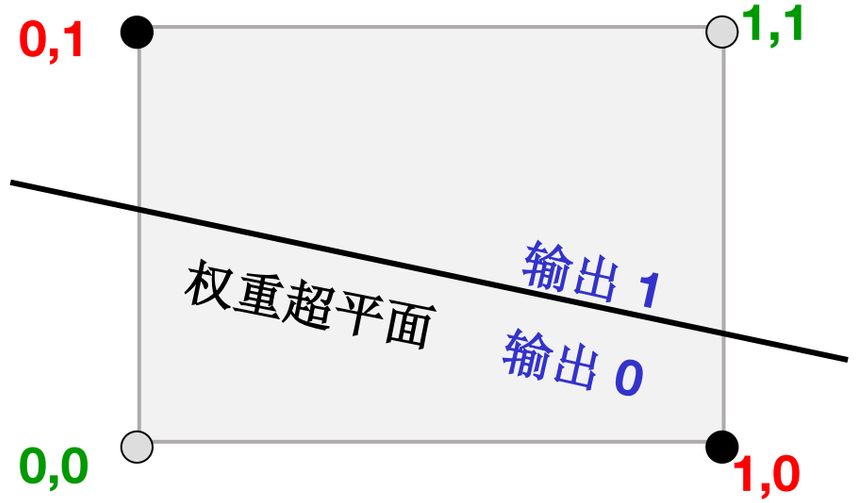
神经网络发展中遇到的挑战

- 异或难题

  异或运算真值表:　(1,1) → 1;　(0,0) → 1

  (1,0) → 0;　(0,1) → 0

  无法用卷积模块拟合

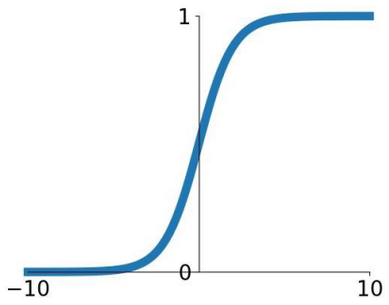


- 神经网络的解决方法：引入非线性激活函数攀隐藏层

激活函数

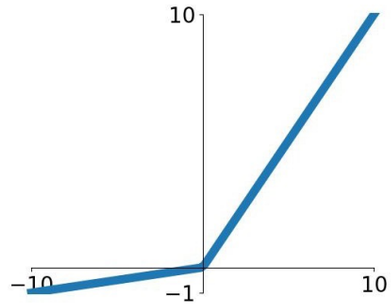
Sigmoid

$$\sigma(x) = \frac{1}{1+e^{-x}}$$



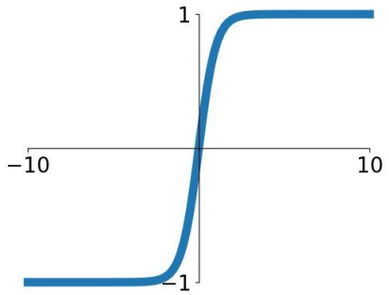
Leaky ReLU

$$\max(0.1x, x)$$



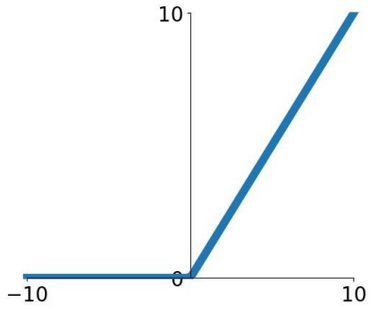
tanh

$$\tanh(x)$$



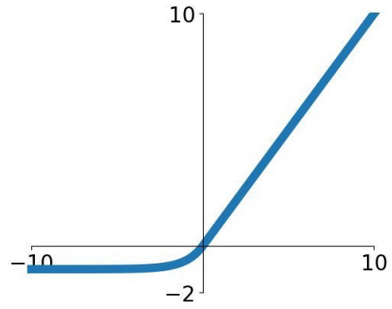
ReLU

$$\max(0, x)$$



ELU

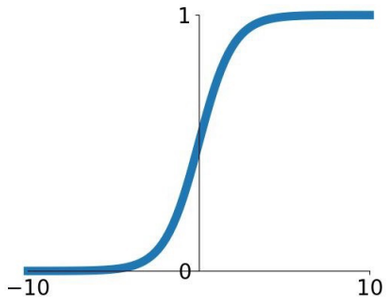
$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



