北京邮电大学

Beijing University of Posts and Telecommunications

数据科学

题目：期中实验

学院：信息与通信工程学院

班级：2019211127

姓名：王兴睿

学号：2018211756

目录

任务一……3
任务二……3
任务三……4
任务四……5
　　更改学习率的影响……5
　　dropout……6
　　batch_size……7
任务五……9
任务六……10
任务八……11

- 任务一：安装配置 Pytorch 环境，检测 Pytorch 安装情况。（10 分）

```
In [1]:  import torch
         import torchvision

In [2]:  torch.__version__

Out[2]: '1.7.0'

In [3]:  torchvision.__version__

Out[3]: '0.8.1'
```

- 任务二：使用包含三层以上个卷积层的神经网络对 CIFAR-10 数据集分类。对生成网络结构进行截图（如例 1 所示），并对训练过程的精度增长和 loss 收敛情况进行截图（如例 2 所示） 。（15 分）

本次实验尝试了课件上给出的 ConNet 和 AlexNet 两种网络结构，其网络结构如下，经测试发现 ConNet 训练的效果要更为准确一点，故后续实验均在 ConNet 网络上进行。

```
In [13]:  print(model)

ConvNet(
  (conv1): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv3): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc1): Sequential(
    (0): Linear(in_features=1024, out_features=32, bias=True)
    (1): ReLU()
  )
  (fc2): Linear(in_features=32, out_features=10, bias=True)
)
```

图 1 ConvNet 生成网络结构

```
AlexNet(
  (conv1): Sequential(
    (0): Conv2d(3, 6, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv2): Sequential(
    (0): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv3): Sequential(
    (0): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv4): Sequential(
    (0): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (conv5): Sequential(
    (0): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
```

```
(dense): Sequential(
  (0): Linear(in_features=128, out_features=120, bias=True)
  (1): ReLU()
  (2): Linear(in_features=120, out_features=84, bias=True)
  (3): ReLU()
  (4): Linear(in_features=84, out_features=10, bias=True)
)
```

图 2 AlexNet 生成网络结构

```
Train Epoch: 3 [9750/50000 (20%)]	Loss: 1.347469
Train Epoch: 3 [19750/50000 (40%)]	Loss: 1.292282
Train Epoch: 3 [29750/50000 (60%)]	Loss: 1.310190
Train Epoch: 3 [39750/50000 (80%)]	Loss: 1.322784
Train Epoch: 3 [49750/50000 (100%)]	Loss: 1.333996

Test set: Average loss: 1.2428, Accuracy: 5428/10000 (54%)

Train Epoch: 4 [9750/50000 (20%)]	Loss: 1.212287
Train Epoch: 4 [19750/50000 (40%)]	Loss: 1.264023
Train Epoch: 4 [29750/50000 (60%)]	Loss: 1.142816
Train Epoch: 4 [39750/50000 (80%)]	Loss: 1.088728
Train Epoch: 4 [49750/50000 (100%)]	Loss: 1.035411

Test set: Average loss: 1.1725, Accuracy: 5791/10000 (58%)

Train Epoch: 5 [9750/50000 (20%)]	Loss: 1.307816
Train Epoch: 5 [19750/50000 (40%)]	Loss: 1.150361
Train Epoch: 5 [29750/50000 (60%)]	Loss: 1.151052
Train Epoch: 5 [39750/50000 (80%)]	Loss: 1.105514
Train Epoch: 5 [49750/50000 (100%)]	Loss: 1.081377

Test set: Average loss: 1.0931, Accuracy: 6041/10000 (60%)
```

图 3 loss 收敛情况和测试集精度

- 任务三：对 CIFAR-10 数据进行解析和可视化展示。输出 CIFAR-10 数据集训练集、测试集大小；输出数据集包含的所有类别名称及与 label 对应情况；输出数据集中一张图片的数组 size，并将数据集测试集三张图片进行可视化展示。（15 分）

In [14]:
```
print(len(train_loader.dataset))
print(len(test_loader.dataset))
```

```
50000
10000
```

