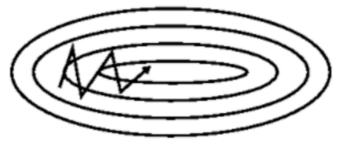
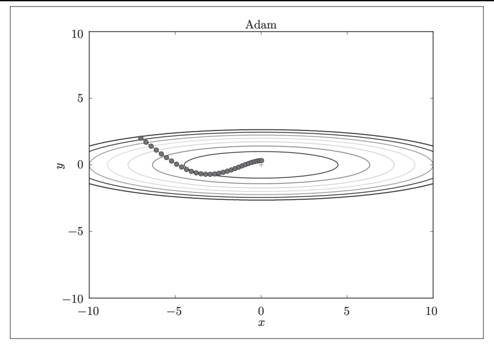
组会汇报

1.28

优化器

An overview of gradient descent optimization algorithms

	特点	缺点	
BGD	整个训练集数据计算梯度	慢	陷入局部最小值或者鞍点
SGD/MBGD	随机一个样本/每次一小批	不一定是全局最优	
Momentum/NAG	$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$ $\theta = \theta - v_t$	缺乏适应性	 Image 2: SGD without momentum  Image 3: SGD with momentum
Adagrad/Adadelta	为参数的每个元素适当地调整学习率	学习越深入，更新的幅度就越小	$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i}$
Adam	计算每个参数的自适应学习率		 图6-7 基于Adam的最优化的更新路径

An overview of gradient descent optimization algorithms*

Sebastian Ruder
Insight Centre for Data Analytics, NUI Galway
Aylien Ltd., Dublin
ruder.sebastian@gmail.com

Abstract

Gradient descent optimization algorithms, while increasingly popular, are often used as black-box optimizers, as practical explanations of their strengths and weaknesses are hard to come by. This article aims to provide the reader with intuitions with regard to the behaviour of different algorithms that will allow her to put them to use. In the course of this overview, we look at different variants of gradient descent, summarize challenges, introduce the most common optimization algorithms, review architectures in a parallel and distributed setting, and investigate additional strategies for optimizing gradient descent.