激活函数

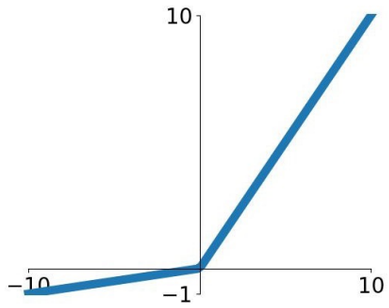
Sigmoid

`torch.nn.Sigmoid`



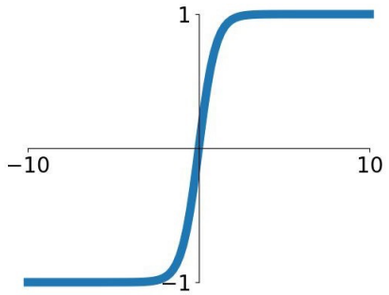
Leaky ReLU

`torch.nn.LeakyReLU`



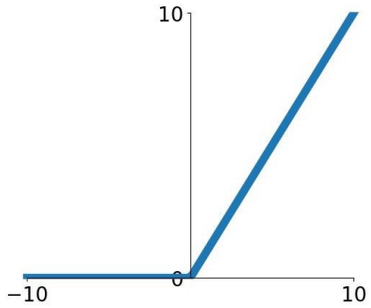
tanh

`torch.nn.Tanh`



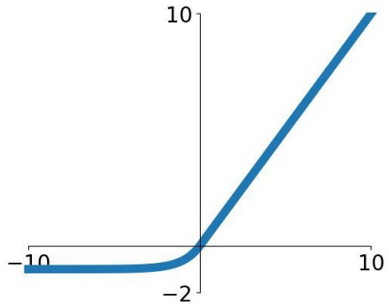
ReLU

`torch.nn.ReLU`



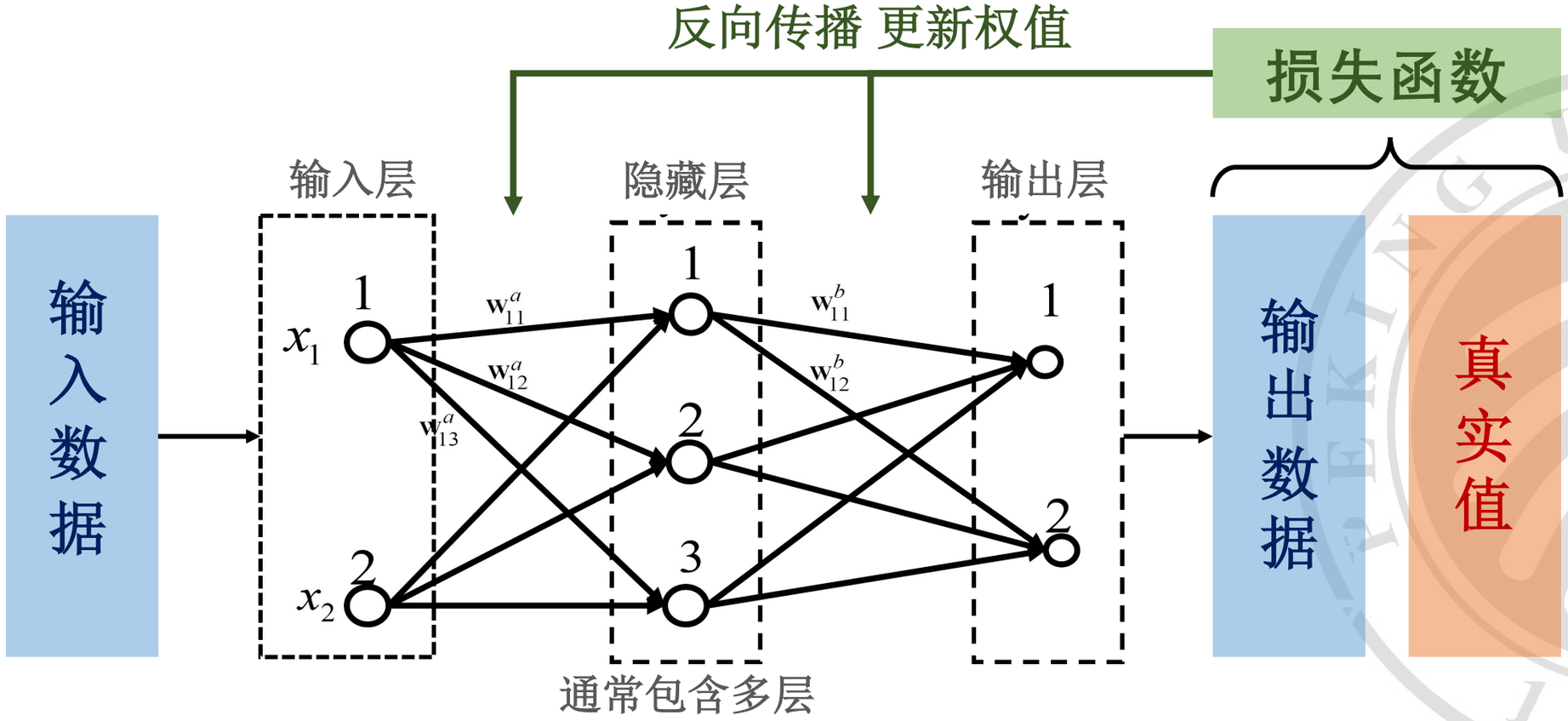
ELU

`torch.nn.ELU`



现代人工神经网络

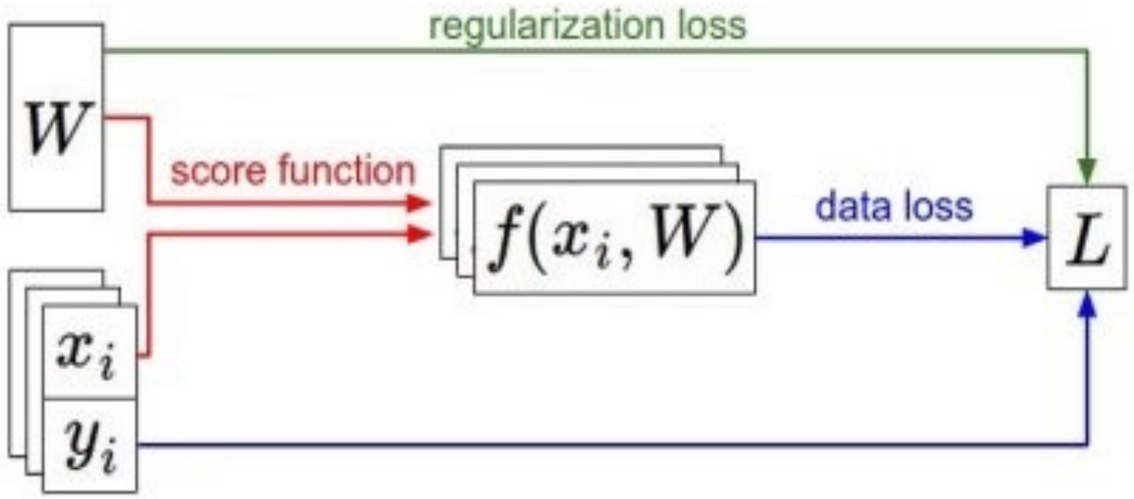
- 多层神经元组成神经元网络
- 能够拟合多种函数
- 损失函数表达神经网络训练的目标
- 随机梯度下降法求解关于参数的最优化问题
- 反向传播算法计算多层参数的梯度

