deep_learning

2025 年 10 月 31 日

1 一、PyTorch 基础

https://docs.pytorch.org/tutorials/beginner/basics/quickstart_tutorial.html ## 张量与自动求导

类似于数组、矩阵，但可在 GPU 上进行计算

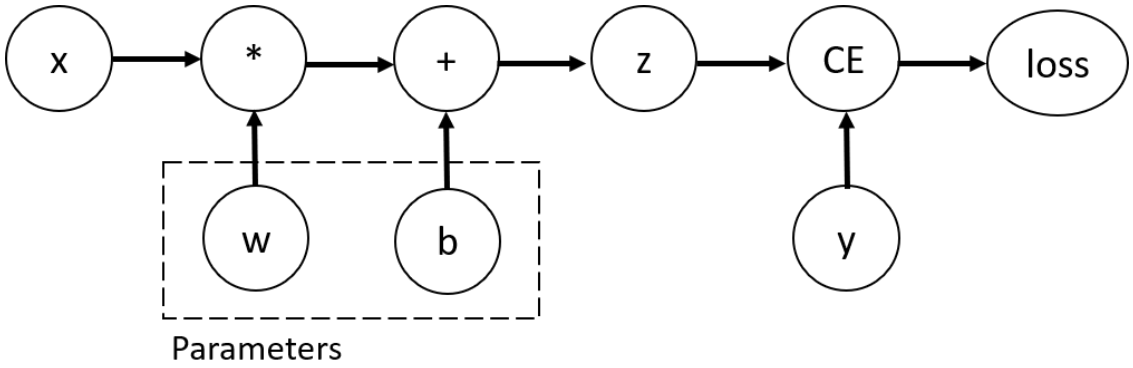
[2]:
```
import torch
import numpy as np
data = [[1, 2],[3, 4]]
data_inv = [[-2,1],[1.5,-0.5]]

print(torch.tensor(data))

tensor = torch.rand(3,4)
print(tensor)
print(f"Shape of tensor: {tensor.shape}")
print(f"Datatype of tensor: {tensor.dtype}")
print(f"Device tensor is stored on: {tensor.device}")
if torch.accelerator.is_available():
    tensor = tensor.to(torch.accelerator.current_accelerator())
print(f"Device tensor is stored on: {tensor.device}")

tensor = torch.tensor(data)
print(f"First row: {tensor[0]}")
print(f"First column: {tensor[:, 0]}")
print(f"Last column: {tensor[..., -1]}")
tensor[:,1] = 0
```

```
print(tensor)

print(torch.ones(2,2))
print(torch.cat([tensor,torch.ones(2,2)]))

tensor1 = torch.tensor(data, dtype=torch.float32)
tensor2 = torch.tensor(data_inv, dtype=torch.float32)

print("T1 @ T2:\n",tensor1 @ tensor2)
```

```
tensor([[1, 2],
        [3, 4]])
tensor([[0.5263, 0.0916, 0.8477, 0.6796],
        [0.3819, 0.5075, 0.1863, 0.6267],
        [0.2974, 0.8337, 0.8945, 0.5362]])
Shape of tensor: torch.Size([3, 4])
Datatype of tensor: torch.float32
Device tensor is stored on: cpu
Device tensor is stored on: cuda:0
First row: tensor([1, 2])
First column: tensor([1, 3])
Last column: tensor([2, 4])
tensor([[1, 0],
        [3, 0]])
tensor([[1., 1.],
        [1., 1.]])
tensor([[1., 0.],
        [3., 0.],
        [1., 1.],
        [1., 1.]])
T1 @ T2:
 tensor([[1., 0.],
        [0., 1.]])
```

梯度的自动运算

《PyTorch 深度学习实践》完结合集 _ 哔哩哔哩 _bilibili



[3]:
```python
import torch



x = torch.tensor([1.0, 2.0, 3.0])   # 输入向量
w = torch.tensor([4.0, 5.0, 6.0], requires_grad=True)  # 权重
b = torch.tensor(1.0, requires_grad=True)  # 偏置
y = torch.tensor(1.0)                                  # 目标值

# 前向传播: z = x * w + b
z = torch.dot(x, w) + b
# sigmoid 激活并计算二元交叉熵损失
loss = torch.nn.functional.binary_cross_entropy_with_logits(z, y)

# 自动求导
loss.backward()

print("===== Autograd 结果 =====")
print(f"z = {z.item():.4f}")
print(f"loss = {loss.item():.4f}")
print(f"w.grad = {w.grad}")
print(f"b.grad = {b.grad}")



x = torch.tensor(1.0, requires_grad=True)
# y = torch.sin(x) + torch.cos(2 * x)
y = x**3
y.backward()
```

```
print(x.grad)  # 输出梯度 dy/dx
```

```
===== Autograd 结果 =====
z = 33.0000
loss = 0.0000
w.grad = tensor([0., 0., 0.])
b.grad = 0.0
tensor(3.)
```

1.1 数据集加载与预处理

数据加载 `torch.utils.data.DataLoader` `torch.utils.data.Dataset`

[4]:
```
import torch
from torch.utils.data import Dataset
from torchvision import datasets
from torchvision.transforms import ToTensor
import matplotlib.pyplot as plt


training_data = datasets.FashionMNIST(
    root="data",
    train=True,
    download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data",
    train=False,
    download=True,
    transform=ToTensor()
)
#
# labels_map = {
#     0: "T-Shirt",
```

```
#     1: "Trouser",
#     2: "Pullover",
#     3: "Dress",
#     4: "Coat",
#     5: "Sandal",
#     6: "Shirt",
#     7: "Sneaker",
#     8: "Bag",
#     9: "Ankle Boot",
# }
# figure = plt.figure(figsize=(8, 8))
# cols, rows = 3, 3
# for i in range(1, cols * rows + 1):
#     sample_idx = torch.randint(len(training_data), size=(1,)).item()
#     img, label = training_data[sample_idx]
#     figure.add_subplot(rows, cols, i)
#     plt.title(labels_map[label])
#     plt.axis("off")
#     plt.imshow(img.squeeze(), cmap="gray")
# plt.show()


import os
import pandas as pd
from torchvision.io import decode_image

class CustomImageDataset(Dataset):
    def __init__(self, annotations_file, img_dir, transform=None,␣
 ↪target_transform=None):
        # 数据集初始化
        self.img_labels = pd.read_csv(annotations_file)
        self.img_dir = img_dir
        self.transform = transform
        self.target_transform = target_transform

    def __len__(self):
```

```
        # 数据集大小
        return len(self.img_labels)

    def __getitem__(self, idx):
        # 返回指定索引的数据
        img_path = os.path.join(self.img_dir, self.img_labels.iloc[idx, 0])
        image = decode_image(img_path)
        label = self.img_labels.iloc[idx, 1]
        if self.transform:
            image = self.transform(image)
        if self.target_transform:
            label = self.target_transform(label)
        return image, label


from torch.utils.data import DataLoader

train_dataloader = DataLoader(training_data, batch_size=64, shuffle=True)
test_dataloader = DataLoader(test_data, batch_size=64, shuffle=True)
# for epoch in range(1):
#     for batch_idx, (inputs, labels) in enumerate(train_dataloader):
#         print(f'Batch {batch_idx + 1}:')
#         print(f'Inputs: {inputs}')
#         print(f'Labels: {labels}')

# Display image and label.
# train_features, train_labels = next(iter(train_dataloader))
# print(f"Feature batch shape: {train_features.size()}")
# print(f"Labels batch shape: {train_labels.size()}")
# img = train_features[0].squeeze()
# label = train_labels[0]
# plt.imshow(img, cmap="gray")
# plt.show()
# print(f"Label: {label}")
```

数据预处理 torchvision.transforms 是一个用于图像数据预处理的模块。

它的作用是把原始数据（通常是 PIL 图片或 NumPy 数组）转换成神经网络可以处理的形式（如 Tensor），并进行一些增强（旋转、裁剪、归一化等）。

transforms.Lambda() 作用

Lambda 可以让你自定义任意变换函数。

它接收一个函数作为参数，这个函数会应用到每个样本上。

torchvision.transforms 模块来进行常见的图像预处理和增强操作，如旋转、裁剪、归一化等。

[5]:
```
import torchvision.transforms as transforms
from PIL import Image

# 定义数据预处理的流水线
transform = transforms.Compose([
    transforms.Resize((128, 128)),  # 将图像调整为 128x128
    transforms.ToTensor(),  # 将图像转换为张量
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])␣
 ↪ # 标准化
])
#
# # 加载图像
# image = Image.open('image.jpg')
#
# # 应用预处理
# image_tensor = transform(image)
# print(image_tensor.shape)  # 输出张量的形状
#
# transform = transforms.Compose([
#     transforms.RandomHorizontalFlip(),  # 随机水平翻转
#     transforms.RandomRotation(30),  # 随机旋转 30 度
#     transforms.RandomResizedCrop(128),  # 随机裁剪并调整为 128x128
#     transforms.ToTensor(),
#     transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.
 ↪225])
# ])
```

2 二、利用 PyTorch 构建神经网络

https://www.runoob.com/pytorch/pytorch-neural-network.html

PyTorch 提供了许多常见的神经网络层，以下是几个常见的：

1. nn.Linear(in_features, out_features)：全连接层，输入 in_features 个特征，输出 out_features 个特征。
2. nn.Conv2d(in_channels, out_channels, kernel_size)：2D 卷积层，用于图像处理。
3. nn.MaxPool2d(kernel_size)：2D 最大池化层，用于降维。
4. nn.ReLU()：ReLU 激活函数，常用于隐藏层。
5. nn.Softmax(dim)：Softmax 激活函数，通常用于输出层，适用于多类分类问题。

2.1 激活函数（Activation Function）

激活函数决定了神经元是否应该被激活。它们是非线性函数，使得神经网络能够学习和执行更复杂的任务。常见的激活函数包括：

1. Sigmoid：用于二分类问题，输出值在 0 和 1 之间。
2. Tanh：输出值在 -1 和 1 之间，常用于输出层之前。
3. ReLU（Rectified Linear Unit）：目前最流行的激活函数之一，定义为 f(x) = max(0, x)，有助于解决梯度消失问题。
4. Softmax：常用于多分类问题的输出层，将输出转换为概率分布。

2.2 损失函数（Loss Function）

损失函数用于衡量模型的预测值与真实值之间的差异。

常见的损失函数包括：

1. 均方误差（MSELoss）：回归问题常用，计算输出与目标值的平方差。
2. 交叉熵损失（CrossEntropyLoss）：分类问题常用，计算输出和真实标签之间的交叉熵。
3. BCEWithLogitsLoss：二分类问题，结合了 Sigmoid 激活和二元交叉熵损失。

MSELoss - 输入 output 和 Y 必须形状一致 - 用于回归问题

CrossEntropyLoss

- 用于多分类（类别标签是整数）
- 输入 output：[batch, num_classes] → logits

- 标签 Y：[batch] → 每个样本的类别索引

BCEWithLogitsLoss - 用于二分类或多标签分类 - 输入 output：[batch, 1] 或 [batch, num_classes] - 标签 Y：[0,1] 的 float