图 4 训练集、测试集大小

In [5]:
```
print(train_data.class_to_idx)
```

```
{'airplane': 0, 'automobile': 1, 'bird': 2, 'cat': 3, 'deer': 4, 'dog': 5, 'frog': 6, 'horse': 7, 'ship': 8, 'truck': 9}
```

图 5 对应情况

```
print(train_data.data[0].shape)
```

```
(32, 32, 3)
```

图 6 一张图片大小

```
print("label",train_data.classes[train_data.targets[3]])
print("data",train_data.data[3])
```

```
label deer
data [[[ 28  25  10]
  [ 37  34  19]
  [ 38  35  20]
  ...
  [ 76  67  39]
  [ 81  72  43]
  [ 85  76  47]]

 [[ 33  28  13]
  [ 34  30  14]
  [ 32  27  12]
  ...
  [ 95  82  55]
  [ 96  82  56]
  [ 85  72  45]]

 [[ 39  32  15]
  [ 40  33  17]
  [ 57  50  33]
  ...
  [ 93  76  52]
  [107  89  66]
  [ 95  77  54]]
```

图 7 一张图片的数组

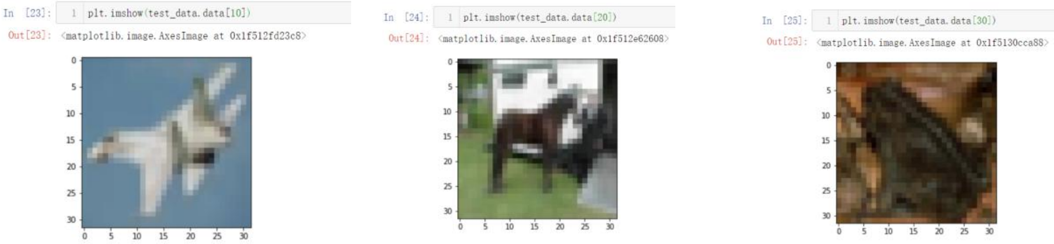


图 8 测试集可视化

- 任务四：修改网络结构（调整网络深度，使用不同的激活函数，调整神经元数量）或更改训练参数（学习率，batch_size），分析不同网络参数对于检测结果影响（至少分析两个变量，应有改动的关键代码段截图、前后效果对比与文字解析）（20 分）

➢ 更改学习率的影响

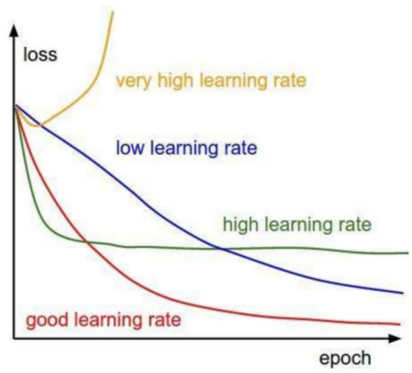


图 9 学习率的影响

学习速率代表了神经网络中随时间推移，信息累积的速度。学习率是最影响性能的超参数之一。首先使用较大的学习率 0.01 进行训练：

```
In [8]:
model = ConvNet().to(DEVICE)   # 将网络放到GPU设备上
#optimizer = optim.Adam(model.parameters(),lr=0.001,betas=(0.9,0.999),eps=1e-8,weight_decay=0,amsgrad=False)   # 优化器我们选择Adam
optimizer = optim.Adam(model.parameters(),lr=0.1 betas=(0.9,0.999),eps=1e-8,weight_decay=0,amsgrad=False)   # 优化器我们选择Adam
loss_func = torch.nn.CrossEntropyLoss()
```

```
Train Epoch: 2 [29750/50000 (60%)]	Loss: 2.298251
Train Epoch: 2 [39750/50000 (80%)]	Loss: 2.307706
Train Epoch: 2 [49750/50000 (100%)]	Loss: 2.313305

Test set: Average loss: 2.3075, Accuracy: 1000/10000 (10%)

Train Epoch: 3 [9750/50000 (20%)]	Loss: 2.312392
Train Epoch: 3 [19750/50000 (40%)]	Loss: 2.302892
Train Epoch: 3 [29750/50000 (60%)]	Loss: 2.313453
Train Epoch: 3 [39750/50000 (80%)]	Loss: 2.304964
Train Epoch: 3 [49750/50000 (100%)]	Loss: 2.298656

Test set: Average loss: 2.3051, Accuracy: 1000/10000 (10%)

Train Epoch: 4 [9750/50000 (20%)]	Loss: 2.302849
Train Epoch: 4 [19750/50000 (40%)]	Loss: 2.311271
Train Epoch: 4 [29750/50000 (60%)]	Loss: 2.291427
Train Epoch: 4 [39750/50000 (80%)]	Loss: 2.318124
Train Epoch: 4 [49750/50000 (100%)]	Loss: 2.304232

Test set: Average loss: 2.3063, Accuracy: 1000/10000 (10%)

Train Epoch: 5 [9750/50000 (20%)]	Loss: 2.301415
Train Epoch: 5 [19750/50000 (40%)]	Loss: 2.299007
Train Epoch: 5 [29750/50000 (60%)]	Loss: 2.306200
Train Epoch: 5 [39750/50000 (80%)]	Loss: 2.309871
Train Epoch: 5 [49750/50000 (100%)]	Loss: 2.293775

Test set: Average loss: 2.3056, Accuracy: 1000/10000 (10%)
```