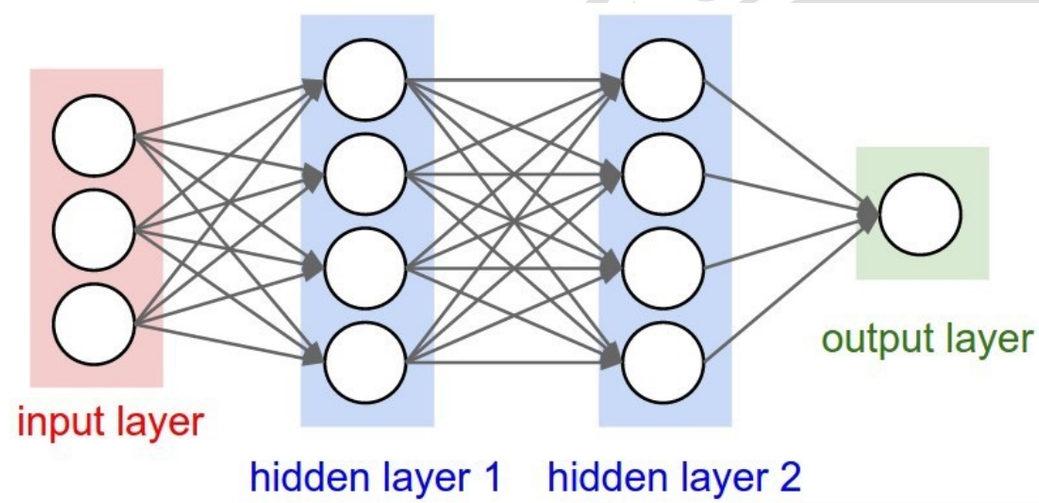


人工神经网络

多层神经元网络的训练过程

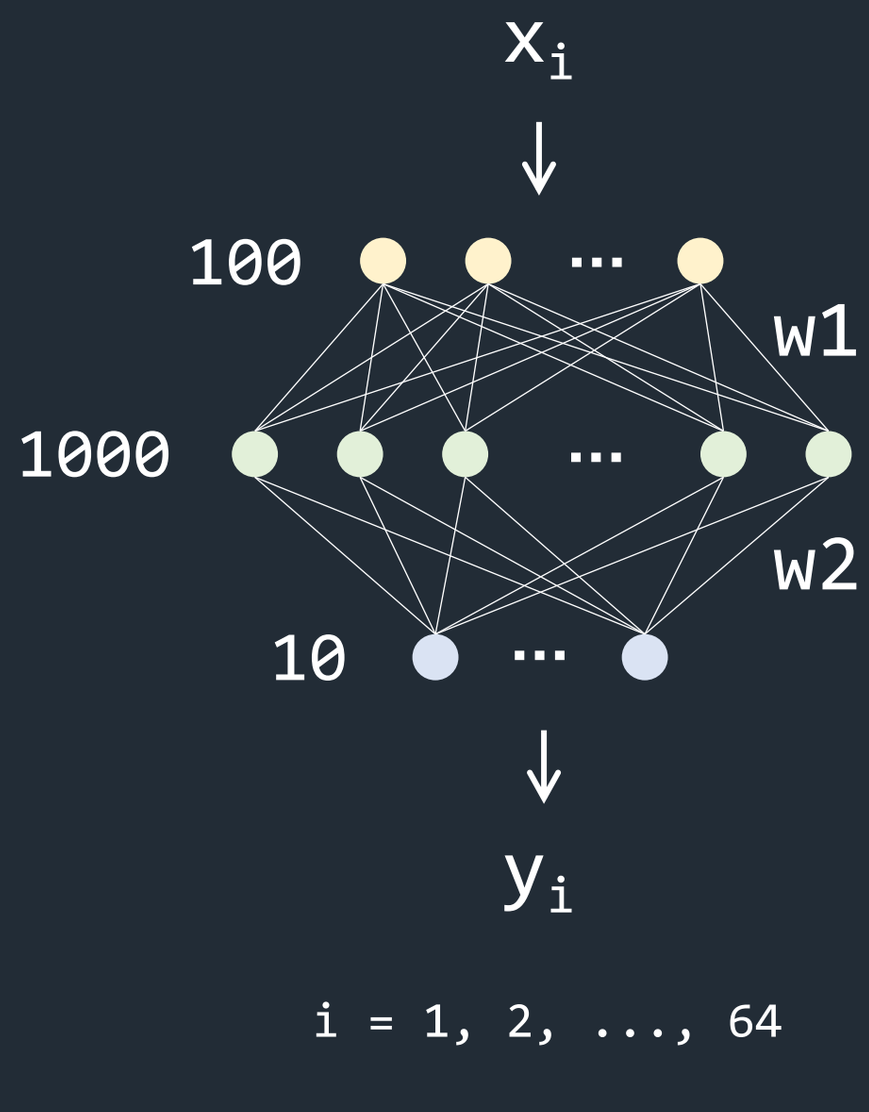
- 准备数据（处理成 batch)
- 计算前向传播结果 (Feed-Forward)
- 计算损失函数 (Loss Function)
- 反向传播 (Back-Propagation)
- 随机梯度下降 (Stochastic Gradient Descent)