2.3 优化器（Optimizer）

优化器负责在训练过程中更新网络的权重和偏置。

常见的优化器包括：

1. SGD（随机梯度下降）
2. Adam（自适应矩估计）
3. RMSprop（均方根传播）

2.4 训练过程（Training Process）

训练神经网络涉及以下步骤：

1. 准备数据：通过 DataLoader 加载数据。
2. 定义损失函数和优化器。
3. 前向传播：计算模型的输出。
4. 计算损失：与目标进行比较，得到损失值。
5. 反向传播：通过 loss.backward() 计算梯度。
6. 更新参数：通过 optimizer.step() 更新模型的参数。
7. 重复上述步骤，直到达到预定的训练轮数。

[6]:
```
from torch import optim
import torch
import torch.nn as nn

# 定义一个简单的神经网络模型
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        # 使用 nn.Sequential 定义顺序层
        self.layer = nn.Sequential(
            nn.Linear(2,2),
```

```
            nn.ReLU(),
            nn.Linear(2,2),
        )

    def forward(self, x):
        # 前向传播直接用顺序模块
        x = self.layer(x)
        return x

# 创建模型实例
model = SimpleNN()
print(model)


# 假设已经定义好了模型、损失函数和优化器

# 训练数据示例
X = torch.randn(10,2)
Y = torch.randint(0,2,(10,))  # 分类标签 0 或 1

# 优化器 & 损失
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

# 训练循环
for epoch in range(100):
    model.train()
    optimizer.zero_grad()
    output = model(X)           # [10,2] logits
    loss = criterion(output, Y)
    loss.backward()
    optimizer.step()

    if (epoch+1)%10==0:
        print(f'Epoch {epoch+1}, Loss={loss.item():.4f}')
```

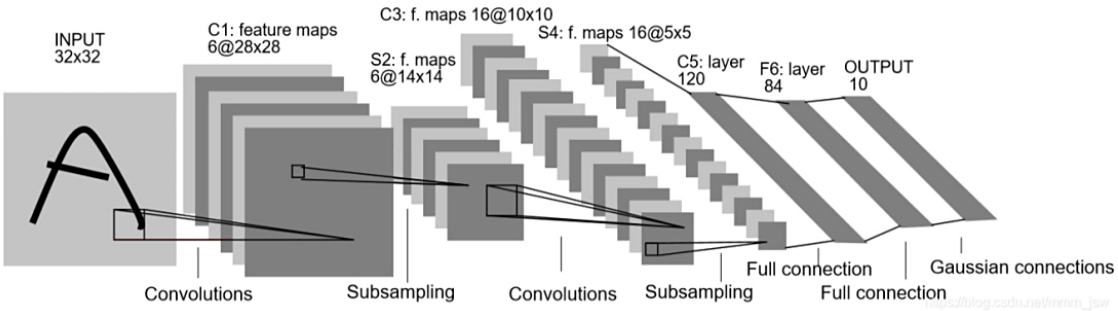
```
# 保存模型参数
torch.save(model.state_dict(), "simple_nn.pth")
print("模型参数已保存到 simple_nn.pth")

model = SimpleNN()              # 先实例化网络
model.load_state_dict(torch.load("simple_nn.pth"))
model.eval()                    # 切换到评估模式
```

```
SimpleNN(
  (layer): Sequential(
    (0): Linear(in_features=2, out_features=2, bias=True)
    (1): ReLU()
    (2): Linear(in_features=2, out_features=2, bias=True)
  )
)
Epoch 10, Loss=0.7469
Epoch 20, Loss=0.7397
Epoch 30, Loss=0.7326
Epoch 40, Loss=0.7255
Epoch 50, Loss=0.7188
Epoch 60, Loss=0.7128
Epoch 70, Loss=0.7067
Epoch 80, Loss=0.7005
Epoch 90, Loss=0.6942
Epoch 100, Loss=0.6878
模型参数已保存到 simple_nn.pth
```

[6]:
```
SimpleNN(
  (layer): Sequential(
    (0): Linear(in_features=2, out_features=2, bias=True)
    (1): ReLU()
    (2): Linear(in_features=2, out_features=2, bias=True)
  )
)
```

3 三、卷积神经网络（CNN）

3.1 LeNet



层编号	层类型	输入尺寸	输出尺寸	核心操作	激活函数	作用
1	输入层	1×32×32	-	灰度图像	-	padding 防止边缘损失
2	卷积层 C1	1×32×32	6×28×28	Conv(5×5)，stride=1，无 padding	Tanh	提取局部低层特征
3	池化层 S2	6×28×28	6×14×14	AvgPool(2×2)	-	降低分辨率减小计算量，保留局部平均特征
4	卷积层 C3	6×14×14	16×10×10	Conv(5×5)	Tanh	提取更高层、更复杂的特征组合
5	池化层 S4	16×10×10	16×5×5	AvgPool(2×2)	-	进一步降采样，减小维度
6	全连接层 F5	16×5×5	120	Linear	Tanh	将 2D 特征图展平成 1D 向量
7	全连接层 F6	120	84	Linear	Tanh	进一步特征组合和非线性变换
8	输出层	84	10	Linear	Softmax(隐式)	将特征映射到类别

设计思想 | 设计思想 | 含义 | 现代延续 | | :——— | :————— | :————- | | 局部连接 | 每个神经元只与局部区域相连，而非全连接 | 卷积层的核心思想 | | 权值共享 | 同一卷积核在整张图

片上滑动 | 大大减少参数量 | | 池化层 | 提取局部统计信息，增强平移不变性 | 现代 CNN 仍保留池化或替代结构 | | 分层特征抽取 | 从低层到高层逐步提取特征 | 所有深度网络的基础思想 | | 使用 tanh 激活 | 早期激活函数选择，防止非线性消失 | 现代版本多用 ReLU 取代 |

[7]:
```
# LeNet5
# LeNet-5 训练 MNIST 手写数字识别
import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# ================================
# 1. 数据加载与预处理
# ================================
transform = transforms.Compose([
    transforms.Resize((32, 32)),
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])

train_dataset = datasets.MNIST(root='./data', train=True, transform=transform,␣
 ↪download=True)
test_dataset = datasets.MNIST(root='./data', train=False, transform=transform,␣
 ↪download=True)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=1000, shuffle=False)

# ================================
# 2. 定义 LeNet-5 模型
# ================================
class LeNet5(nn.Module):
    def __init__(self):
        super(LeNet5, self).__init__()
        self.conv1 = nn.Conv2d(1, 6, kernel_size=5)   # 输入 1 通道，输出 6 通道
```

```
        self.conv2 = nn.Conv2d(6, 16, kernel_size=5)  # 输入 6 通道，输出 16 通道
        self.fc1 = nn.Linear(16*5*5, 120)             # 展平后输入全连接层
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)                  # 输出 10 类（0-9）

    def forward(self, x):
        x = F.tanh(self.conv1(x))
        x = F.avg_pool2d(x, 2)
        x = F.tanh(self.conv2(x))
        x = F.avg_pool2d(x, 2)
        x = x.view(-1, 16*5*5)
        x = F.tanh(self.fc1(x))
        x = F.tanh(self.fc2(x))
        x = self.fc3(x)
        return x

# ================================
# 3. 训练与测试函数
# ================================
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = LeNet5().to(device)
optimizer = optim.Adam(model.parameters(), lr=0.001)
criterion = nn.CrossEntropyLoss()

def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad()
        output = model(data)
        loss = criterion(output, target)
        loss.backward()
        optimizer.step()

        if batch_idx % 100 == 0:
```

```
            print(f"Train Epoch: {epoch} [{batch_idx * len(data)}/
 ↪{len(train_loader.dataset)} "
                  f"({100. * batch_idx / len(train_loader):.0f}%)]\tLoss: {loss.
 ↪item():.6f}")

def test():
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item()
            pred = output.argmax(dim=1, keepdim=True)
            correct += pred.eq(target.view_as(pred)).sum().item()
    test_loss /= len(test_loader)
    print(f"\nTest set: Average loss: {test_loss:.4f}, "
          f"Accuracy: {correct}/{len(test_loader.dataset)} "
          f"({100. * correct / len(test_loader.dataset):.2f}%)\n")

# ================================
# 4. 主训练循环
# ================================
for epoch in range(1, 6):  # 训练 5 轮即可达到约 99% 准确率
    train(epoch)
    test()

# ================================
# 5. 保存模型
# ================================
torch.save(model.state_dict(), "lenet5_mnist.pth")
print("模型已保存为 lenet5_mnist.pth")
```

```
Train Epoch: 1 [0/60000 (0%)]   Loss: 2.303271
Train Epoch: 1 [6400/60000 (11%)]       Loss: 0.405319
Train Epoch: 1 [12800/60000 (21%)]      Loss: 0.279895
```

```
Train Epoch: 1 [19200/60000 (32%)]      Loss: 0.145995
Train Epoch: 1 [25600/60000 (43%)]      Loss: 0.131634
Train Epoch: 1 [32000/60000 (53%)]      Loss: 0.123086
Train Epoch: 1 [38400/60000 (64%)]      Loss: 0.067663
Train Epoch: 1 [44800/60000 (75%)]      Loss: 0.232855
Train Epoch: 1 [51200/60000 (85%)]      Loss: 0.178002
Train Epoch: 1 [57600/60000 (96%)]      Loss: 0.136303

Test set: Average loss: 0.0810, Accuracy: 9747/10000 (97.47%)

Train Epoch: 2 [0/60000 (0%)]   Loss: 0.038945
Train Epoch: 2 [6400/60000 (11%)]       Loss: 0.078147
Train Epoch: 2 [12800/60000 (21%)]      Loss: 0.072432
Train Epoch: 2 [19200/60000 (32%)]      Loss: 0.075886
Train Epoch: 2 [25600/60000 (43%)]      Loss: 0.067637
Train Epoch: 2 [32000/60000 (53%)]      Loss: 0.077059
Train Epoch: 2 [38400/60000 (64%)]      Loss: 0.039917
Train Epoch: 2 [44800/60000 (75%)]      Loss: 0.134900
Train Epoch: 2 [51200/60000 (85%)]      Loss: 0.007279
Train Epoch: 2 [57600/60000 (96%)]      Loss: 0.160271

Test set: Average loss: 0.0493, Accuracy: 9852/10000 (98.52%)

Train Epoch: 3 [0/60000 (0%)]   Loss: 0.116203
Train Epoch: 3 [6400/60000 (11%)]       Loss: 0.013806
Train Epoch: 3 [12800/60000 (21%)]      Loss: 0.063120
Train Epoch: 3 [19200/60000 (32%)]      Loss: 0.121587
Train Epoch: 3 [25600/60000 (43%)]      Loss: 0.017577
Train Epoch: 3 [32000/60000 (53%)]      Loss: 0.035370
Train Epoch: 3 [38400/60000 (64%)]      Loss: 0.052557
Train Epoch: 3 [44800/60000 (75%)]      Loss: 0.014256
Train Epoch: 3 [51200/60000 (85%)]      Loss: 0.088636
Train Epoch: 3 [57600/60000 (96%)]      Loss: 0.031196

Test set: Average loss: 0.0528, Accuracy: 9838/10000 (98.38%)

Train Epoch: 4 [0/60000 (0%)]   Loss: 0.125415
```

```
Train Epoch: 4 [6400/60000 (11%)]       Loss: 0.021453
Train Epoch: 4 [12800/60000 (21%)]      Loss: 0.041542
Train Epoch: 4 [19200/60000 (32%)]      Loss: 0.009439
Train Epoch: 4 [25600/60000 (43%)]      Loss: 0.003189
Train Epoch: 4 [32000/60000 (53%)]      Loss: 0.107843
Train Epoch: 4 [38400/60000 (64%)]      Loss: 0.006169
Train Epoch: 4 [44800/60000 (75%)]      Loss: 0.050137
Train Epoch: 4 [51200/60000 (85%)]      Loss: 0.011617
Train Epoch: 4 [57600/60000 (96%)]      Loss: 0.016562

Test set: Average loss: 0.0458, Accuracy: 9862/10000 (98.62%)

Train Epoch: 5 [0/60000 (0%)]   Loss: 0.035851
Train Epoch: 5 [6400/60000 (11%)]       Loss: 0.001429
Train Epoch: 5 [12800/60000 (21%)]      Loss: 0.096324
Train Epoch: 5 [19200/60000 (32%)]      Loss: 0.007473
Train Epoch: 5 [25600/60000 (43%)]      Loss: 0.020111
Train Epoch: 5 [32000/60000 (53%)]      Loss: 0.061460
Train Epoch: 5 [38400/60000 (64%)]      Loss: 0.035190
Train Epoch: 5 [44800/60000 (75%)]      Loss: 0.006228
Train Epoch: 5 [51200/60000 (85%)]      Loss: 0.084266
Train Epoch: 5 [57600/60000 (96%)]      Loss: 0.117010

Test set: Average loss: 0.0413, Accuracy: 9876/10000 (98.76%)
```