当选用较大的学习率时，最后在测试集上的准确率只有 10%。选择较高的学习率，它可能在你的损失函数上带来不理想的后果，因此几乎从来不能到达全局最小值，因为你很可能跳过它。

选用较小的学习率 0.0005 进行训练：

In [13]:

```
model = ConvNet().to(DEVICE)  # 将网络放到GPU设备上
#optimizer = optim.Adam(model.parameters(), lr=0.001, betas=(0.9, 0.999), eps=1e-8, weight_decay=0, amsgrad=False)  # 优化器我们选择Adam
#optimizer = optim.Adam(model.parameters(), lr=0.1, betas=(0.9, 0.999), eps=1e-8, weight_decay=0, amsgrad=False)  # 优化器我们选择Adam
optimizer = optim.Adam(model.parameters(), lr=0.0005, betas=(0.9, 0.999), eps=1e-8, weight_decay=0, amsgrad=False)  # 优化器我们选择Adam
loss_func = torch.nn.CrossEntropyLoss()
```

```
Test set: Average loss: 1.4802, Accuracy: 4556/10000 (46%)

Train Epoch: 3 [9750/50000 (20%)]	Loss: 1.359419
Train Epoch: 3 [19750/50000 (40%)]	Loss: 1.443448
Train Epoch: 3 [29750/50000 (60%)]	Loss: 1.348773
Train Epoch: 3 [39750/50000 (80%)]	Loss: 1.451087
Train Epoch: 3 [49750/50000 (100%)]	Loss: 1.387597

Test set: Average loss: 1.3910, Accuracy: 4965/10000 (50%)

Train Epoch: 4 [9750/50000 (20%)]	Loss: 1.309474
Train Epoch: 4 [19750/50000 (40%)]	Loss: 1.378480
Train Epoch: 4 [29750/50000 (60%)]	Loss: 1.313047
Train Epoch: 4 [39750/50000 (80%)]	Loss: 1.365146
Train Epoch: 4 [49750/50000 (100%)]	Loss: 1.319191

Test set: Average loss: 1.3226, Accuracy: 5234/10000 (52%)

Train Epoch: 5 [9750/50000 (20%)]	Loss: 1.242631
Train Epoch: 5 [19750/50000 (40%)]	Loss: 1.428601
Train Epoch: 5 [29750/50000 (60%)]	Loss: 1.227036
Train Epoch: 5 [39750/50000 (80%)]	Loss: 1.247752
Train Epoch: 5 [49750/50000 (100%)]	Loss: 1.419856

Test set: Average loss: 1.2998, Accuracy: 5344/10000 (53%)
```

当选用较小的学习率时，最后再测试集上的准确率为 53%，低于学习率为 0.001 时的 60%。如果学习率设置太小，网络收敛非常缓慢，会增大找到最优值的时间，也就是说从山坡上像蜗牛一样慢慢地爬下去。虽然设置非常小的学习率是可以到达，但是这很可能会进入局部极值点就收敛，没有真正找到的最优解，换句话说就是它步长太小，跨不出这个坑。