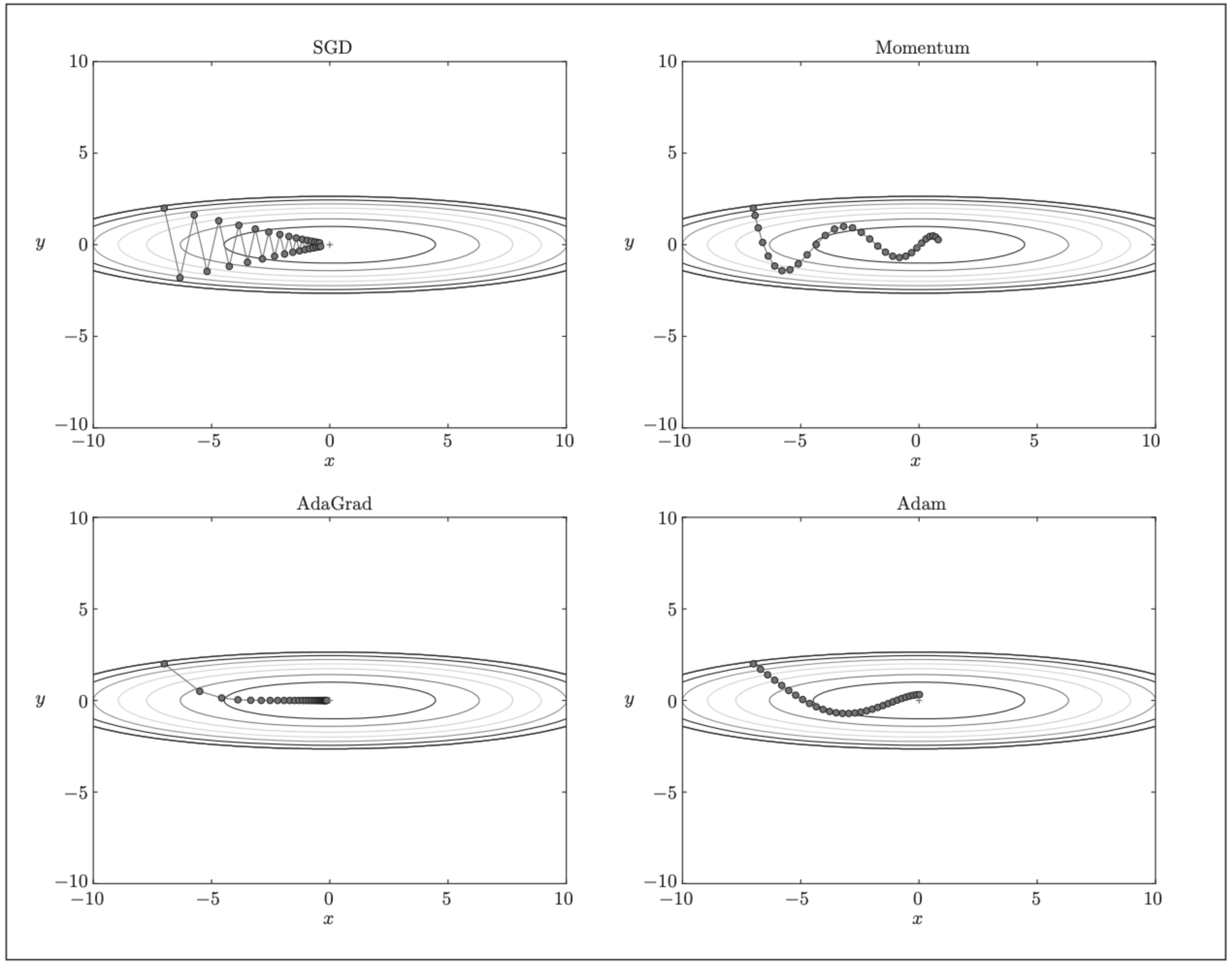
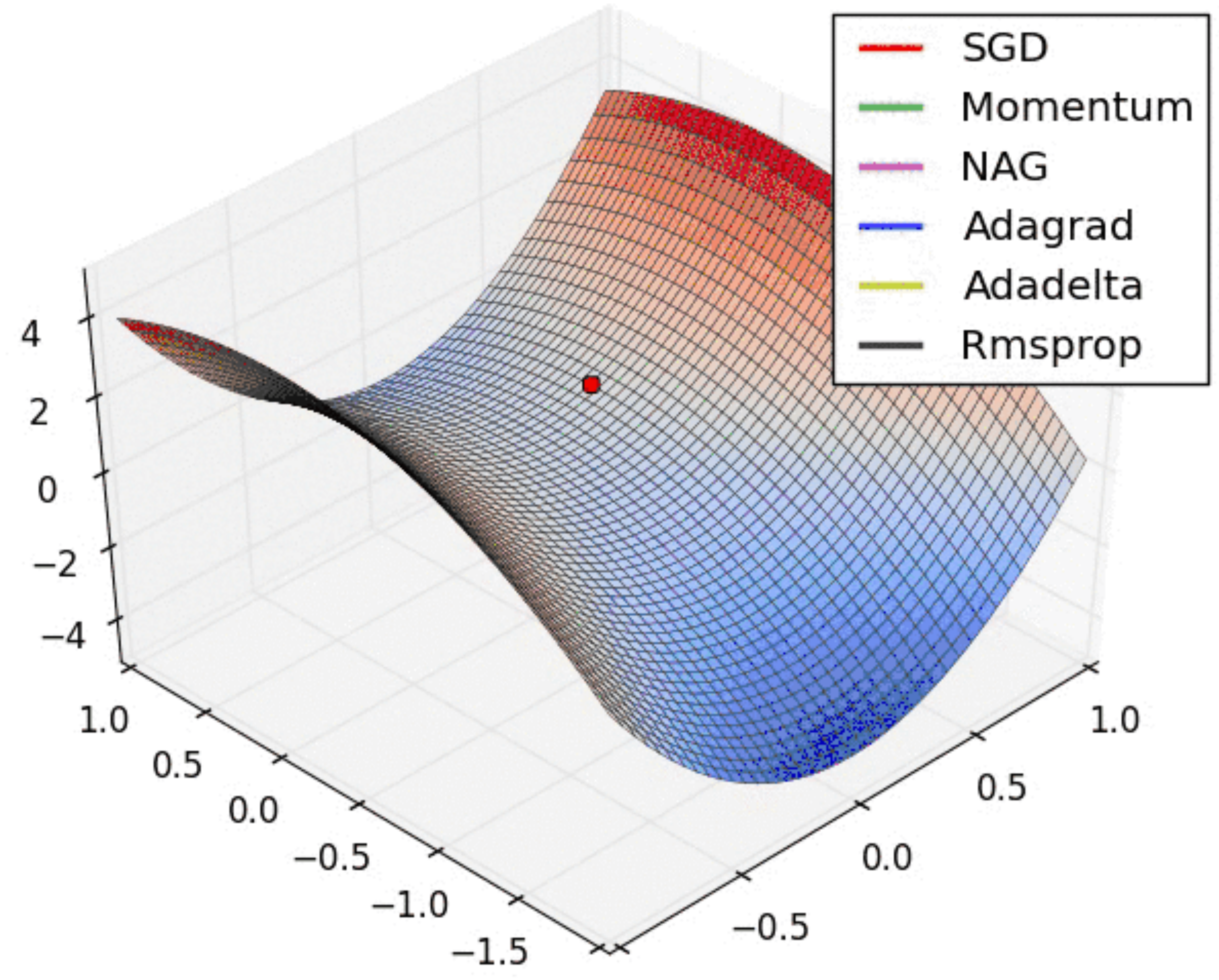