模型已保存为 `lenet5_mnist.pth`

3.2 AlexNet

- 更深的神经网络结构
- ReLU 激活函数的使用
- 局部响应归一化（LRN）的使用

LRN是在卷积层和池化层之间添加的一种归一化操作。在卷积层中，每个卷积核都对应一个特征图（feature map），LRN就是对这些特征图进行归一化。具体来说，对于每个特征图上的每个位置，计算该位置周围的像素的平方和，然后将当前位置的像素值除以这个和。计算过程可以用以下公式表示：

$$b_{x,y}^i = \frac{a_{x,y}^i}{(k + \alpha \sum_{j=\max(0,i-n/2)}^{\min(N-1,i+n/2)} (a_{x,y}^j)^2)^\beta}$$

其中，$b_{x,y}^i$ 表示第 i 个特征图在位置 (x, y) 的归一化值，$a_{x,y}^i$ 表示第 i 个特征图在位置 (x, y) 的原始值，n 表示邻域大小，N 表示特征图数量，k、α 和 β 是超参数，通常取 $k = 2$、$\alpha = 10^{-4}$ 和 $\beta = 0.75$。

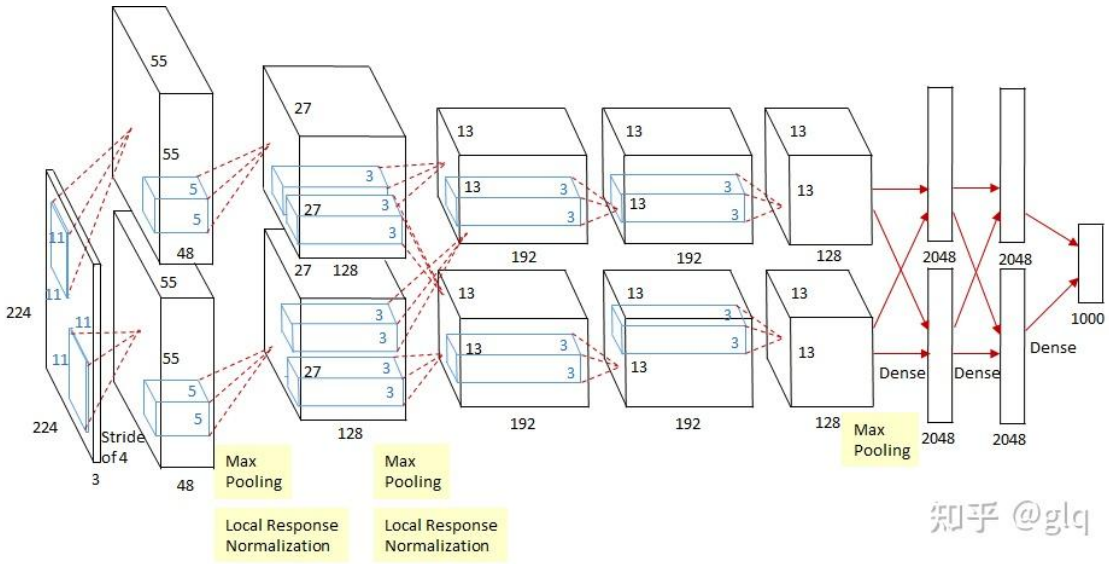
LRN本质是抑制邻近神经元的响应，从而增强了神经元的较大响应。这种技术在一定程度上能够避免过拟合，并提高网络的泛化能力。

- 数据增强和 Dropout

为了防止过拟合，AlexNet 引入了数据增强和 Dropout 技术。数据增强可以通过对图像进行旋转、翻转、裁剪等变换，增加训练数据的多样性，提高模型的泛化能力。Dropout 则是在训练过程中随机删除一定比例的神经元，强制网络学习多个互不相同的子网络，从而提高网络的泛化能力。Dropout 简单来说就是在前向传播的时候，让某个神经元的激活值以一定的概率 p 停止工作，这样可以使模型泛化性更强，因为它不会太依赖某些局部的特征。

- 大规模分布式训练

卷积神经网络经典回顾之 AlexNet



层次	类型	参数	输入大小	输出计算	输出大小
1	Conv1	kernel=11, stride=4, pad=2, out_ch=96	224×224×3	$(224+2\times2-11)/4+1=55$	55×55×96
2	ReLU	—	55×55×96	不改变尺寸	55×55×96
3	MaxPool1	kernel=3, stride=2	55×55×96	$((55-3)/2+1=27)$	27×27×96
4	Conv2	kernel=5, stride=1, pad=2, out_ch=256	27×27×96	$((27+2\times2-5)/1+1=27)$	27×27×256
5	ReLU	—	27×27×256	不变	27×27×256
6	MaxPool2	kernel=3, stride=2	27×27×256	$(27-3)/2+1=13)$	13×13×256
7	Conv3	kernel=3, stride=1, pad=1, out_ch=384	13×13×256	$(13+2\times1-3)/1+1=13)$	13×13×384
8	Conv4	kernel=3, stride=1, pad=1, out_ch=384	13×13×384	$(13+2\times1-3)/1+1=13)$	13×13×384
9	Conv5	kernel=3, stride=1, pad=1, out_ch=256	13×13×384	$(13+2\times1-3)/1+1=13)$	13×13×256
10	MaxPool3	kernel=3, stride=2	13×13×256	$(13-3)/2+1=6)$	6×6×256
11	Flatten	—	6×6×256	展平	9216
12	FC1	Linear(9216→4096)	9216	—	4096
13	FC2	Linear(4096→4096)	4096	—	4096
14	FC3	Linear(4096→1000)	4096	—	1000

[8]:
```python
import torch
import torch.nn as nn

class AlexNet(nn.Module):
    def __init__(self, config):
        super(AlexNet, self).__init__()
        self._config = config
        # 定义卷积层和池化层
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, stride=1, padding=2),
            nn.ReLU(inplace=True),
```

```
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        # 自适应层，将上一层的数据转换成 6x6 大小
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        # 全连接层
        self.classifier = nn.Sequential(
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 1024),
            nn.ReLU(inplace=True),
            nn.Linear(1024, self._config['num_classes']),
        )

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x


def saveModel(self):
        torch.save(self.state_dict(), self._config['model_name'])

def loadModel(self, map_location):
        state_dict = torch.load(self._config['model_name'],␣
 ↪map_location=map_location)
```

```
        self.load_state_dict(state_dict, strict=False)


import torchvision
from torch.utils.data import DataLoader
import torchvision.transforms as transforms
# 定义构造数据加载器的函数
def Construct_DataLoader(dataset, batchsize):
    return DataLoader(dataset=dataset, batch_size=batchsize, shuffle=True)
# 图像预处理
transform = transforms.Compose([
    transforms.Resize(96),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
])
# 加载 CIFAR-10 数据集函数
def LoadCIFAR10(download=False):
    # Load CIFAR-10 dataset
    train_dataset = torchvision.datasets.CIFAR10(root='../CIFAR10', train=True,␣
 ↪transform=transform, download=download)
    test_dataset = torchvision.datasets.CIFAR10(root='../CIFAR10', train=False,␣
 ↪transform=transform)
    return train_dataset[:10], test_dataset[:10]

from torch.autograd import Variable
from torch.utils.data import DataLoader

import torch.nn as nn

class Trainer(object):
    # 初始化模型、配置参数、优化器和损失函数
    def __init__(self, model, config):
        self._model = model
        self._config = config
        self._optimizer = torch.optim.Adam(self._model.parameters(),\
```

```
                                            lr=config['lr'],␣
↪weight_decay=config['l2_regularization'])
        self.loss_func = nn.CrossEntropyLoss()
    # 对单个小批量数据进行训练，包括前向传播、计算损失、反向传播和更新模型参数
    def _train_single_batch(self, images, labels):
        y_predict = self._model(images)

        loss = self.loss_func(y_predict, labels)
        # 先将梯度清零，如果不清零，那么这个梯度就和上一个 mini-batch 有关
        self._optimizer.zero_grad()
        # 反向传播计算梯度
        loss.backward()
        # 梯度下降等优化器 更新参数
        self._optimizer.step()
        # 将 loss 的值提取成 python 的 float 类型
        loss = loss.item()

        # 计算训练精确度
        # 这里的 y_predict 是一个多个分类输出，将 dim 指定为 1，即返回每一个分类输
出最大的值以及下标
        _, predicted = torch.max(y_predict.data, dim=1)
        return loss, predicted

    def _train_an_epoch(self, train_loader, epoch_id):
        """
        训练一个 Epoch，即将训练集中的所有样本全部都过一遍
        """
        # 设置模型为训练模式，启用 dropout 以及 batch normalization
        self._model.train()
        total = 0
        correct = 0

        # 从 DataLoader 中获取小批量的 id 以及数据
        for batch_id, (images, labels) in enumerate(train_loader):
            images = Variable(images)
            labels = Variable(labels)
```

```
            if self._config['use_cuda'] is True:
                images, labels = images.cuda(), labels.cuda()

            loss, predicted = self._train_single_batch(images, labels)

            # 计算训练精确度
            total += labels.size(0)
            correct += (predicted == labels.data).sum()

            # print('[Training Epoch: {}] Batch: {}, Loss: {}'.format(epoch_id,␣
↪batch_id, loss))
        print('Training Epoch: {}, accuracy rate: {}%%'.format(epoch_id,␣
↪correct / total * 100.0))

    def train(self, train_dataset):
        # 是否使用 GPU 加速
        self.use_cuda()
        for epoch in range(self._config['num_epoch']):
            print('-' * 20 + ' Epoch {} starts '.format(epoch) + '-' * 20)
            # 构造 DataLoader
            data_loader = DataLoader(dataset=train_dataset, batch_size=self.
↪_config['batch_size'], shuffle=True)
            # 训练一个轮次
            self._train_an_epoch(data_loader, epoch_id=epoch)

    # 用于将模型和数据迁移到 GPU 上进行计算，如果 CUDA 不可用则会抛出异常
    def use_cuda(self):
        if self._config['use_cuda'] is True:
            assert torch.cuda.is_available(), 'CUDA is not available'
            torch.cuda.set_device(self._config['device_id'])
            self._model.cuda()

    # 保存训练好的模型
    def save(self):
        self._model.saveModel()
```

```
from torch.autograd import Variable
# 定义参数配置信息
alexnet_config = \
{
    'num_epoch': 20,              # 训练轮次数
    'batch_size': 500,            # 每个小批量训练的样本数量
    'lr': 1e-3,                   # 学习率
    'l2_regularization':1e-4,     # L2 正则化系数
    'num_classes': 10,            # 分类的类别数目
    'device_id': 0,               # 使用的 GPU 设备的 ID 号
    'use_cuda': True,             # 是否使用 CUDA 加速
    'model_name': '../AlexNet.model' # 保存模型的文件名
}
#
# if __name__ == "__main__":
#     ␣
 ↪##############################################################################
#     # AlexNet 模型
#     ␣
 ↪##############################################################################
#     train_dataset, test_dataset = LoadCIFAR10(True)
#     # define AlexNet model
#     alexNet = AlexNet(alexnet_config)
#
#     ␣
 ↪##############################################################################
#     # 模型训练阶段
#     ␣
 ↪##############################################################################
#     # # 实例化模型训练器
#     trainer = Trainer(model=alexNet, config=alexnet_config)
#     # # 训练
#     trainer.train(train_dataset)
#     # # 保存模型
#     trainer.save()
#
```

```
#     ␣
 ↪#############################################################################################
#     # 模型测试阶段
#     ␣
 ↪#############################################################################################
#     alexNet.eval()
#     alexNet.loadModel(map_location=torch.device('cpu'))
#     if alexnet_config['use_cuda']:
#         alexNet = alexNet.cuda()
#
#     correct = 0
#     total = 0
#    # 对测试集中的每个样本进行预测，并计算出预测的精度
#     for images, labels in Construct_DataLoader(test_dataset,␣
 ↪alexnet_config['batch_size']):
#         images = Variable(images)
#         labels = Variable(labels)
#         if alexnet_config['use_cuda']:
#             images = images.cuda()
#             labels = labels.cuda()
#
#         y_pred = alexNet(images)
#         _, predicted = torch.max(y_pred.data, 1)
#         total += labels.size(0)
#         temp = (predicted == labels.data).sum()
#         correct += temp
#     print('Accuracy of the model on the test images: %.2f%%' % (100.0 *␣
 ↪correct / total))
```