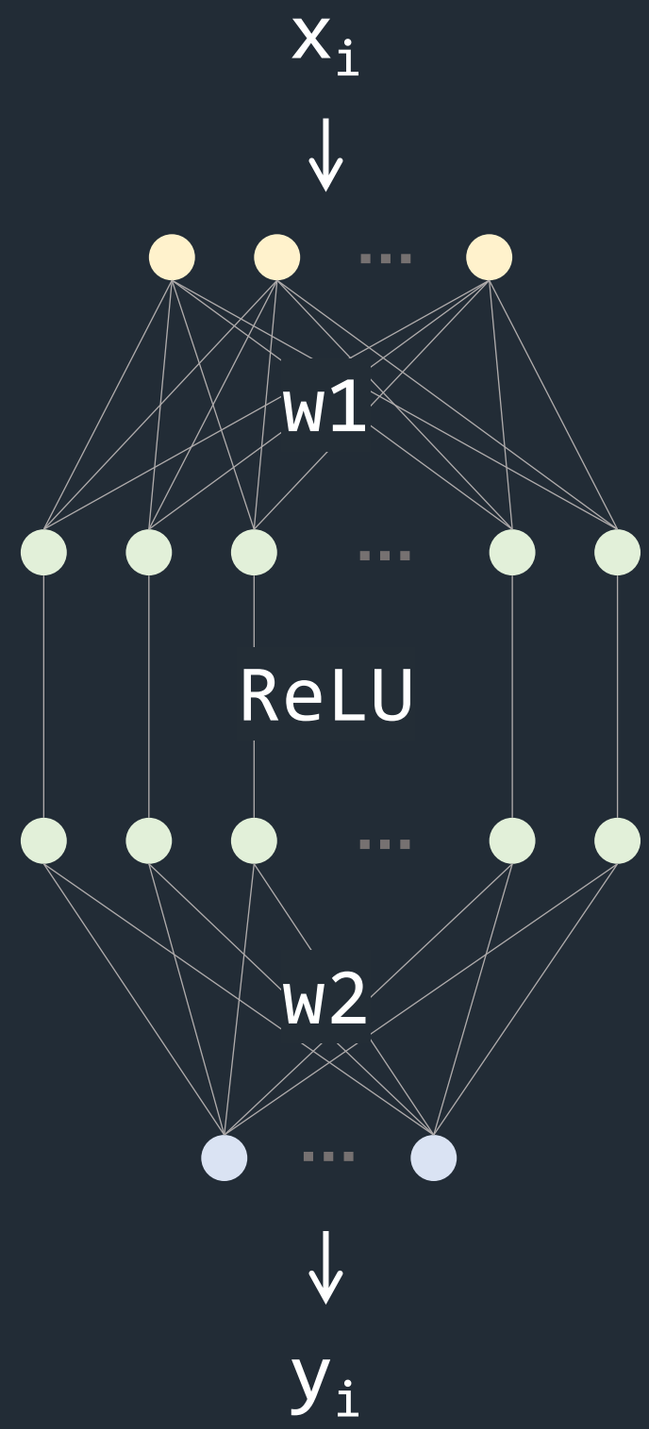
神经网络原型

```python
import numpy as np
# N=batchsize, D_in=输入维度
# H=隐藏层神经元数量（隐藏层输出维度）, D_out=网络输出维度
N, D_in, H, D_out = 64, 100, 1000, 10

# 样例：随机矩阵
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# 随机初始化网络内部权重
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
```



神经网络原型

```python
for t in range(500):
    # 前向传播 y = ReLU(x·w1)·w2
    h = x.dot(w1)
    h_relu = np.maximum(h, 0)
    y_pred = h_relu.dot(w2)

    # 计算损失函数 MSE
    # 为后续求导方便， mean 改为 sum
    loss = np.square(y_pred - y).sum()
    print(t, loss)
```



神经网络原型

$$\frac{\partial\ loss}{\partial\ y\_pred_{i,j}} = \frac{\partial\ \sum_{a,b}(y\_pred_{a,b} - y_{a,b})^2}{\partial\ y\_pred_{i,j}} = \frac{\partial\ (y\_pred_{i,j} - y_{i,j})^2}{\partial\ y\_pred_{i,j}} = 2(y\_pred_{i,j} - y_{i,j})$$

```
# 反向传播：求关于各个变量的梯度并且存到 grad_ 开头的变量里
grad_y_pred = 2.0 * (y_pred - y)
```

$$\frac{\partial\ loss}{\partial\ w2} = (h\_relu)^T \cdot \frac{\partial\ loss}{\partial\ y\_pred}$$

```
# y_pred = h_relu.dot(w2)
grad_w2 = h_relu.T.dot(grad_y_pred)
```

神经网络原型

$$\frac{\partial\ loss}{\partial\ w2} = (h\_relu)^T \cdot \frac{\partial\ loss}{\partial\ y\_pred}$$

推导： $y\_pred_{i,j} = \sum_k h\_relu_{i,k} \cdot w2_{k,j}$

$$\frac{\partial\ loss}{\partial\ w2_{i,j}} = \sum_{a,b} \frac{\partial\ loss}{\partial\ y\_pred_{a,b}} \frac{\partial\ y\_pred_{a,b}}{\partial\ w2_{i,j}}$$

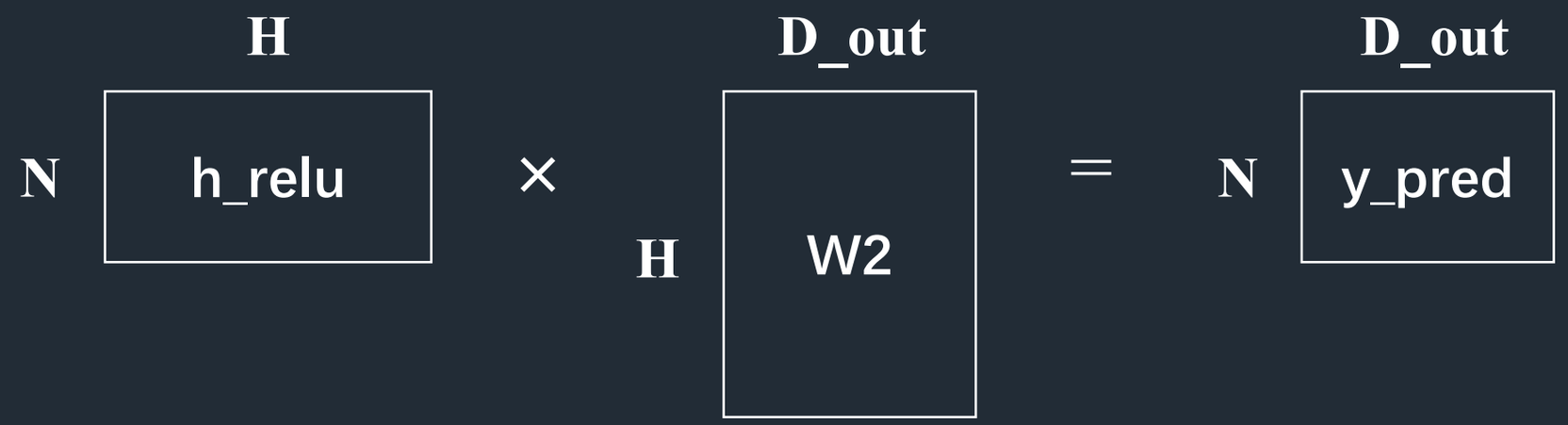
$$= \sum_{a,b} \frac{\partial\ loss}{\partial\ y\_pred_{a,b}} \frac{\partial\ \sum_c h\_relu_{a,c} \cdot w2_{c,b}}{\partial\ w2_{i,j}}$$