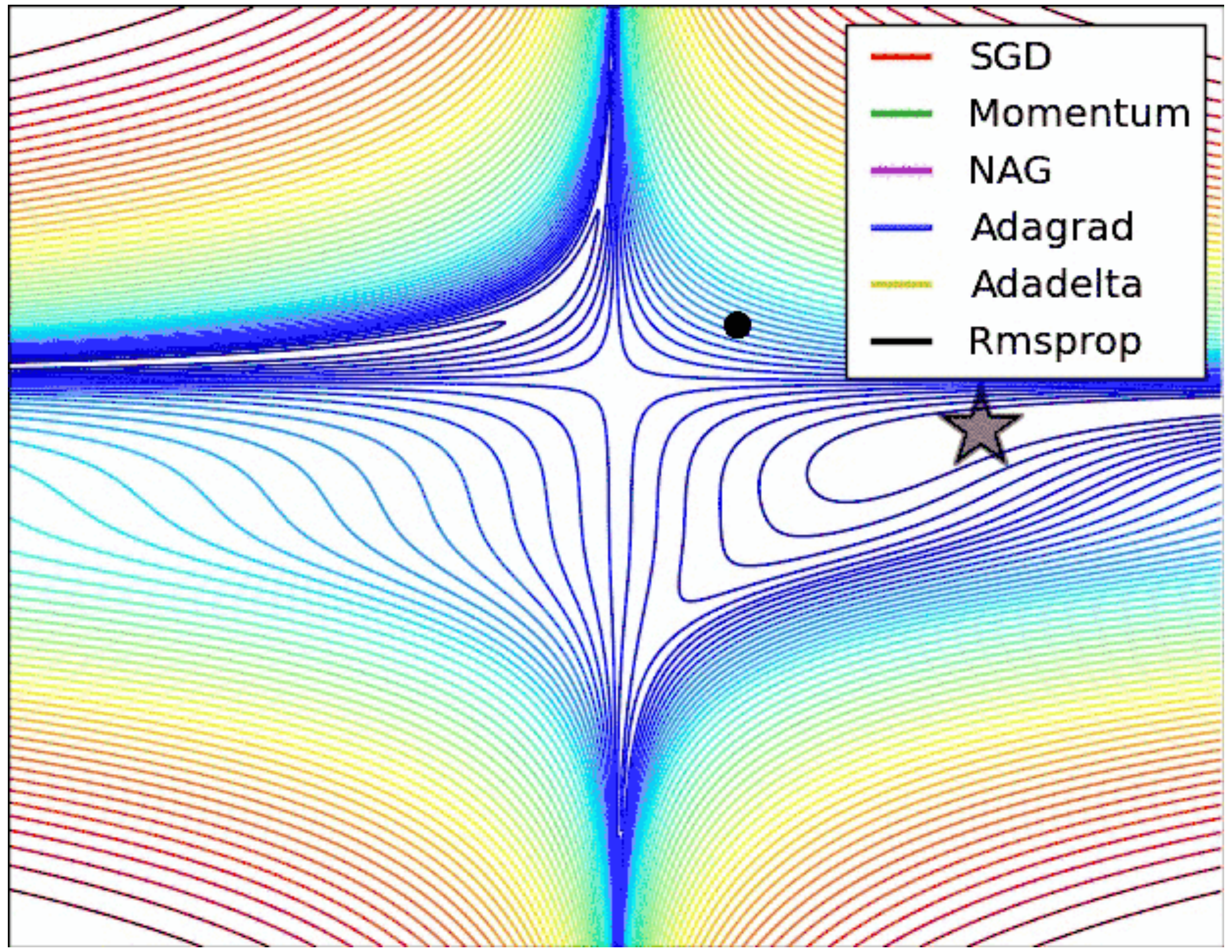