3.3 ResNet 残差神经网络

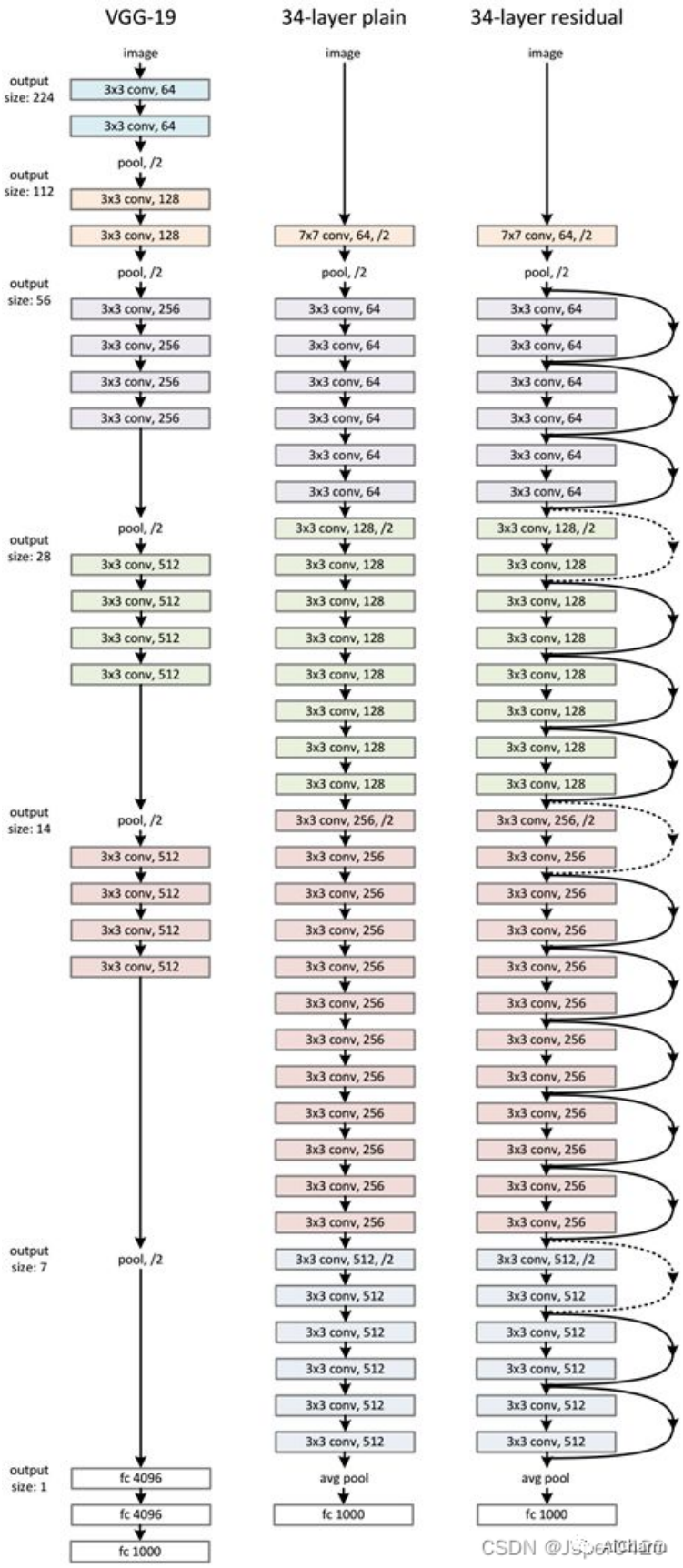
随着网络层级的不断增加，模型精度不断得到提升，而当网络层级增加到一定的数目以后，训练精度和测试精度迅速下降，这说明当网络变得很深以后，深度网络就变得更加难以训练了。是因为神经网络在反向传播过程中要不断地传播梯度，而当网络层数加深时，梯度在传播过程中会逐渐消失（假如采用 Sigmoid 函数，对于幅度为 1 的信号，每向后传递一层，梯度就衰减为原来的 0.25，层数越多，衰减越厉害），导致无法对前面网络层的权重进行有效的调整。

批标准化（batch normlization）和正则化可以解决了梯度传播的问题，但网络性能可能会退化。深度加深了，错误率却上升了。

为了使深度加深的同时性能不退化提出残差网络。

https://cloud.tencent.com/developer/article/2286760

ResNet 网络参考了 VGG19 网络，在其基础上进行了修改，并通过短路机制加入了残差单元。


VGG-19
34-layer plain
34-layer residual
image
output size: 224
3x3 conv, 64
3x3 conv, 64
pool, /2
output size: 112
3x3 conv, 128
3x3 conv, 128
pool, /2
output size: 56
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
3x3 conv, 256
pool, /2
output size: 28
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 14
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
3x3 conv, 512
pool, /2
output size: 7
fc 4096
fc 4096
fc 1000
output size: 1
7x7 conv, 64, /2
3x3 conv, 128, /2
3x3 conv, 256, /2
3x3 conv, 512, /2
avg pool
fc 1000
CSDN @JSpeedChen

阶段	层次	类型	参数配置	输入尺寸	输出计算	输出尺寸
输入阶段	1	**Conv1**	kernel=7×7, stride=2, pad=3, out_ch=64	224×224×3	(224+2×3−7)/2+1=112	**112×112×64**
	2	**MaxPool1**	kernel=3×3, stride=2	112×112×64	(112−3)/2+1=56	**56×56×64**
Conv2_x 阶段	3	**Residual Block ×3**	[1×1,64; 3×3,64; 1×1,256]	56×56×64	输出通道 256	**56×56×256**
Conv3_x 阶段	4	**Residual Block ×4**	[1×1,128; 3×3,128; 1×1,512] (stride=2)	56×56×256	空间减半、通道加倍	**28×28×512**
Conv4_x 阶段	5	**Residual Block ×6**	[1×1,256; 3×3,256; 1×1,1024] (stride=2)	28×28×512	空间减半、通道加倍	**14×14×1024**
Conv5_x 阶段	6	**Residual Block ×3**	[1×1,512; 3×3,512; 1×1,2048] (stride=2)	14×14×1024	空间减半、通道加倍	**7×7×2048**
分类阶段	7	**AvgPool**	kernel=7×7	7×7×2048	输出平均池化	**1×1×2048**
	8	**FC（全连接）**	Linear(2048→1000)	2048	—	**1000 类输出**

阶段	名称	输入通道	输出通道	特征图尺寸	Block 数	作用
输入	Conv1+Pool	3 → 64	64	224→56	—	初步提取边缘特征
Conv2_x	Stage 1	64 → 256	256	56×56	3	第一组残差学习，保持分辨率
Conv3_x	Stage 2	256 → 512	512	28×28	4	降采样 + 特征加深
Conv4_x	Stage 3	512 → 1024	1024	14×14	6	语义层特征抽取
Conv5_x	Stage 4	1024 → 2048	2048	7×7	3	高级语义，分类前特征
分类	Pool + FC	2048	1000	1×1	—	输出 ImageNet 1000 类

有了残差链接之后假设卷积层学习的变换为 ( )，残差结构的输出是 ( )，则有：

( ) = ( ) + 那计算梯度（求导）的时候：'( ) = '( ) + 1

无论 '( ) 为多大，总的梯度至少有个 1，从而保证了梯度的回传，网络得到更好的训练。

也就是说：

残差结构能够避免普通的卷积层堆叠存在信息丢失问题，保证前向信息流的顺畅。残差结构能够应

对梯度反传过程中的梯度消失问题，保证反向梯度流的通顺。

从上图图可以看出，怎么有一些跳跃链接是实线，有一些是虚线。

因为经过跳跃链接后，H(x)=F(x)+x，如果 F(x) 和 x 的通道相同，则可直接相加。若通道不同：

实线的 Connection 部分，表示通道相同，如上图的第一个粉色矩形和第三个粉色矩形，都是 3x3x64 的特征图，由于通道相同，所以采用计算方式为 ( ) = ( ) +

虚线的的 Connection 部分，表示通道不同，如上图的第一个绿色矩形和第三个绿色矩形，分别是 3x3x64 和 3x3x128 的特征图，通道不同，采用的计算方式为 ( ) = ( ) + w ，其中 w 是卷积操作，用来调整 x 维度的。