$c = i,\ b = j$

$$= \sum_a \frac{\partial\ loss}{\partial\ y\_pred_{a,j}} \frac{\partial\ h\_relu_{a,i} \cdot w2_{i,j}}{\partial\ w2_{i,j}}$$

$$= \sum_a grad\_y\_pred_{a,j} \cdot h\_relu_{a,i}$$

神经网络原型



$$\boxed{\frac{\partial\, loss}{\partial\, w2}} = (h\_relu)^T \cdot \boxed{\frac{\partial\, loss}{\partial\, y\_pred}}$$

H × D_out　　N × D_out

↓

H × N

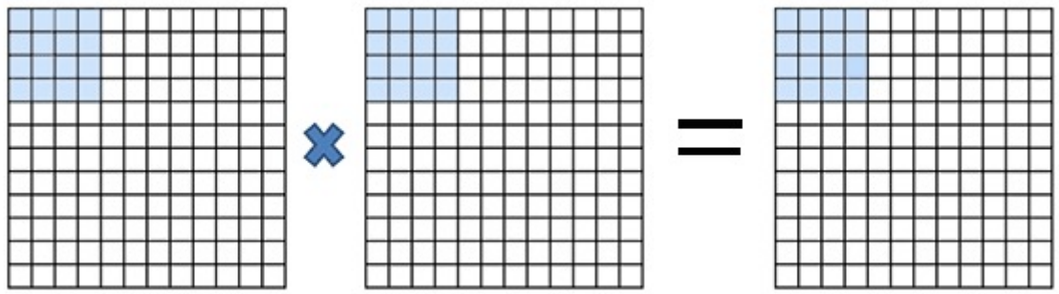
神经网络原型

```
# 对称地，关于 h_relu 的梯度
grad_h_relu = grad_y_pred.dot(w2.T)

# ReLU后为0的部分，其梯度也是0
grad_h = grad_h_relu.copy()
grad_h[h < 0] = 0

grad_w1 = x.T.dot(grad_h)

# 更新权值
w1 -= learning_rate * grad_w1
w2 -= learning_rate * grad_w2
```

神经网络原型

观察发现，神经网络程序有以下特性

- 运算量大，但运算多为矩阵乘法，具有潜在并行性



- 求导表示繁琐，但具有模块化性质
  - 在每一层操作确定的情况下（例如矩阵乘法），梯度计算公式也完全确定
  - 梯度计算公式只与前一层传播过来的梯度矩阵以及本层的参数、输出有关
- 可设计GPU加速和自动求导

深度学习框架

Caffe
(UC Berkeley) → **Caffe2**
(Facebook)

Torch
(NYU / Facebook) → **PyTorch**
(Facebook)

Theano
(U Montreal) → **TensorFlow**
(Google)

MXNet
(Amazon)
Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK
(Microsoft)

Deeplearning4j

And others...

深度学习框架

Caffe
(UC Berkeley)

Caffe2
(Facebook)

Torch
(NYU / Facebook)

PyTorch
(Facebook)

Theano
(U Montreal)

TensorFlow
(Google)

MXNet
(Amazon)

Developed by U Washington, CMU, MIT, Hong Kong U, etc but main framework of choice at AWS

CNTK
(Microsoft)

Deeplearning4j

And others...

PyTorch

TensorFlow

NVIDIA CUDA

PyTorch 神经网络原型

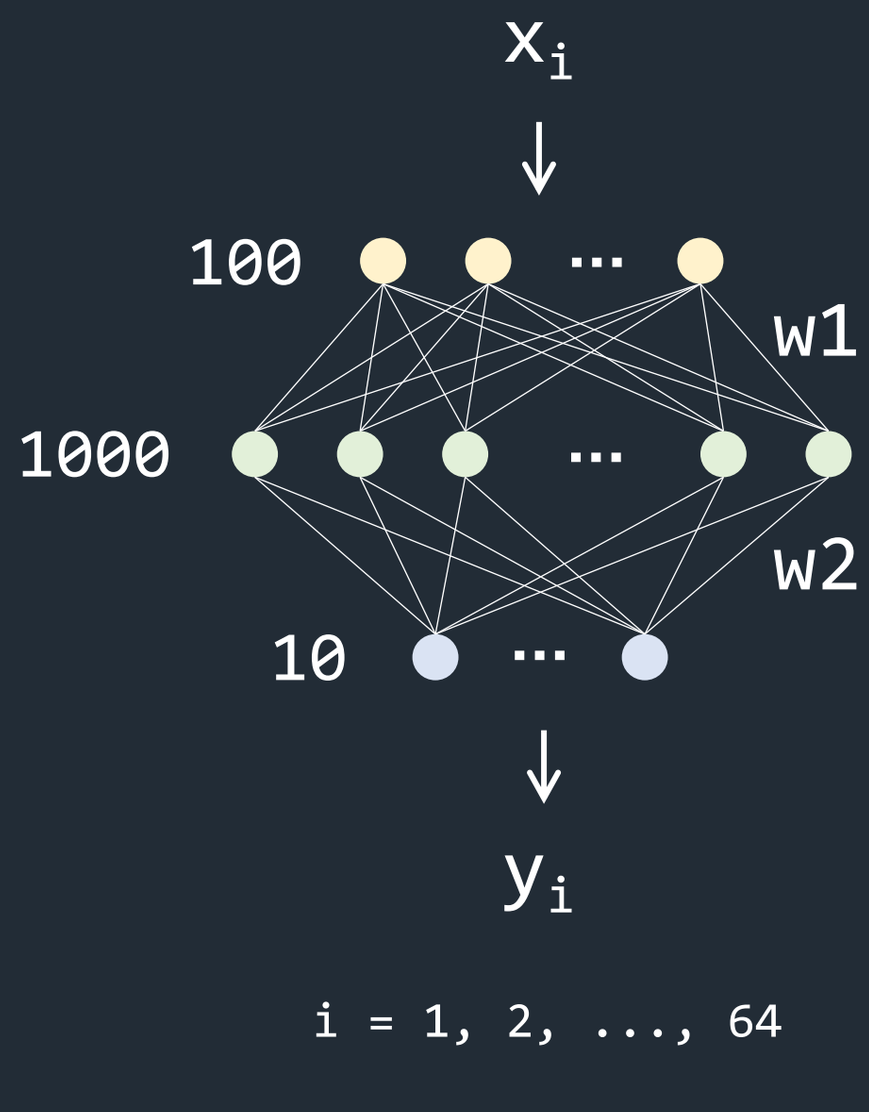
```
import torch
dtype = torch.float                    # 32bit float
device = torch.device("cpu")
# device = torch.device("cuda:0") # 改为这一行来使用 GPU

N, D_in, H, D_out = 64, 1000, 100, 10

# 与numpy接口类似，可查询文档
x = torch.randn(N, D_in, device=device, dtype=dtype)
y = torch.randn(N, D_out, device=device, dtype=dtype)
```