图6-8　最优化方法的比较：SGD、Momentum、AdaGrad、Adam

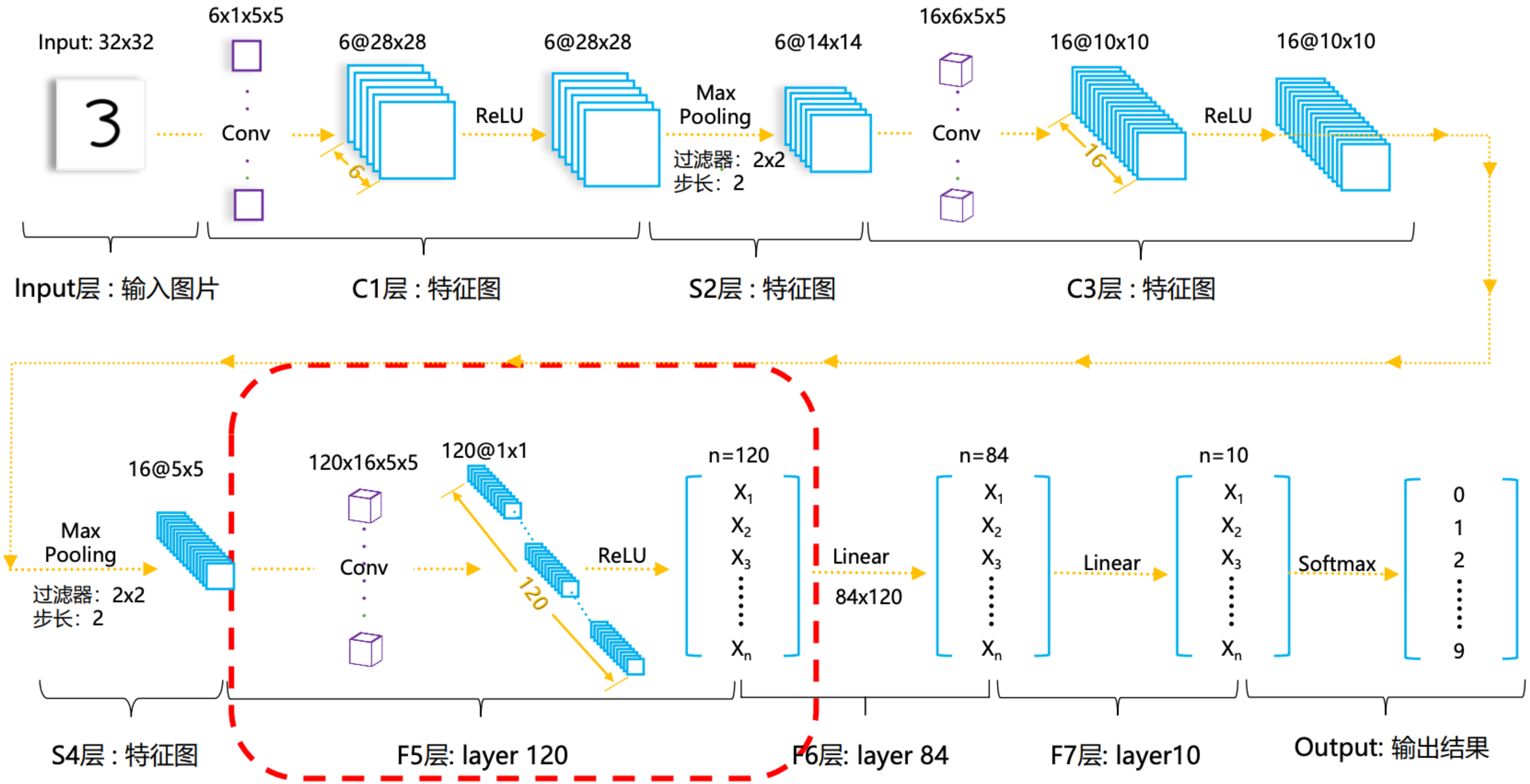




LeNet5与Mnist结合

```
RuntimeError: Calculated padded input size per channel: (4 x 4). Kernel size: (5 x 5). Kernel size can't be greater than actual input size
```