[1]:
```
import torch
import torch.nn as nn
import torch.nn.functional as F

# ============================================================
# 定义 Bottleneck 残差块（ResNet-50 的核心结构）
# ============================================================
class Bottleneck(nn.Module):
    expansion = 4  # 输出通道扩展倍数：ResNet-50/101 使用 4 倍

    def __init__(self, in_channels, mid_channels, stride=1, downsample=None):
        super(Bottleneck, self).__init__()
        # 1×1 降维
        self.conv1 = nn.Conv2d(in_channels, mid_channels, kernel_size=1,␣
↪bias=False)
        self.bn1 = nn.BatchNorm2d(mid_channels)
        # 3×3 卷积
        self.conv2 = nn.Conv2d(mid_channels, mid_channels, kernel_size=3,
                               stride=stride, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(mid_channels)
        # 1×1 升维
        self.conv3 = nn.Conv2d(mid_channels, mid_channels * self.expansion,␣
↪kernel_size=1, bias=False)
        self.bn3 = nn.BatchNorm2d(mid_channels * self.expansion)

        self.relu = nn.ReLU(inplace=True)
```

```
        self.downsample = downsample  # 如果输入输出尺寸不匹配，用于捷径分支
        self.stride = stride

    def forward(self, x):
        identity = x  # 保存输入以用于残差连接

        # 主分支
        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
        out = self.relu(out)

        out = self.conv3(out)
        out = self.bn3(out)

        # 如果通道或尺寸不匹配，则捷径也进行卷积调整
        if self.downsample is not None:
            identity = self.downsample(x)

        # 残差相加
        out += identity
        out = self.relu(out)

        return out


# ============================================================
# 定义 ResNet 主体结构
# ============================================================
class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=1000):
        """
        block: 残差块类型（Bottleneck）
```

```
        layers: 每个阶段的残差块数量列表 [3, 4, 6, 3]
        """
        super(ResNet, self).__init__()
        self.in_channels = 64  # 初始通道数

        # 输入阶段
        self.conv1 = nn.Conv2d(3, 64, kernel_size=7, stride=2, padding=3,␣
↪bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)
        self.maxpool = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)

        # 残差阶段（Conv2_x ~ Conv5_x）
        self.layer1 = self._make_layer(block, 64,  layers[0], stride=1)  # 56×56
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)  # 28×28
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)  # 14×14
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)  # 7×7

        # 分类阶段
        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))  # 输出 1×1 特征
        self.fc = nn.Linear(512 * block.expansion, num_classes)

        # 参数初始化
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out',␣
↪nonlinearity='relu')
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)

    # 构建每个 stage（多个残差块堆叠）
    def _make_layer(self, block, mid_channels, blocks, stride=1):
        downsample = None
        # 若需要改变尺寸或通道数，用 1×1 卷积调整捷径
        if stride != 1 or self.in_channels != mid_channels * block.expansion:
```

```
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, mid_channels * block.expansion,
                          kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(mid_channels * block.expansion),
            )

        layers = []
        # 第一个 block 可能需要下采样
        layers.append(block(self.in_channels, mid_channels, stride, downsample))
        self.in_channels = mid_channels * block.expansion
        # 其余 block 尺寸保持不变
        for _ in range(1, blocks):
            layers.append(block(self.in_channels, mid_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        # 输入阶段
        x = self.conv1(x)
        x = self.bn1(x)
        x = self.relu(x)
        x = self.maxpool(x)

        # 残差阶段
        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        # 分类阶段
        x = self.avgpool(x)
        x = torch.flatten(x, 1)  # 展平
        x = self.fc(x)

        return x
```

```python
# ============================================================
# 构建 ResNet-50 实例
# ============================================================
def ResNet50(num_classes=1000):
    return ResNet(Bottleneck, [3, 4, 6, 3], num_classes)


# ============================================================
# 测试网络结构
# ============================================================
if __name__ == "__main__":
    model = ResNet50(num_classes=1000)
    x = torch.randn(1, 3, 224, 224)
    y = model(x)
    print("输出维度:", y.shape)  # torch.Size([1, 1000])
```

输出维度: torch.Size([1, 1000])

4 四、循环神经网络

4.1 RNN

https://zhuanlan.zhihu.com/p/19700766543

https://blog.csdn.net/weixin_63685622/article/details/138789686

https://www.cnblogs.com/flyup/p/18903402

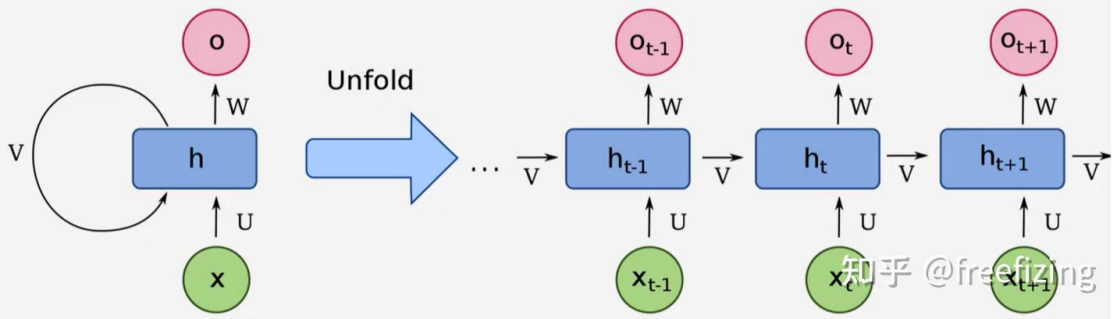
https://www.runoob.com/nlp/recurrent-neural-network.html

RNN（Recurrent Neural Network）是一类用于处理序列数据的神经网络，它通过在网络中引入循环，使得网络的隐藏状态（hidden state）不仅依赖于当前的输入，还依赖于前一时间步的隐藏状态。RNN 对具有序列特性（符合时间顺序，逻辑顺序，或者其他顺序就叫序列特性）的数据非常有效，它能挖掘数据中的时序信息以及语义信息。利用 RNN 的这种能力，深度学习模型在解决语音识别、语言模型、机器翻译以及时序分析等 NLP 领域的问题时有所突破。

在训练 RNN 时，通常使用反向传播通过时间（Backpropagation Through Time，BPTT）来计算

梯度并更新权重。BPTT 将 RNN 展开为一个时间步长的序列，并对每个时间步长进行反向传播。

朴素 RNN（Vanilla RNN）的隐藏状态更新公式为：$h_t = tanh(W_h h_{t-1} + W_x x_t + b_h)$ - 其中 W_h 和 W_x 是权重矩阵，分别用于隐藏状态和输入向量。- b_h 是偏置值 - $tanh$ 激活函数，引入非线性



梯度消失/梯度爆炸的原因

假设我们的时间序列只有三段——t_1,t_2,$t3$ 为给定值，先假设神经元没有激活函数，则 RNN 最简单的前向传播过程如下（o 是每个时刻的预测值，y 是每个时刻的标准值，用二者之差求损失）：

$$h_1 = Ux_1 + Vh_0 + b_1, o_1 = Wh_1 + b_2$$

$$h_2 = Ux_2 + Vh_1 + b_1, o_2 = Wh_2 + b_2$$

$$h_3 = Ux_3 + Vh_2 + b_1, o_3 = Wh_3 + b_2$$

假设在 t=3 时刻，损失函数 $L_3 = \frac{1}{2}(y_3 - o_3)^2$

$$\frac{\partial L_3}{\partial W} = \frac{\partial L_3}{\partial o_3}\frac{\partial o_3}{\partial W}$$

$$\frac{\partial L_3}{\partial U} = \frac{\partial L_3}{\partial o_3}\frac{\partial o_3}{\partial h_3}\frac{\partial h_3}{\partial U} + \frac{\partial L_3}{\partial o_3}\frac{\partial o_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial U} + \frac{\partial L_3}{\partial o_3}\frac{\partial o_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial U}$$

$$\frac{\partial L_3}{\partial V} = \frac{\partial L_3}{\partial o_3}\frac{\partial o_3}{\partial h_3}\frac{\partial h_3}{\partial V} + \frac{\partial L_3}{\partial o_3}\frac{\partial o_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial V} + \frac{\partial L_3}{\partial o_3}\frac{\partial o_3}{\partial h_3}\frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial V}$$

可以看出对 W 求偏导并没有长期依赖，但是对 U、V 求偏导，会随着时间序列产生长期依赖。因为 h_t 随着时间序列向前传播，而 h_t 又是 U,V 的函数

可以得出任意时刻对 U、V 求偏导的公式：

$$\frac{\partial L_t}{\partial U} = \sum_{k=0}^{t} \frac{\partial L_t}{\partial o_t}\frac{\partial o_t}{\partial h_t}\left(\prod_{j=k+1}^{t}\frac{\partial h_j}{\partial h_{j-1}}\right)\frac{\partial h_k}{\partial U}$$

加入激活函数

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

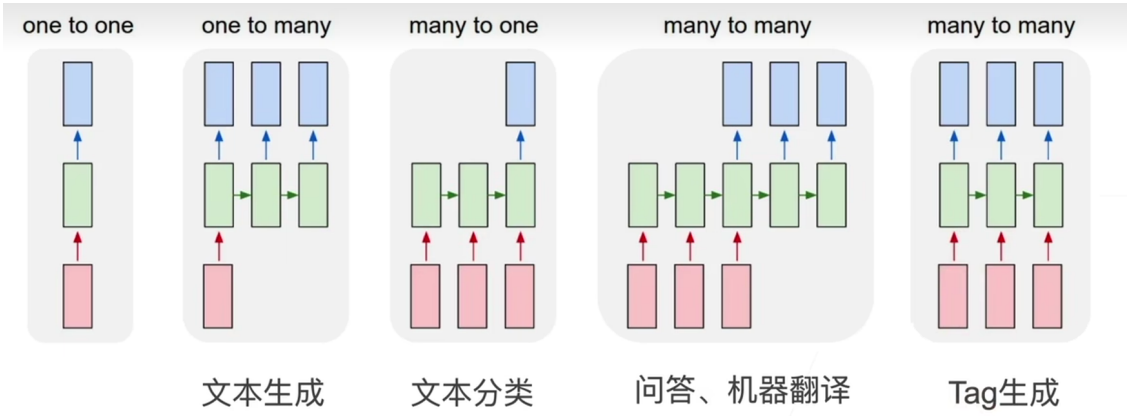
$$h_j = tanh(Ux_j + Vh_{j-1} + b_1)$$

则

$$\prod_{j=k+1}^{t} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=k+1}^{t} tanh' V$$

激活函数 tanh 的最大值小于 1，总体仍是小于 1 的，多个小于 1 的数连乘之后，那将会越来越小，导致靠近输入层的层的权重的偏导几乎为 0，也就是说几乎不更新，这就是梯度消失的根本原因。（梯度爆炸是也是同样的原因，只是如果初始权重大于 1，或者更大一些，多个大于 1 的值连乘，将会很大或溢出，导致梯度更新过大，模型无法收敛。）