PyTorch 神经网络原型

```
# 指定requires_grad, pytorch会自动建立变量来存储梯度
# 类似于我们之前自己定义的grad_变量
w1 = torch.randn(D_in, H, device=device, dtype=dtype,
                 requires_grad=True)
w2 = torch.randn(H, D_out, device=device, dtype=dtype,
                 requires_grad=True)

learning_rate = 1e-6
```

对比numpy形式

```python
import numpy as np
# N=batchsize, D_in=输入维度
# H=隐藏层神经元数量（隐藏层输出维度）, D_out=网络输出维度
N, D_in, H, D_out = 64, 100, 1000, 10

# 样例：随机矩阵
x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

# 随机初始化网络内部权重
w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6
```



PyTorch 神经网络原型

```
for t in range(500):
    # 前向传播，mm: matrix multiplication
    # 因为有自动求导，不需要存中间变量
    y_pred = x.mm(w1).clamp(min=0).mm(w2)

    # 计算损失函数
    # loss.item() 获取loss中的标量数值（而不是矩阵）
    loss = (y_pred - y).pow(2).mean()
    if t % 100 == 99:
        print(t, loss.item())
```

PyTorch 神经网络原型

```
# 自动求导会把梯度存到每个矩阵的grad属性（成员变量）里
loss.backward()
# 使用 no_grad 环境，因为我们不想让这部分运算也被求导
with torch.no_grad():
    w1 -= learning_rate * w1.grad
    w2 -= learning_rate * w2.grad

# 清空各个矩阵的grad成员变量，以备下次求导
w1.grad.zero_()
w2.grad.zero_()
```

进一步封装

```
N, D_in, H, D_out = 64, 1000, 100, 10
x = torch.randn(N, D_in)
y = torch.randn(N, D_out)

# 矩阵乘法 → Linear 层
# 多层 → Sequential model
model = torch.nn.Sequential(
    torch.nn.Linear(D_in, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, D_out),
)
```

进一步封装

```
loss_fn = torch.nn.MSELoss(reduction='mean')

learning_rate = 1e-4

# 使用 SGD 优化器
optimizer = torch.optim.SGD(model.parameters(),
                            lr=learning_rate)
```

进一步封装

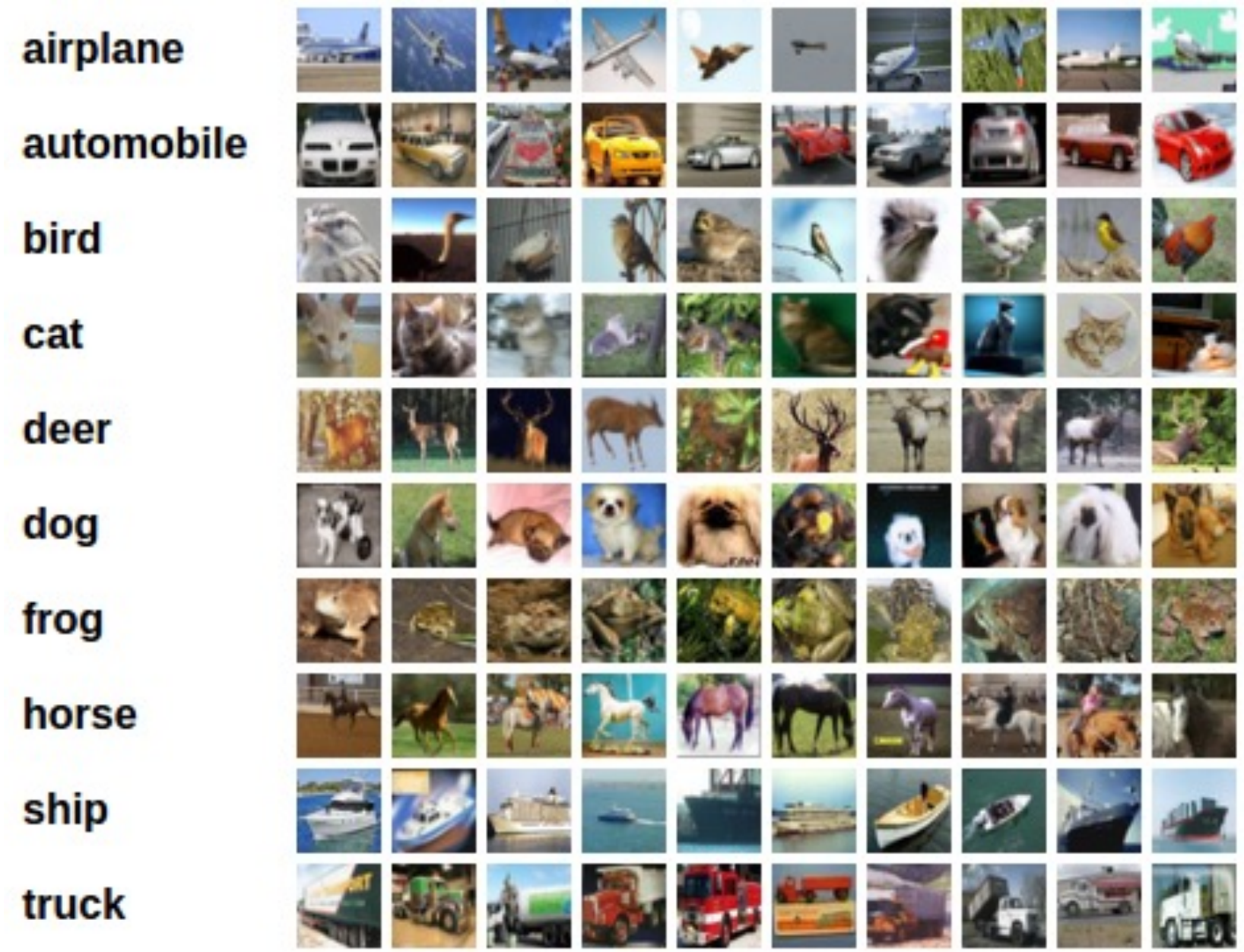
```
for t in range(500):
    y_pred = model(x)                # 前向传播

    loss = loss_fn(y_pred, y)  # 计算损失函数
    if t % 100 == 99:
        print(t, loss.item())

    optimizer.zero_grad()        # 清空之前各个变量的 grad
    loss.backward()              # 反向传播
    optimizer.step()             # 更新参数
```