	Input	C1	S2	C3	S4	F5	F6	Output
原LeNet	32*32	6@28*28	6@14*14	16@10*10	16@5*5	120@1*1	84	10
Mnist	28*28	6@24*24	6@12*12	16@8*8	16@4*4	120@1*1	84	10



Input: 32x32
3
6x1x5x5
Conv
6@28x28
6
ReLU
6@28x28
Max Pooling
过滤器：2x2
步长：2
6@14x14
16x6x5x5
Conv
16@10x10
16
ReLU
16@10x10
Input层 : 输入图片
C1层 : 特征图
S2层 : 特征图
C3层 : 特征图
Max Pooling
过滤器：2x2
步长：2
16@5x5
120x16x5x5
Conv
120@1x1
120
ReLU
n=120
X1
X2
X3
Xn
Linear
84x120
n=84
Linear
n=10
Softmax
0
1
2
9
S4层 : 特征图
F5层: layer 120
F6层: layer 84
F7层: layer10
Output: 输出结果

改进1

- 直接将F5这次卷积定义为全连接

```python
# 卷积层
self.conv = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=0),  # 28x28x1-->24x24x6
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),  # 12x12x6
    nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),  # 8x8x16
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),  # 4x4x16 = 256
)
# 全连接层
self.fc = nn.Sequential(
    nn.Linear(256, 120),
    nn.ReLU(),
    nn.Linear(120, 4),
    nn.ReLU(),
    nn.Linear(84, 10),
)
```

```
epoch 0, loss 0.7103, train accuracy 0.760, test accuracy 0.967
epoch 1, loss 0.0978, train accuracy 0.969, test accuracy 0.980
epoch 2, loss 0.0669, train accuracy 0.979, test accuracy 0.980
epoch 3, loss 0.0532, train accuracy 0.983, test accuracy 0.985
epoch 4, loss 0.0445, train accuracy 0.986, test accuracy 0.987
epoch 5, loss 0.0362, train accuracy 0.988, test accuracy 0.987
epoch 6, loss 0.0308, train accuracy 0.990, test accuracy 0.988
epoch 7, loss 0.0275, train accuracy 0.991, test accuracy 0.989
epoch 8, loss 0.0238, train accuracy 0.992, test accuracy 0.989
epoch 9, loss 0.0210, train accuracy 0.993, test accuracy 0.990
```

改进2

- C1的Padding设置为2

```python
# 卷积层
self.conv = nn.Sequential(
    nn.Conv2d(1, 6, kernel_size=5, stride=1, padding=2),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=2, stride=2),
    nn.Conv2d(16, 120, kernel_size=5, stride=1, padding=0),
    nn.ReLU(),
)
# 全连接层
self.fc = nn.Sequential(

    nn.Linear(120, 84),
    nn.ReLU(),
    nn.Linear(84, 10),
)
```

```
epoch 0, loss 0.7952, train accuracy 0.734, test accuracy 0.971
epoch 1, loss 0.0883, train accuracy 0.972, test accuracy 0.983
epoch 2, loss 0.0621, train accuracy 0.982, test accuracy 0.986
epoch 3, loss 0.0465, train accuracy 0.985, test accuracy 0.987
epoch 4, loss 0.0408, train accuracy 0.987, test accuracy 0.987
epoch 5, loss 0.0327, train accuracy 0.989, test accuracy 0.989
epoch 6, loss 0.0283, train accuracy 0.991, test accuracy 0.989
epoch 7, loss 0.0247, train accuracy 0.992, test accuracy 0.987
epoch 8, loss 0.0211, train accuracy 0.993, test accuracy 0.987
epoch 9, loss 0.0193, train accuracy 0.994, test accuracy 0.990
```

代码细节

```python
# argument: Output, 0/1
# 0/1: column-0; row-1
# Return:a namedtuple(values, indices)
ret_val, predicted = torch.max(output, 1)
_, predicted = torch.max(output, 1)
```

Docs > torch > torch.max

Parameters:

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the dimension to reduce.
- **keepdim** (*bool*) – whether the output tensor has `dim` retained or not. Default: `False`.