RNN 的多种变体



[10]:
```
import torch
import torch.nn as nn
import torch.optim as optim

# 定义一个简单的 RNN 模型
class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, num_classes):
        super(SimpleRNN, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # RNN 层
        self.rnn = nn.RNN(input_size, hidden_size, num_layers, batch_first=True)
```

```
        # 全连接层
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        # 初始化隐藏状态 h0
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size).to(x.
↪device)

        # 前向传播
        out, _ = self.rnn(x, h0)   # out: (batch, seq_len, hidden_size)

        # 取最后一个时间步的输出进行分类
        out = self.fc(out[:, -1, :])
        return out


# 假设我们输入的是 10 个样本，每个样本是一个长度为 5 的序列，每个元素是 3 维特征
batch_size = 10
seq_length = 5
input_size = 3
hidden_size = 8
num_layers = 1
num_classes = 2

# 随机输入数据与标签
x = torch.randn(batch_size, seq_length, input_size)
y = torch.randint(0, num_classes, (batch_size,))

# 实例化模型
model = SimpleRNN(input_size, hidden_size, num_layers, num_classes)

# 定义损失函数与优化器
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

# 训练若干轮
```

```python
for epoch in range(100):
    # 前向传播
    outputs = model(x)
    loss = criterion(outputs, y)

    # 反向传播与更新
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if (epoch + 1) % 20 == 0:
        print(f'Epoch [{epoch+1}/100], Loss: {loss.item():.4f}')
```

```
Epoch [20/100], Loss: 0.4287
Epoch [40/100], Loss: 0.1068
Epoch [60/100], Loss: 0.0075
Epoch [80/100], Loss: 0.0031
Epoch [100/100], Loss: 0.0021
```

4.2 LSTM

https://zhuanlan.zhihu.com/p/123857569

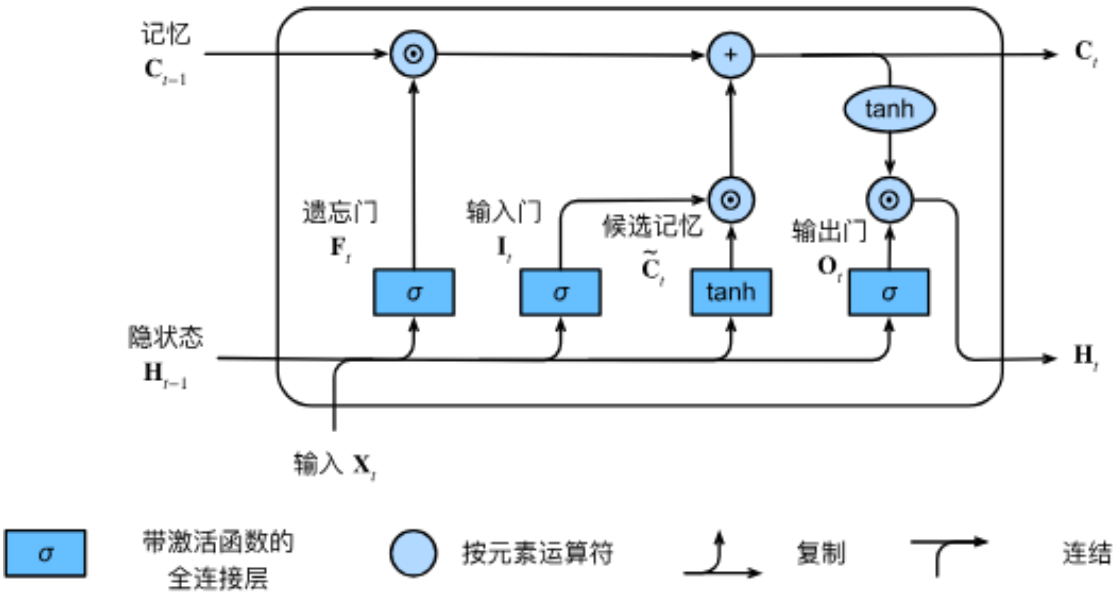
https://blog.csdn.net/mary19831/article/details/129570030

https://zh.d2l.ai/chapter_recurrent-modern/lstm.html

https://mp.weixin.qq.com/s?__biz=MjM5NzEyMzg4MA==&mid=2649509456&idx=5&sn=08259f76992575ab2

LSTM（长短时记忆网络）是一种常用于处理序列数据的深度学习模型，与传统的 RNN（循环神经网络）相比，LSTM 引入了三个门（输入门、遗忘门、输出门，如下图所示）和一个细胞状态（cell state），这些机制使得 LSTM 能够更好地处理序列中的长期依赖关系。

LSTM 从被设计之初就被用于解决一般 RNN 中普遍存在的长期依赖问题，使用 LSTM 可以有效的传递和表达长时间序列中的信息并且不会导致长时间前的有用信息被忽略（遗忘）。与此同时，LSTM 还可以解决 RNN 中的梯度消失/爆炸问题。



LSTM 的核心改进点在于引入了记忆单元（Cell State）和一系列门控机制：

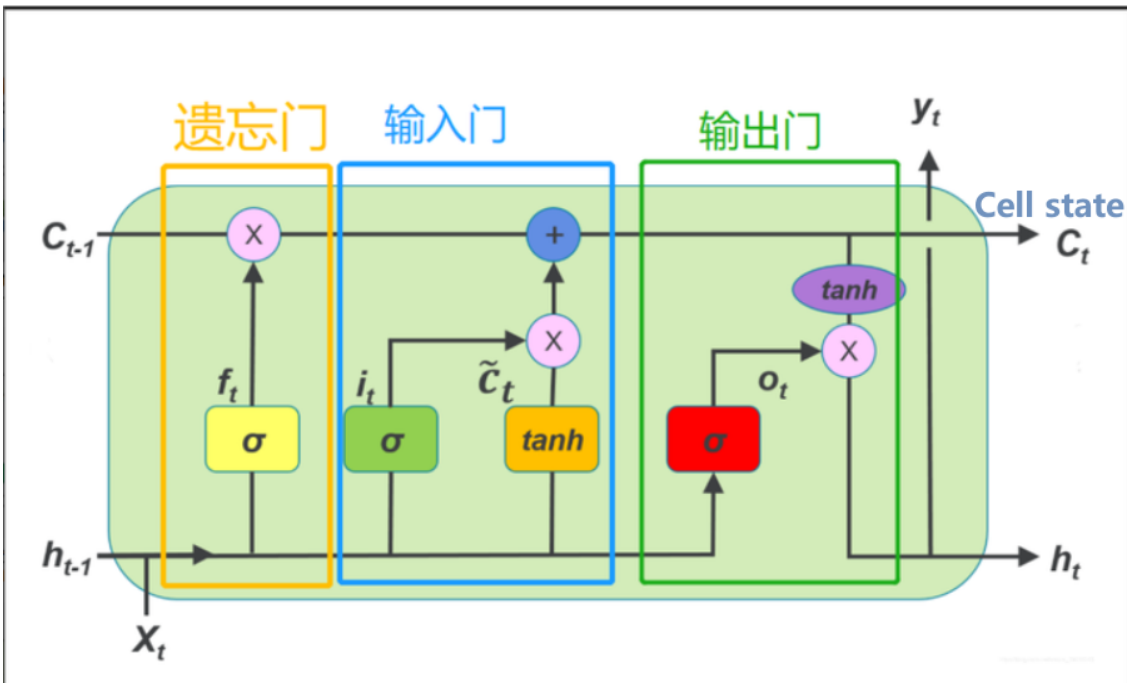
记忆单元（Cell State）：是一个贯穿整个序列的数据通道，用于存储和传递关键信息。记忆单元在 LSTM 的时间步之间保持信息的持久性，可以看作是一个长期记忆的 “容器”。这种设计使得 LSTM 能够有效 “记住” 长时间序列中的重要信息，避免了传统 RNN 在长序列中遗忘早期信息的问题。记忆单元状态会随着序列的每个时间步逐步更新，新的信息可以写入，旧的信息可以通过 “遗忘” 清除。这个过程由一系列门控机制控制，从而确保记忆单元在更新时仅存储关键信息。这种状态的传递和更新赋予了 LSTM 对长期依赖关系的捕捉能力。

门控机制：LSTM 的结构包含三个关键的门：输入门、遗忘门和输出门，这些门用于控制信息的流动和更新。

输入门：决定将当前输入的信息写入记忆单元的程度。

遗忘门：决定是否 “遗忘” 记忆单元中已存储的历史信息，从而允许模型有选择性地 “清除” 无关信息。

输出门：决定记忆单元中的信息在当前时间步的输出。



4.2.1 遗忘门

决定是否“遗忘”记忆单元中已存储的历史信息，从而允许模型有选择性地“清除”无关信息。遗忘门决定哪些信息从记忆单元中移除。每一个时间步，LSTM 会读取当前输入以及上一时间步的隐藏状态，通过 sigmoid 函数输出一个 0 到 1 的值，对应着遗忘的比例。输出值越接近 1 表示该信息越重要，应该被保留；越接近 0 则表示该信息可以被遗忘。

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

其中: - f_t 为遗忘门的输出 - W_f, b_f 为遗忘门权重、偏置 - h_{t-1} 为上一时刻的隐藏状态 - x_t 为当前时刻的输入 - $(*)$：sigmoid 激活函数，用于输出 0-1 之间的概率值，决定遗忘多少信息

如果 f_t 近似为 0，则表示遗忘过去信息

如果 f_t 近似为 1，则表示完全保留过去信息

4.2.2 输入门

决定将当前输入的信息写入记忆单元的程度。它决定了当前的输入值以及上一时间步的隐藏状态（或称短期记忆）中哪些信息需要添加到记忆单元中。

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$$

其中：- i_t 为输入门的输出（0-1 之间的值），决定当前输入信息的重要性 - $\tilde{C}_t$ 候选记忆单元状态，用 tanh 激活，使其取值为 $(-1,1)$ - 其余为参数

i_t 控制新信息的引入程度

$\tilde{C}_t$ 是新加入的候选记忆信息

4.2.3 细胞状态更新

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t$$

上一时刻的记忆单元信息 C_{t-1} 经过 f_t 处理后，决定要保留多少记忆信息新信息（候选记忆单元）$\tilde{C}_t i_t$ 处理后，决定要新加入多少信息到记忆单元中

4.2.4 输出门

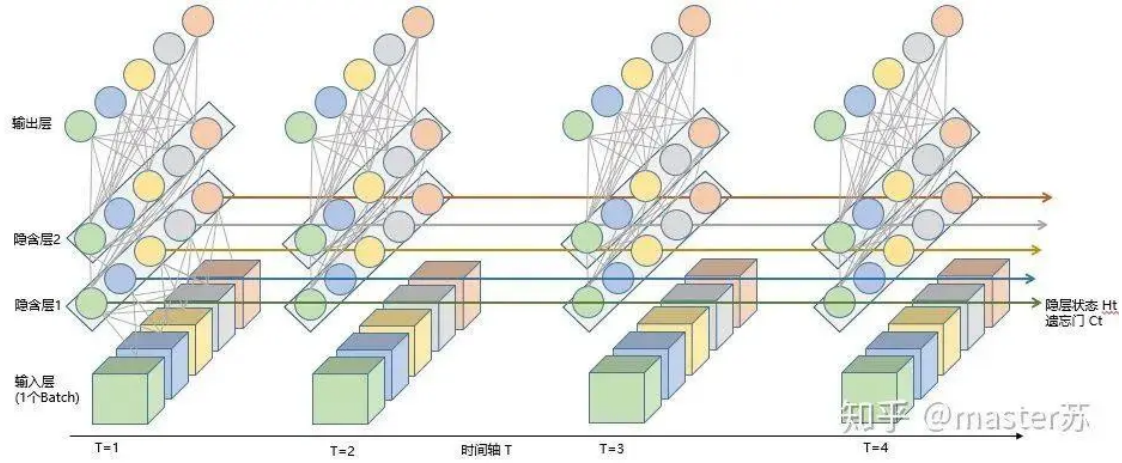
决定记忆单元中的信息在当前时间步的输出。输出门决定了当前时间步的记忆单元状态如何影响到输出的隐藏状态。即，它控制了记忆单元的内容在当前时间步的输出情况。