- dropout

在每次训练的时候，让某些的特征检测器停过工作，即让神经元以一定的概率不被激活，这样可以防止过拟合，提高泛化能力。

```python
        )
        self.conv3 = torch.nn.Sequential(
            torch.nn.Conv2d(32, 64, 3, padding=1),
            torch.nn.ReLU(),
            torch.nn.MaxPool2d(2, 2)
        )
        self.fc1 = torch.nn.Sequential(
            torch.nn.Linear(64*4*4, 32),
            torch.nn.ReLU(),
            torch.nn.Dropout()
        )
        self.fc2 = torch.nn.Linear(32, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = self.conv2(x)
        x = self.conv3(x)
        x = x.view(-1, 64*4*4)
        x = self.fc1(x)
        x = self.fc2(x)
        out = F.log_softmax(x, dim=1)
        return out
```

```
Test set: Average loss: 1.5480, Accuracy: 4438/10000 (44%)

Train Epoch: 3 [9750/50000 (20%)]	Loss: 1.787992
Train Epoch: 3 [19750/50000 (40%)]	Loss: 1.616673
Train Epoch: 3 [29750/50000 (60%)]	Loss: 1.704777
Train Epoch: 3 [39750/50000 (80%)]	Loss: 1.765565
Train Epoch: 3 [49750/50000 (100%)]	Loss: 1.640251

Test set: Average loss: 1.4498, Accuracy: 4730/10000 (47%)

Train Epoch: 4 [9750/50000 (20%)]	Loss: 1.619659
Train Epoch: 4 [19750/50000 (40%)]	Loss: 1.580410
Train Epoch: 4 [29750/50000 (60%)]	Loss: 1.676825
Train Epoch: 4 [39750/50000 (80%)]	Loss: 1.647552
Train Epoch: 4 [49750/50000 (100%)]	Loss: 1.564072

Test set: Average loss: 1.4057, Accuracy: 5052/10000 (51%)

Train Epoch: 5 [9750/50000 (20%)]	Loss: 1.626527
Train Epoch: 5 [19750/50000 (40%)]	Loss: 1.575165
Train Epoch: 5 [29750/50000 (60%)]	Loss: 1.633077
Train Epoch: 5 [39750/50000 (80%)]	Loss: 1.575032
Train Epoch: 5 [49750/50000 (100%)]	Loss: 1.566144

Test set: Average loss: 1.3582, Accuracy: 5092/10000 (51%)
```

添加了 dropout 层后，在训练集上的准确率与原来相差不大，但是因为引入 dropout 之后相当于每次只是训练的原先网络的一个子网络，为了达到同样的精度需要的训练次数会增多。dropout 的缺点就在于训练时间是没有 dropout 网络的 2-3 倍。

- batch_size

  将 batch_size 改为 50：

```python
In [3]:
# 定义参数
BATCH_SIZE = 50
EPOCHS = 5  # 总共训练批次
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")  # 让torch判断是否使用GPU，建议使用GPU环境，因为会快很多
```

```
Train Epoch: 2 [39950/50000 (80%)]	Loss: 0.978940
Train Epoch: 2 [49950/50000 (100%)]	Loss: 1.046316

Test set: Average loss: 1.1353, Accuracy: 5906/10000 (59%)

Train Epoch: 3 [9950/50000 (20%)]	Loss: 1.286583
Train Epoch: 3 [19950/50000 (40%)]	Loss: 0.767761
Train Epoch: 3 [29950/50000 (60%)]	Loss: 1.281869
Train Epoch: 3 [39950/50000 (80%)]	Loss: 1.195579
Train Epoch: 3 [49950/50000 (100%)]	Loss: 1.280758

Test set: Average loss: 0.9999, Accuracy: 6552/10000 (66%)

Train Epoch: 4 [9950/50000 (20%)]	Loss: 0.723947
Train Epoch: 4 [19950/50000 (40%)]	Loss: 1.131622
Train Epoch: 4 [29950/50000 (60%)]	Loss: 0.988611
Train Epoch: 4 [39950/50000 (80%)]	Loss: 0.913112
Train Epoch: 4 [49950/50000 (100%)]	Loss: 0.705873

Test set: Average loss: 0.9306, Accuracy: 6779/10000 (68%)

Train Epoch: 5 [9950/50000 (20%)]	Loss: 0.795945
Train Epoch: 5 [19950/50000 (40%)]	Loss: 0.875804
Train Epoch: 5 [29950/50000 (60%)]	Loss: 0.712615
Train Epoch: 5 [39950/50000 (80%)]	Loss: 0.872339
Train Epoch: 5 [49950/50000 (100%)]	Loss: 0.755342

Test set: Average loss: 0.8724, Accuracy: 7040/10000 (70%)
```