PyTorch 神经网络程序设计

- 准备数据
- 定义模型和优化器
- 循环迭代
  - 前向传播
  - 计算损失函数
  - 清空梯度
  - 反向传播
  - 更新梯度
  - 可选：打印损失函数值
  - 存储模型

例子：CIFAR10

- CIFAR 10，简单图像分类任务，10类，分辨率 32x32



机器学习第一步：准备数据

- PyTorch Dataset 类
- 开发者可继承 Dataset 类实现自己的数据加载器

准备数据

```python
import torch
import torchvision
import torchvision.transforms as transforms
import numpy as np

class CIFAR10Dataset(torch.utils.data.Dataset):
  def __init__(self, transform, data, label):
    super(CIFAR10Dataset, self).__init__()
    self.transform = transform
    self.images = data
    self.labels = label

def __getitem__(self, idx):
  img = self.images[idx]
  img = self.transform(img)
  label = self.labels[idx]
  return img, label

def __len__(self):
  return len(self.images)
```

准备数据

```python
class CIFAR10Dataset(torch.utils.data.Dataset):
  def __init__(self, transform, data, label):
    super(CIFAR10Dataset, self).__init__() # 调用父类的构造函数

    self.transform = transform                  # 设置对象属性 transform

    self.images = data # 假设data的shape为（图片数，32，32，3）
                       # 假设data的数据类型是 np.float32，值域 [0,1]

    self.labels = label # 假设label的shape为（图片数，）
                        # PyTorch 会在计算交叉熵时
                        # 自动转为 one-hot 编码
```

准备数据

```
class CIFAR10Dataset(torch.utils.data.Dataset):
  ......

  def __getitem__(self, idx):
    img = self.images[idx]
    img = self.transform(img)
    label = self.labels[idx]
    return img, label

# 对于一个这个类的对象，可以用 len(obj) 来获取长度
def __len__(self):
    return len(self.images)
```

准备数据

```
# 定义 transform：包括两个顺序步骤
# 1 把numpy数组转化为pytorch张量
# 2 归一化到 [-0.5, 0.5]，有利于 ReLU
transform = transforms.Compose(
    [transforms.ToTensor(),
     transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))]
    )

# 假设我们把数据都做好了
train_data  = np.load('train_data.npy')
train_label = np.load('train_label.npy')
test_data   = np.load('test_data.npy')
test_label  = np.load('test_label.npy')
```

准备数据

```
# trainset 是一个 CIFAR10Dataset 实例，可以用下标索引
# 下标索引会返回一个 sample 的 data 和 label
trainset = CIFAR10Dataset(transform=transform,
                          data=train_data, label=train_label)

# PyTorch 提供的 dataloader 可以方便地控制 batchsize 和 shuffle
# 以及提供了异步接口。如果出现异步问题，设置num_workers=0
trainloader = torch.utils.data.DataLoader(trainset,
                    batch_size=4, shuffle=True, num_workers=2)
```

准备数据

```
# 类似地构造testset
testset = CIFAR10Dataset(transform=transform,
                         data=test_data, label=test_label)
testloader = torch.utils.data.DataLoader(testset,
                batch_size=4, shuffle=False, num_workers=2)

classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

迭代训练

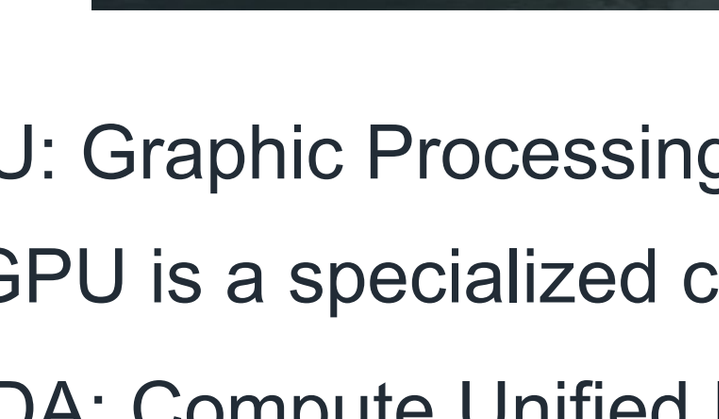
```
for epoch in range(10):
    for i, data in enumerate(trainloader, 0):
        # 获取输入数据，trainloader 每次会产生一个元组
        # 也就是 Dataset 返回的 data 和 label
        inputs, labels = data
        # 因为用 DataLoader 包装了，因此 PyTorch 会自动帮组成 batch
        # data.shape: (batchsize, 3, 32, 32)
        # label.shape: (batchsize, 10)