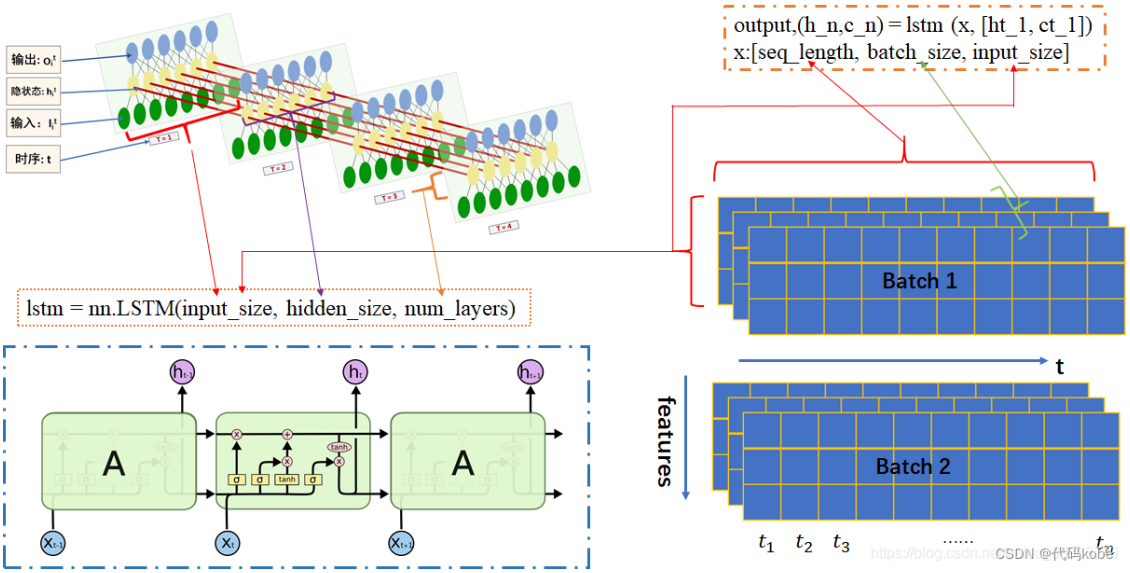
$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

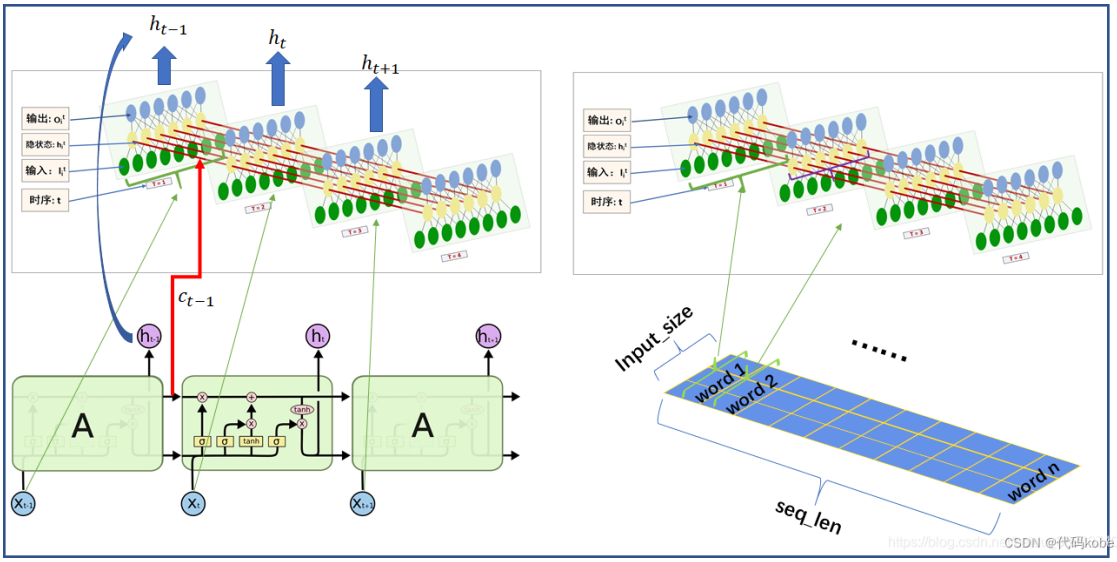
- o_t 控制 LSTM 单元的输出信息量
- h_t 为当前状态的隐状态

4.2.5 隐藏状态更新

$$h_t = o_t \cdot tanh(C_t)$$


输出层
隐含层2
隐含层1
输入层
(1个Batch)
隐层状态 Ht
遗忘门 Ct
T=1
T=2
时间轴 T
T=3
T=4
知乎 @master苏


output,(h_n,c_n) = lstm (x, [ht_1, ct_1])
x:[seq_length, batch_size, input_size]
lstm = nn.LSTM(input_size, hidden_size, num_layers)
Batch 1
Batch 2
features
t
A
A
CSDN @代码kobe

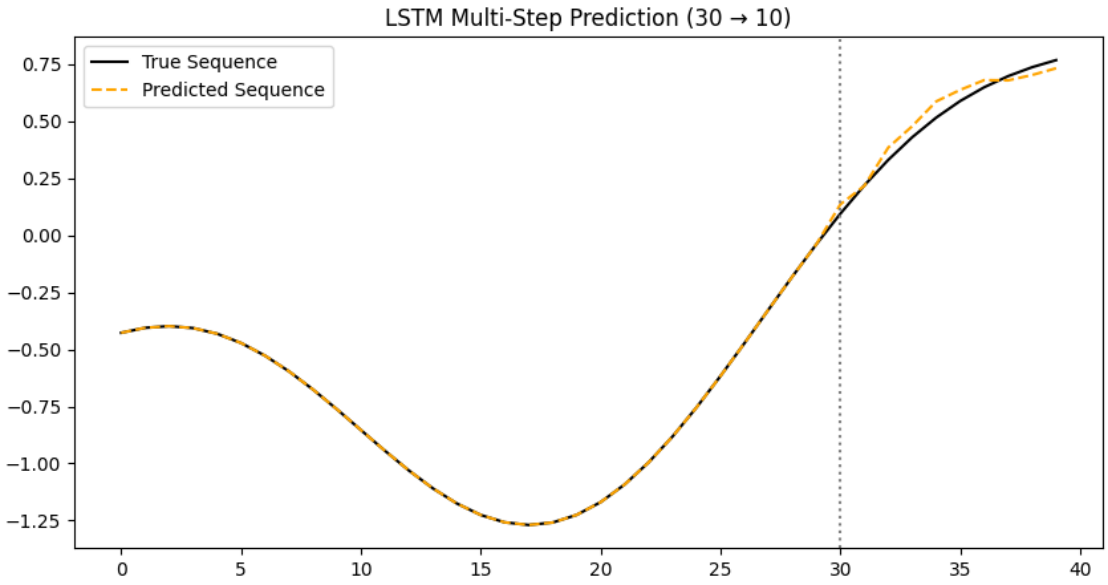

h_{t-1}
h_t
h_{t+1}
c_{t-1}
A
A
Input_size
word 1
word 2
word n
seq_len
CSDN @代码kobe

[11]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt
# 生成更复杂的时间序列
x = np.linspace(0, 120, 2500)
y = np.sin(x) + 0.5*np.sin(3*x) + 0.2*np.cos(5*x)

# 超参数
seq_len = 30   # 输入长度
pred_len = 10  # 预测长度

X, Y = [], []
for i in range(len(y) - seq_len - pred_len):
    X.append(y[i:i+seq_len])
    Y.append(y[i+seq_len:i+seq_len+pred_len])

X = np.array(X)
Y = np.array(Y)

# 转换为 tensor
X_tensor = torch.tensor(X, dtype=torch.float32).unsqueeze(-1)  # [N, seq_len, 1]
Y_tensor = torch.tensor(Y, dtype=torch.float32)  # [N, pred_len]



class MultiStepLSTM(nn.Module):
    def __init__(self, input_size=1, hidden_size=64, num_layers=2,␣
 ↪output_size=10):
        super(MultiStepLSTM, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        self.lstm = nn.LSTM(input_size, hidden_size, num_layers,␣
 ↪batch_first=True, dropout=0.2)
```

```
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)
        c0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)

        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :])  # 取最后时间步的输出
        return out

model = MultiStepLSTM(output_size=pred_len)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.005)
num_epochs = 100

for epoch in range(num_epochs):
    model.train()
    output = model(X_tensor)
    loss = criterion(output, Y_tensor)

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

    if (epoch+1) % 10 == 0:
        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.6f}")

model.eval()
with torch.no_grad():
    pred = model(X_tensor).numpy()

# 可视化第 100 段预测结果
idx = 100
true_seq = np.concatenate((X_tensor[idx,:,0].numpy(), Y[idx]))
pred_seq = np.concatenate((X_tensor[idx,:,0].numpy(), pred[idx]))
```

```
plt.figure(figsize=(10,5))
plt.plot(true_seq, label='True Sequence', color='black')
plt.plot(pred_seq, label='Predicted Sequence', color='orange', linestyle='--')
plt.axvline(x=seq_len, color='gray', linestyle=':')
plt.legend()
plt.title("LSTM Multi-Step Prediction (30 → 10)")
plt.show()
```