Keyword Arguments:

out (*tuple, optional*) – the result tuple of two output tensors (max, max_indices)

Example:

```
>>> a = torch.randn(4, 4)
>>> a
tensor([[-1.2360, -0.2942, -0.1222,  0.8475],
        [ 1.1949, -1.1127, -2.2379, -0.6702],
        [ 1.5717, -0.9207,  0.1297, -1.8768],
        [-0.6172,  1.0036, -0.6060, -0.2432]])
>>> torch.max(a, 1)
torch.return_types.max(values=tensor([0.8475, 1.1949, 1.5717, 1.0036]),
indices=tensor([3, 0, 0, 1]))
```

```python
test_img = Image.open('./3.png')
resize_img = test_img.resize((28,28))
gray_img = resize_img.convert('L')

#squeeze()
#unsqueeze()
trans_img = transform(gray_img).unsqueeze(0)
input_img = trans_img.to(device)
```

```python
trans = torchvision.transforms.Compose([
        torchvision.transforms.ToTensor()])
```

Conversion Transforms

`ToPILImage`([mode])	Convert a tensor or an ndarray to PIL Image.
`ToTensor`()	Convert a `PIL Image` or `numpy.ndarray` to tensor.
`PILToTensor`()	Convert a `PIL Image` to a tensor of the same type.

TORCH.UNSQUEEZE

torch.unsqueeze(*input*, *dim*) → Tensor

Returns a new tensor with a dimension of size one inserted at the specified position.

The returned tensor shares the same underlying data with this tensor.

A `dim` value within the range `[-input.dim() - 1, input.dim() + 1)` can be used. Negative `dim` will correspond to `unsqueeze()` applied at `dim` = `dim + input.dim() + 1`.

Parameters:

- **input** (*Tensor*) – the input tensor.
- **dim** (*int*) – the index at which to insert the singleton dimension

预测结果

```
begin to recogition
output:tensor([[-1.5554, -0.6227,  0.5524,  0.2460, -1.5750,  1.7873,  0.8835, -0.7103,
          2.7275, -2.7394]], grad_fn=<AddmmBackward0>)
output shape:torch.Size([1, 10])
output size:torch.Size([1, 10])
ret_val:tensor([2.7275], grad_fn=<MaxBackward0>)
predicted:tensor([8])
tensor([8])
result is :8
```

```
begin to recogition
output:tensor([[-1.5002, -3.6027,  0.4499,  4.3016, -0.9503,  1.5012, -5.2479, -0.8262,
          1.1664,  3.0939]], grad_fn=<AddmmBackward0>)
output shape:torch.Size([1, 10])
output size:torch.Size([1, 10])
ret_val:tensor([4.3016], grad_fn=<MaxBackward0>)
predicted:tensor([3])
tensor([3])
result is :3
```

```
begin to recogition
output:tensor([[-1.1553, -0.1450, -1.0998, -0.1578, -1.0885,  3.5577, -3.4440,  2.7548,
         -3.1142,  4.3891]], grad_fn=<AddmmBackward0>)
output shape:torch.Size([1, 10])
output size:torch.Size([1, 10])
ret_val:tensor([4.3891], grad_fn=<MaxBackward0>)
predicted:tensor([9])
tensor([9])
result is :9
```

```
epoch 0, loss 1.2744, train accuracy 0.513, test accuracy 0.685
epoch 1, loss 0.5536, train accuracy 0.790, test accuracy 0.822
epoch 2, loss 0.4363, train accuracy 0.838, test accuracy 0.854
epoch 3, loss 0.3835, train accuracy 0.858, test accuracy 0.836
epoch 4, loss 0.3519, train accuracy 0.869, test accuracy 0.867
epoch 5, loss 0.3306, train accuracy 0.877, test accuracy 0.870
epoch 6, loss 0.3132, train accuracy 0.883, test accuracy 0.874
epoch 7, loss 0.2992, train accuracy 0.889, test accuracy 0.882
epoch 8, loss 0.2856, train accuracy 0.894, test accuracy 0.879
epoch 9, loss 0.2755, train accuracy 0.898, test accuracy 0.886
```