        # 如果使用GPU，要通过 inputs = inputs.cuda() 把数据放到GPU上
        # 同样的也要 labels = labels.cuda()
```

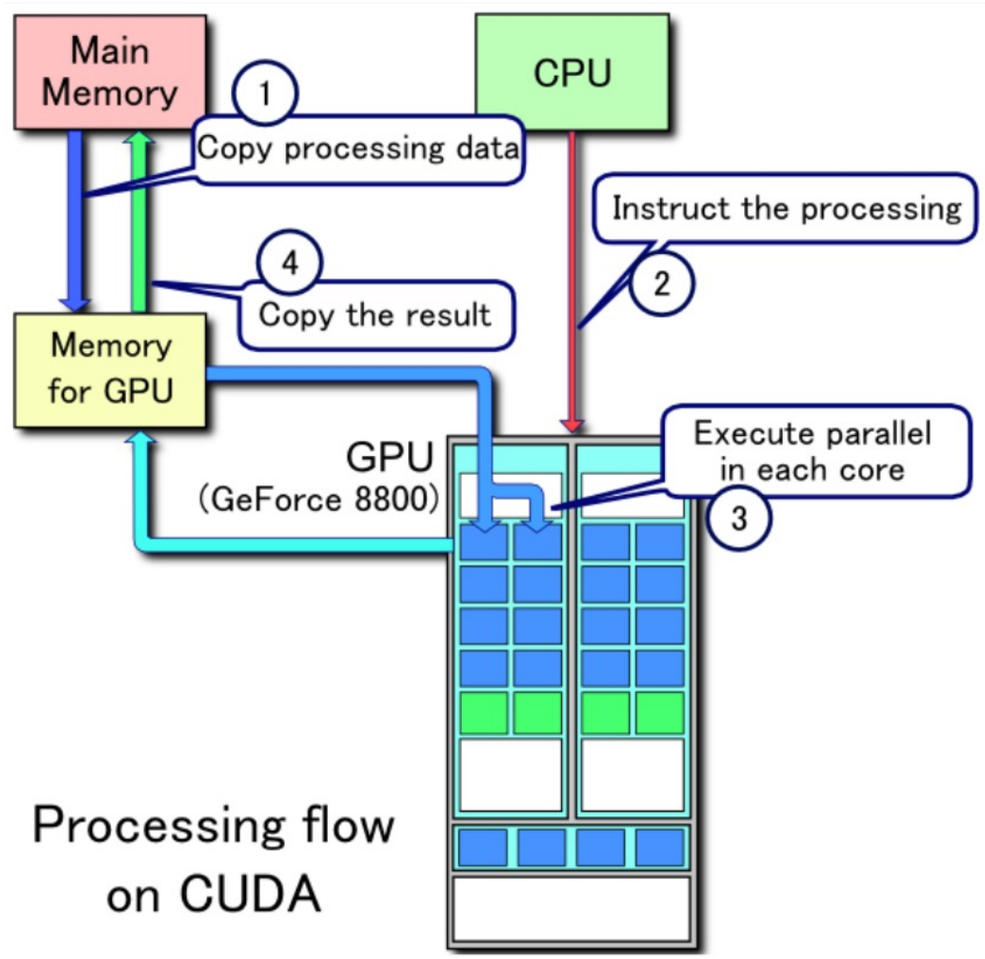
GPU

- GPU: Graphic Processing Unit
- GPU 对于计算机来说，更像是一种专用硬件而不是处理单元
  - CPU 发送“程序”到 GPU
  - CPU 发送数据到 GPU
  - GPU 执行计算
  - CPU　　GPU 取回结果
  - CPU 处理结果（显示，整合）
- 因此 GPU 的调用不是自然而然的

GPU 和 CUDA

- GPU: Graphic Processing Unit
  - GPU is a specialized computer processor
- CUDA: Compute Unified Device Architecture
  - CUDA is a parallel computing platform and programming model created by NVIDIA.





PyTorch GPU 调用

- 模型实例化之后，使用 net = net.cuda() 放到 GPU 上
- 数据从 loader 取出来之后，使用 x = x.cuda() 放到 GPU 上
- 想要把 PyTorch 计算结果转回numpy数组，调用 y.cpu().numpy()

迭代训练

```
for epoch in range(10):
    for i, data in enumerate(trainloader, 0):
        # 获取输入数据
        inputs, labels = data

        optimizer.zero_grad()
        # 前传
        outputs = net(inputs)
        # 计算 loss
        loss = criterion(outputs, labels)
        # 反传
        loss.backward()
        # 更新
        optimizer.step()
```

网络结构

```
import torch.nn as nn
import torch.nn.functional as F
# 继承一个nn.Module，实现了构造函数和forward方法，就是一个网络模型
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 2维卷积，输入通道3，输出通道6，卷积核大小5x5
        # 还有其它参数可以设置（stride, padding）
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # fc fully connected，全连接层
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)
```

网络结构

```
def forward(self, x):
    x = self.pool(F.relu(self.conv1(x)))
    x = self.pool(F.relu(self.conv2(x)))
    x = x.reshape(-1, 16 * 5 * 5)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = self.fc3(x)
    return x
```

迭代训练

```
# 实例化一个网络
net = Net()
# net = Net().cuda() 改成这个把网络放到GPU上

import torch.optim as optim
# 交叉熵，PyTorch 默认自带 SoftMax
criterion = nn.CrossEntropyLoss()

# Stochastic Gradient Descent
optimizer = optim.SGD(net.parameters(),
                      lr=0.001, momentum=0.9)
```

迭代训练

```
for epoch in range(10):
    for i, data in enumerate(trainloader, 0):
        # 获取输入数据
        inputs, labels = data

        optimizer.zero_grad()
        # 前传
        outputs = net(inputs)
        # 计算 loss
        loss = criterion(outputs, labels)
        # 反传
        loss.backward()
        # 更新
        optimizer.step()
```

迭代训练

```
for epoch in range(10):
  for i, data in enumerate(trainloader, 0):
    ……
    # 统计 training loss
    running_loss += loss.item()
  if i % 2000 == 1999: # print every 2000 mini-batches
    print('[%d, %5d] loss: %.3f' %
          (epoch + 1, i + 1, running_loss / 2000))
    running_loss = 0.0
    # 存模型
    PATH = './cifar_net.pth'
    torch.save(net.state_dict(), PATH))
```

测试训练好的网络

```
net = Net()
# 加载之前训好的模型参数
net.load_state_dict(torch.load(PATH))
cnt, correct = 0, 0

# 这部分不要反传
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)
        _, predicted = torch.max(outputs, 1)
        c = (predicted == labels).sum()
        correct += c
        cnt += data.shape[0]
acc = correct / cnt
```

测试训练好的网络

```
with torch.no_grad():
    for data in testloader:
        images, labels = data
        outputs = net(images)

        # torch.max 执行在outputs的维度1上
        # 每次会输出两项，即max和argmax
        _, predicted = torch.max(outputs, 1)

        # 一个简便的方法标出正确的预测
        c = (predicted == labels).sum()
        correct += c
        cnt += data.shape[0]
acc = correct / cnt
```