将 batch_size 改为 50 之后，经过五次 epoch 之后，在训练集上的损失减少，在测试集上的准确率相较之前提升 10%，达到了 70%，效果较原来有明显提升。

```
In [13]:
# 定义参数
BATCH_SIZE = 10
EPOCHS = 5  # 总共训练批次
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
Test set: Average loss: 0.9973, Accuracy: 6494/10000 (65%)

Train Epoch: 3 [9990/50000 (20%)]	Loss: 1.446447
Train Epoch: 3 [19990/50000 (40%)]	Loss: 0.765520
Train Epoch: 3 [29990/50000 (60%)]	Loss: 0.524015
Train Epoch: 3 [39990/50000 (80%)]	Loss: 0.593756
Train Epoch: 3 [49990/50000 (100%)]	Loss: 0.572244

Test set: Average loss: 0.9025, Accuracy: 6866/10000 (69%)

Train Epoch: 4 [9990/50000 (20%)]	Loss: 0.858200
Train Epoch: 4 [19990/50000 (40%)]	Loss: 1.367505
Train Epoch: 4 [29990/50000 (60%)]	Loss: 0.467243
Train Epoch: 4 [39990/50000 (80%)]	Loss: 0.690809
Train Epoch: 4 [49990/50000 (100%)]	Loss: 0.510455

Test set: Average loss: 0.8808, Accuracy: 6952/10000 (70%)

Train Epoch: 5 [9990/50000 (20%)]	Loss: 0.227624
Train Epoch: 5 [19990/50000 (40%)]	Loss: 0.978279
Train Epoch: 5 [29990/50000 (60%)]	Loss: 0.601727
Train Epoch: 5 [39990/50000 (80%)]	Loss: 0.676288
Train Epoch: 5 [49990/50000 (100%)]	Loss: 1.501132

Test set: Average loss: 0.8670, Accuracy: 7063/10000 (71%)
```

batch_size 为 10 时，性能与 batch_size 为 50 时相差不大，在测试集上的准确率为 71%。

```
In [3]:
# 定义参数
BATCH_SIZE = 500
EPOCHS = 5  # 总共训练批次
DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
Train Epoch: 2 [39500/50000 (79%)]	Loss: 1.398208
Train Epoch: 2 [49500/50000 (99%)]	Loss: 1.413026

Test set: Average loss: 1.4458, Accuracy: 4748/10000 (47%)

Train Epoch: 3 [9500/50000 (19%)]	Loss: 1.395712
Train Epoch: 3 [19500/50000 (39%)]	Loss: 1.369780
Train Epoch: 3 [29500/50000 (59%)]	Loss: 1.505176
Train Epoch: 3 [39500/50000 (79%)]	Loss: 1.482478
Train Epoch: 3 [49500/50000 (99%)]	Loss: 1.344123

Test set: Average loss: 1.3700, Accuracy: 5057/10000 (51%)

Train Epoch: 4 [9500/50000 (19%)]	Loss: 1.281765
Train Epoch: 4 [19500/50000 (39%)]	Loss: 1.403997
Train Epoch: 4 [29500/50000 (59%)]	Loss: 1.356873
Train Epoch: 4 [39500/50000 (79%)]	Loss: 1.368875
Train Epoch: 4 [49500/50000 (99%)]	Loss: 1.289792

Test set: Average loss: 1.3079, Accuracy: 5320/10000 (53%)

Train Epoch: 5 [9500/50000 (19%)]	Loss: 1.342459
Train Epoch: 5 [19500/50000 (39%)]	Loss: 1.300520
Train Epoch: 5 [29500/50000 (59%)]	Loss: 1.271743
Train Epoch: 5 [39500/50000 (79%)]	Loss: 1.273778
Train Epoch: 5 [49500/50000 (99%)]	Loss: 1.254973

Test set: Average loss: 1.2359, Accuracy: 5599/10000 (56%)
```