```
Epoch [10/100], Loss: 0.303230
Epoch [20/100], Loss: 0.125958
Epoch [30/100], Loss: 0.066328
Epoch [40/100], Loss: 0.027445
Epoch [50/100], Loss: 0.011491
Epoch [60/100], Loss: 0.007930
Epoch [70/100], Loss: 0.005459
Epoch [80/100], Loss: 0.004167
Epoch [90/100], Loss: 0.003386
Epoch [100/100], Loss: 0.002768
```



[ ]:

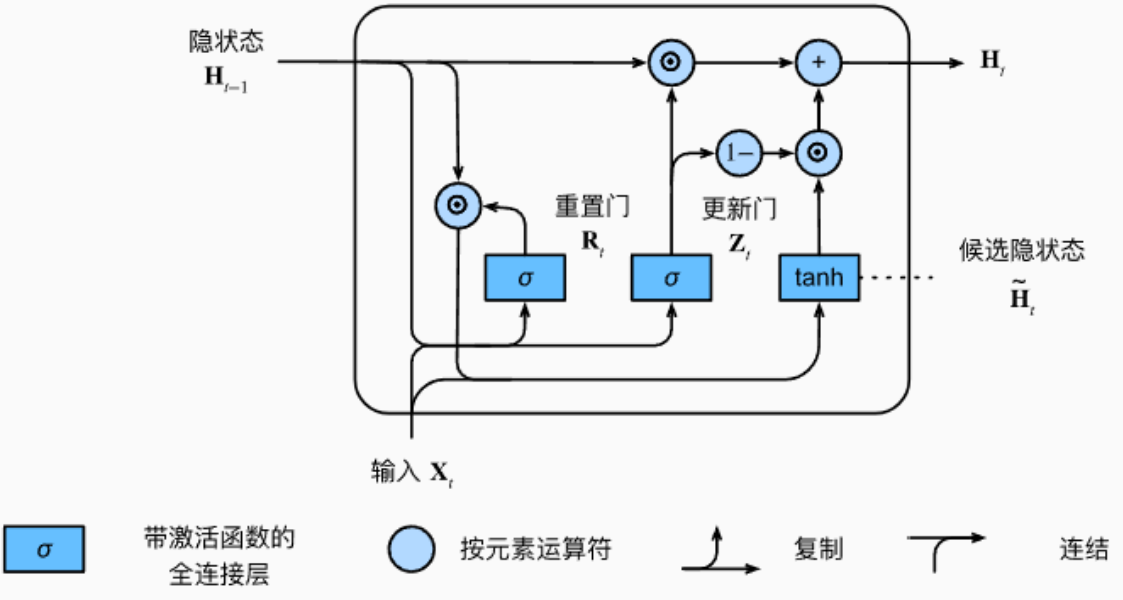
4.3 GRU

https://blog.csdn.net/weixin_43114209/article/details/143735818

https://zhuanlan.zhihu.com/p/1903398877375210503

GRU（Gated Recurrent Unit，门控循环单元）是一种循环神经网络（Recurrent Neural Network, RNN）变体，旨在处理序列数据。GRU 在 LSTM（Long Short-Term Memory，长短期记忆网络）的基础上进行了简化，引入了更少的参数量和结构复杂度。GRU 通过使用门控机制有效解决了传统 RNN 存在的梯度消失和梯度爆炸问题，尤其适合处理长时间依赖的数据。GRU 的设计目的是在保持计算效率的同时，拥有较高的性能，适用于广泛的序列处理任务。

GRU 可以被看作是 LSTM 的简化版。它保留了门控机制中的更新门和重置门，用于控制信息的流动，但省略了 LSTM 中的单独记忆单元。相比 LSTM，GRU 拥有更少的参数，因此计算效率更高，通常在一些任务上可以获得相近甚至更好的效果。



GRU 的核心组件：更新门和重置门

4.3.1 更新门

更新门的作用是决定当前时间步的隐藏状态需要保留多少前一个时间步的信息。更新门的输出值介于 0 和 1 之间，值越大表示保留的过去信息越多，值越小则意味着更多地依赖于当前输入的信息。

$$Z_t = \sigma(X_t W_{xz} + H_{t-1} W_{hz} + b_z)$$

重置门

重置门决定了前一时间步的隐藏状态在多大程度上被忽略。当重置门的输出接近 0 时，网络倾向于“忘记” 前一时间步的信息，仅依赖于当前输入；而当输出接近 1 时，前一时间步的信息将被更多地保留。

$$R_t = \sigma(X_t W_{xr} + H_{t-1} W_{hr} + b_r)$$

4.3.2 前向传播过程

在每一个时间步中，GRU 单元通过以下步骤进行信息处理：

1. 计算更新门和重置门：

根据上面的公式分别计算出更新门 Z_t 和重置门 R_t

这些门的输出将控制隐藏状态的更新和历史信息的保留程度。

2. 计算候选隐藏状态：

GRU 使用重置门控制对先前隐藏状态的依赖程度，计算出候选隐藏状态。

候选隐藏状态的计算公式为：

$$\tilde{H}_t = tanh(X_t \cdot W_{xh} + (R_t \cdot H_{t-1}) \cdot W_{hh} + b_h)$$

其中，$(R_t \cdot H_{t-1})$ 表示将前一个时间步的隐藏状态 H_{t-1} 与重置门 R_t 结合以控制其影响程度，W 和 b 是参数矩阵和偏置。

3. 更新隐藏状态：

最后一步是利用更新门 Z_t 来计算当前的隐藏状态 H_t：

$$H_t = Z_t \cdot H_{t-1} + (1 - Z_t) \cdot \tilde{H}_t$$

其中，$Z_t \cdot H_{t-1}$ 表示保留的历史信息，$(1 - Z_t) \cdot \tilde{H}_t$ 表示新的信息。这一公式决定了当前隐藏状态既保留了上一时间步的部分信息，也引入了基于当前输入的新信息。

项目	**LSTM（Long Short-Term Memory）**	**GRU（Gated Recurrent Unit）**
结构复杂度	较复杂（3 个门 + 1 个单元状态）	较简单（2 个门）
主要门控机制	输入门（Input Gate）遗忘门（Forget Gate）输出门（Output Gate）	更新门（Update Gate）重置门（Reset Gate）
隐藏状态	分为隐藏状态 (h_t) 与细胞状态 (c_t)	仅隐藏状态 (h_t)

项目	LSTM（Long Short-Term Memory）	GRU（Gated Recurrent Unit）
是否有独立记忆单元（cell state）	有	无
参数数量	较多（约为 GRU 的 1.5 倍）	较少，训练速度更快
计算效率	较慢，占用内存较多	较快，适合小模型或实时任务
长期依赖捕获能力	强，适合长序列建模	略弱，但足够应对中短序列
梯度消失问题	有效缓解	也能缓解，但稍弱于 LSTM
应用场景	复杂序列任务，如语音识别、机器翻译、长文本生成等	轻量级任务，如语音指令识别、时间序列预测等
优点	能有效保留长时依赖信息，性能稳定	结构简洁、参数更少、收敛更快
缺点	训练慢，模型大，调参难度高	表达能力略逊于 LSTM（尤其是长依赖任务）

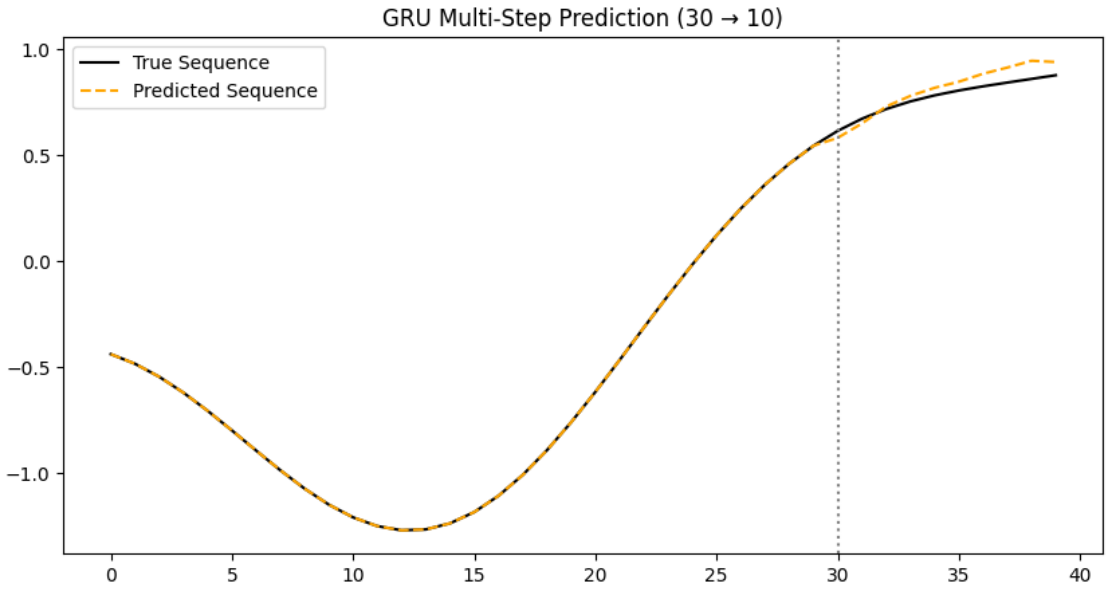
[12]:
```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import matplotlib.pyplot as plt

# 生成复杂波形数据
x = np.linspace(0, 100, 2000)
y = np.sin(x) + 0.5*np.sin(3*x) + 0.2*np.cos(5*x)

# 超参数
seq_len = 30   # 输入长度
pred_len = 10  # 预测长度

X, Y = [], []
for i in range(len(y) - seq_len - pred_len):
    X.append(y[i:i+seq_len])
    Y.append(y[i+seq_len:i+seq_len+pred_len])

X = np.array(X)
```

```
Y = np.array(Y)

# 转为 Tensor
X_tensor = torch.tensor(X, dtype=torch.float32).unsqueeze(-1)  # [N, seq_len, 1]
Y_tensor = torch.tensor(Y, dtype=torch.float32)                # [N, pred_len]

# 定义 GRU 模型
class SimpleGRU(nn.Module):
    def __init__(self, input_size=1, hidden_size=64, num_layers=2,␣
 ↪output_size=10):
        super(SimpleGRU, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        # GRU 层
        self.gru = nn.GRU(input_size, hidden_size, num_layers,␣
 ↪batch_first=True, dropout=0.2)
        # 输出层
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # 初始化隐藏状态
        h0 = torch.zeros(self.num_layers, x.size(0), self.hidden_size)

        out, _ = self.gru(x, h0)
        out = self.fc(out[:, -1, :])  # 取最后时间步输出
        return out

# 初始化模型、损失函数与优化器
model = SimpleGRU(output_size=pred_len)
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.005)

# 训练
num_epochs = 100
for epoch in range(num_epochs):
```

```
    model.train()
    output = model(X_tensor)
    loss = criterion(output, Y_tensor)

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

    if (epoch+1) % 10 == 0:
        print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {loss.item():.6f}")

# 预测
model.eval()
with torch.no_grad():
    pred = model(X_tensor).numpy()

# 可视化某段预测
idx = 100
true_seq = np.concatenate((X_tensor[idx,:,0].numpy(), Y[idx]))
pred_seq = np.concatenate((X_tensor[idx,:,0].numpy(), pred[idx]))

plt.figure(figsize=(10,5))
plt.plot(true_seq, label='True Sequence', color='black')
plt.plot(pred_seq, label='Predicted Sequence', color='orange', linestyle='--')
plt.axvline(x=seq_len, color='gray', linestyle=':')
plt.legend()
plt.title("GRU Multi-Step Prediction (30 → 10)")
plt.show()
```

```
Epoch [10/100], Loss: 0.264187
Epoch [20/100], Loss: 0.143880
Epoch [30/100], Loss: 0.089620
Epoch [40/100], Loss: 0.047578
Epoch [50/100], Loss: 0.018094
Epoch [60/100], Loss: 0.010896
Epoch [70/100], Loss: 0.007737
Epoch [80/100], Loss: 0.005138
```

```
Epoch [90/100], Loss: 0.003838
Epoch [100/100], Loss: 0.003132
```



5 五、自注意力机制与 Transformer

https://zhuanlan.zhihu.com/p/338817680

https://zhuanlan.zhihu.com/p/607423406

注意力机制原理、自注意力计算过程

Transformer 编码器结构

Transformer 实现

BERT