Batch Normalization

In [1]:

```
import d2l
from mxnet import autograd, gluon, init, nd
from mxnet.gluon import nn

def batch_norm(X, gamma, beta, moving_mean, moving_var, eps, momentum):
    # Use autograd to determine whether the current mode is training mode or predi
ction mode.
    if not autograd.is_training():
        # use the moving average
        X_hat = (X - moving_mean) / nd.sqrt(moving_var + eps)
    else:
        assert len(X.shape) in (2, 4)
        if len(X.shape) == 2:  # fully connected layer
            mean = X.mean(axis=0)
            var = ((X - mean) ** 2).mean(axis=0)
        else:                  # convolution, hence per layer
            mean = X.mean(axis=(0, 2, 3), keepdims=True)
            var = ((X - mean) ** 2).mean(axis=(0, 2, 3), keepdims=True)
        # In training mode, the current mean and variance are used for the standar
dization.
        X_hat = (X - mean) / nd.sqrt(var + eps)
        # Update the mean and variance of the moving average.
        moving_mean = momentum * moving_mean + (1.0 - momentum) * mean
        moving_var = momentum * moving_var + (1.0 - momentum) * var
    Y = gamma * X_hat + beta  # Scale and shift.
    return Y, moving_mean, moving_var
```

BatchNorm Layer

In [2]:
```
class BatchNorm(nn.Block):
    def __init__(self, num_features, num_dims, **kwargs):
        super(BatchNorm, self).__init__(**kwargs)
        if num_dims == 2:
            shape = (1, num_features)
        else:
            shape = (1, num_features, 1, 1)
        # The scale parameter and the shift parameter involved in gradient finding
and iteration are initialized to 0 and 1 respectively.
        self.gamma = self.params.get('gamma', shape=shape, init=init.One())
        self.beta = self.params.get('beta', shape=shape, init=init.Zero())
        # All the variables not involved in gradient finding and iteration are ini
tialized to 0 on the CPU.
        self.moving_mean = nd.zeros(shape)
        self.moving_var = nd.zeros(shape)

    def forward(self, X):
        # If X is not on the CPU, copy moving_mean and moving_var to the device wh
ere X is located.
        if self.moving_mean.context != X.context:
            self.moving_mean = self.moving_mean.copyto(X.context)
            self.moving_var = self.moving_var.copyto(X.context)
        # Save the updated moving_mean and moving_var.
        Y, self.moving_mean, self.moving_var = batch_norm(
            X, self.gamma.data(), self.beta.data(), self.moving_mean,
            self.moving_var, eps=1e-5, momentum=0.9)
        return Y
```

LeNet with Batch Norm

In [3]:

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
        BatchNorm(6, num_dims=4),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(16, kernel_size=5),
        BatchNorm(16, num_dims=4),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120),
        BatchNorm(120, num_dims=2),
        nn.Activation('sigmoid'),
        nn.Dense(84),
        BatchNorm(84, num_dims=2),
        nn.Activation('sigmoid'),
        nn.Dense(10))
```

Training Batch Norm LeNet

In [4]:

```
lr, num_epochs, batch_size, ctx = 1.0, 5, 256, d2l.try_gpu()
net.initialize(ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx, num_epochs)
```

```
training on gpu(0)
epoch 1, loss 0.6847, train acc 0.752, test acc 0.792, time 3.7 sec
epoch 2, loss 0.4129, train acc 0.850, test acc 0.855, time 3.6 sec
epoch 3, loss 0.3586, train acc 0.870, test acc 0.866, time 3.6 sec
epoch 4, loss 0.3320, train acc 0.879, test acc 0.874, time 3.5 sec
epoch 5, loss 0.3108, train acc 0.888, test acc 0.864, time 3.5 sec
```

Let's have a look at the scale parameter `gamma` and the shift parameter `beta` learned from the first batch normalization layer.

In [5]:
```
net[1].gamma.data().reshape((-1,)), net[1].beta.data().reshape((-1,))
```

Out[5]:
```
(
 [1.7873654  0.61735505 1.7933072  1.687522   1.5147648  1.6866305 ]
 <NDArray 6 @gpu(0)>,
 [ 0.91468376  0.50423574 -0.01995357  0.7233319  -0.65631443 -1.8838943 ]
 <NDArray 6 @gpu(0)>)
```

Batch Norm in Gluon

In [6]:

```
net = nn.Sequential()
net.add(nn.Conv2D(6, kernel_size=5),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Conv2D(16, kernel_size=5),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.MaxPool2D(pool_size=2, strides=2),
        nn.Dense(120),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.Dense(84),
        nn.BatchNorm(),
        nn.Activation('sigmoid'),
        nn.Dense(10))
```

Use the same hyper-parameter to carry out the training. Note that as always the Gluon variant runs a lot faster since the code that is being executed is compiled C++/CUDA rather than interpreted Python.

In [7]:
```
net.initialize(ctx=ctx, init=init.Xavier())
trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx, num_epochs)
```

```
training on gpu(0)
epoch 1, loss 0.6533, train acc 0.768, test acc 0.802, time 2.3 sec
epoch 2, loss 0.4023, train acc 0.854, test acc 0.876, time 2.4 sec
epoch 3, loss 0.3536, train acc 0.872, test acc 0.863, time 2.4 sec
epoch 4, loss 0.3251, train acc 0.883, test acc 0.885, time 2.4 sec
epoch 5, loss 0.3070, train acc 0.889, test acc 0.871, time 2.3 sec
```