当调大 batch_size 为 500 时，此时在测试集上的准确率与 batch_size 为 250 时的训练效果相差不大，均为 60%左右。batch_size 会影响目标函数的收敛速度，经测试发现在上面的结果显示，batch_size 为 50 等较小的数值时效果反而很好，经查询，如果将 batch_size 调整的较小，其每次迭代下降的方向就不是最准确的，loss 小范围震荡下降反而会跳出局部最优解，从而寻找 loss 更低的区域。

- **任务五：使用 tensorboard 插件对训练过程中的 loss 和精度进行观察，对 tensorboard 中 loss 曲线和 accuracy 曲线进行截图记录（10 分）**

```
(base) C:\Users\DELL>tensorboard --logdir=C:\Users\DELL\Desktop\数据科学
Serving TensorBoard on localhost; to expose to the network, use a proxy or pass --bind_all
TensorBoard 2.4.1 at http://localhost:6006/ (Press CTRL+C to quit)
```

In [39]:

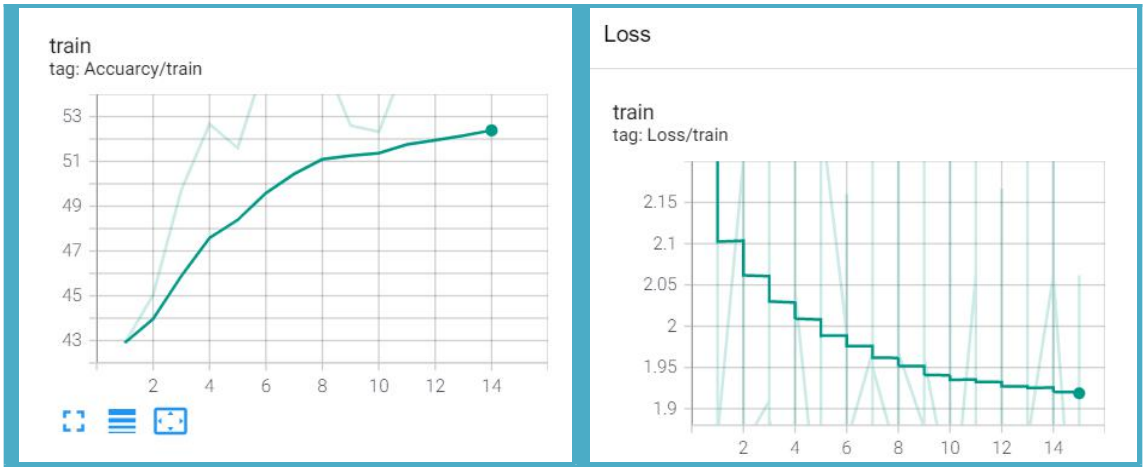
```python
from torch.utils.tensorboard import SummaryWriter
log_writer = SummaryWriter()
```

In [59]:

```python
# 训练函数
def train(model, device, train_loader, optimizer, epoch,loss_func):
    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 = loss_func(output, target)
        log_writer.add_scalar('Loss/train', float(loss), epoch)
        loss.backward()
        optimizer.step()
        if (batch_idx + 1) % 1000 == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                    100. * batch_idx / len(train_loader), loss.item()))
```

In [60]:
```python
# 测试函数
def test(model, device, test_loader):
    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 += F.nll_loss(output, target, reduction='sum').item()
            pred = output.max(1, keepdim=True)[1]
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset)
    Accuracy = 100. * correct / len(test_loader.dataset)
    log_writer.add_scalar('Accuarcy/train', float(Accuracy), epoch)
    print('\nTest set: Average loss: {:.4f}, Accuracy: {}/{} ({:.0f}%)\n'.format(
        test_loss, correct, len(test_loader.dataset),
        100. * correct / len(test_loader.dataset)))
```



- 任务六：使用训练模型对于测试集中第 i 到 i+10 张图片进行预测，输出预测结果与预测概率 softmax（i=学号最后两位*10）（10 分）

In [68]:
```python
classes = ('plane', 'car', 'bird', 'cat','deer', 'dog', 'frog', 'horse', 'ship', 'truck')
for i in range(560,571):
    trans = transforms.Compose(
        [transforms.ToTensor(),
         transforms.Normalize(mean=(0.5, 0.5, 0.5),
                              std=(0.5, 0.5, 0.5))])
    img = test_data.data[i]
    img = trans(img)
    img = img.to(DEVICE)
    img = img.unsqueeze(0)
    #image = test_data.data[i]
    #trans = transforms.ToTensor()
    #im_data = trans(image)
    #im_data = im_data.resize(1, 3, 32, 32)
    output = model(img)
    #print(output)
    print(torch.max(output))
    #print(torch.max(output.data))
    #prob = F.softmax(output,dim=1) #prob是10个分类的概率
    #print(prob)
    value, predicted = torch.max(output.data, 1)
    #print(predicted.item())
    #print(value)
    pred_class = classes[predicted.item()]
    plt.subplot(2,6,i-559)
    plt.imshow(test_data.data[i])
    plt.title(pred_class)
    print(pred_class)
```

预测结果与预测概率：

```
tensor(1., grad_fn=<MaxBackward1>)
plane
tensor(1.0000, grad_fn=<MaxBackward1>)
horse
tensor(1.0000, grad_fn=<MaxBackward1>)
frog
tensor(0.9887, grad_fn=<MaxBackward1>)
deer
tensor(1., grad_fn=<MaxBackward1>)
deer
tensor(0.9971, grad_fn=<MaxBackward1>)
cat
tensor(0.9995, grad_fn=<MaxBackward1>)
truck
tensor(0.9983, grad_fn=<MaxBackward1>)
cat
tensor(1.0000, grad_fn=<MaxBackward1>)
frog
tensor(0.9422, grad_fn=<MaxBackward1>)
frog
tensor(0.9465, grad_fn=<MaxBackward1>)
bird
```

本人的学号为 2018211756，故预测 560-570 号图片。将预测结果和实际图片相对比可得，11 张图片中共预测正确了 6 张图片，其正确率约为 55%。

- **任务八：尝试使用 KNN 等机器学习算法进行分类，并将其结果与卷积神经网络结果进行对比，分析结果差异（选做）**

```python
# K近邻算法
class KNearestNeighbor:
    def __init__(self):
        pass

    def train(self,X,y):
        self.Xtr = X
        self.ytr = y

    def predict(self,X,k):
        num = X.shape[0]
        Ypred = np.zeros(num)
        for i in range(num):
            distance = np.sum((self.Xtr - X[i,:])**2,axis=1)**0.5
            sortedDistanceIndexs = distance.argsort()
            countDict = {}
            for j in range(k):
                countY = self.ytr[sortedDistanceIndexs[j]]
                countDict[countY] = countDict.get(countY,0) + 1
            sortedCountDict = sorted(countDict.items(),key=operator.itemgetter(1),reverse=True)
            Ypred[i] = sortedCountDict[0][0]
        return Ypred
```

```
%%time
# KNN对图像集做分类，计算准确率
top_num = 50
train_data = unpickle('G:/大三上/数据科学/CIFAR10_train/cifar-10-batches-py/data_batch_4')
test_data = unpickle('G:/大三上/数据科学/CIFAR10_train/cifar-10-batches-py/test_batch')

knn = KNearestNeighbor()
knn.train(train_data[b'data'],np.array(train_data[b'labels']))
Ypred = knn.predict(test_data[b'data'][:top_num,:],3)

accur = np.sum(np.array(Ypred)==np.array(test_data[b'labels'][:top_num])) / len(Ypred)
print(accur)
```

```
0.2
Wall time: 6.6 s
```

KNN 算法最后的准确率只有 20%左右，而且与测试时间也比较长。最近邻分类的过程是通过比对所有数据集中训练集的图片来完成的，所以必须将所有图片读取在内存中，容易造成内存爆炸；其次，对一幅图像进行判断类别，需要比对所有训练集的图像，识别的过程消耗计算量巨大。相比之下，CNN 的效果更优秀。