Desarrollo del código Perl/MXNet del notebook #3

Grupo 5

Integrantes: Almeida Edison, Laje Adrian, Mejia Leonardo y Willian Medina

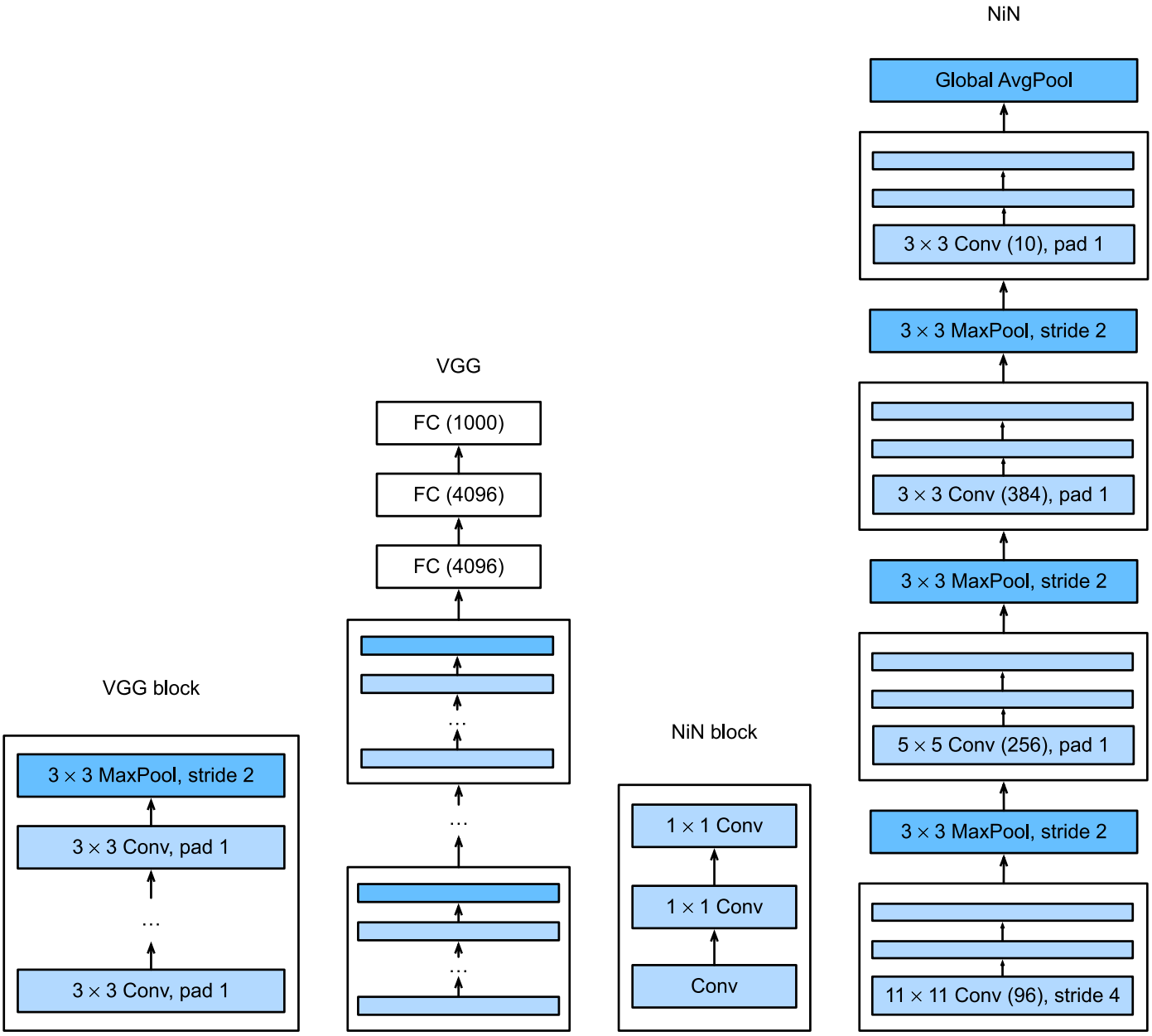
7.3. Network in Network (NiN)

LeNet, AlexNet, and VGG all share a common design pattern: extract features exploiting spatial structure via a sequence of convolution and pooling layers and then post-process the representations via fully-connected layers. The improvements upon LeNet by AlexNet and VGG mainly lie in how these later networks widen and deepen these two modules. Alternatively, one could imagine using fully-connected layers earlier in the process. However, a careless use of dense layers might give up the spatial structure of the representation entirely, network in network (NiN) blocks offer an alternative. They were proposed based on a very simple insight: to use an MLP on the channels for each pixel separately ().

7.3.1. NiN Blocks

Recall that the inputs and outputs of convolutional layers consist of four-dimensional tensors with axes corresponding to the example, channel, height, and width. Also recall that the inputs and outputs of fully-connected layers are typically two-dimensional tensors corresponding to the example and feature. The idea behind NiN is to apply a fully-connected layer at each pixel location (for each height and width). If we tie the weights across each spatial location, we could think of this as a 1×1 convolutional layer (as described in Section 6.4) or as a fully-connected layer acting independently on each pixel location. Another way to view this is to think of each element in the spatial dimension (height and width) as equivalent to an example and a channel as equivalent to a feature.

Fig. 7.3.1 illustrates the main structural differences between VGG and NiN, and their blocks. The NiN block consists of one convolutional layer followed by two 1×1 convolutional layers that act as per-pixel fully-connected layers with ReLU activations. The convolution window shape of the first layer is typically set by the user. The subsequent window shapes are fixed to 1×1.



In [1]:

```
use strict;
use warnings;
use Data::Dump qw(dump);
use AI::MXNet qw(mx);
use d2l;
use d2l::Animator;
IPerl->load_plugin('Chart::Plotly'); # Jupyter
use Chart::Plotly qw(show_plot); # Locally

sub nin_block{
    my ($out_channels, $kernel_size, $strides, $padding, $in_channnels) = @_;
    my $blk = gluon->nn->Sequential();
    $blk->add( gluon->nn->Conv2D( $out_channels, $kernel_size, $strides, $padding, acti
            gluon->nn->Conv2D($out_channels, kernel_size => 1, activation => 'relu', ir
           gluon->nn->Conv2D($out_channels, kernel_size => 1, activation =>'relu', in_c
    return $blk;
}
```

7.3.2. NiN Model

The original NiN network was proposed shortly after AlexNet and clearly draws some inspiration. NiN uses convolutional layers with window shapes of $11x11$, $5x5$, and $3x3$, and the corresponding numbers of output channels are the same as in AlexNet. Each NiN block is followed by a maximum pooling layer with a stride of 2 and a window shape of $3x3$.

One significant difference between NiN and AlexNet is that NiN avoids fully-connected layers altogether. Instead, NiN uses an NiN block with a number of output channels equal to the number of label classes, followed by a global average pooling layer, yielding a vector of logits. One advantage of NiN's design is that it significantly reduces the number of required model parameters. However, in practice, this design sometimes requires increased model training time.

In [2]:

```
my $net = gluon->nn->Sequential();
$net->add(
        nin_block(96, 11, 4, 0, 1),
        gluon->nn->MaxPool2D(pool_size => 3, strides => 2),
        nin_block(256, 5, 1, 2, 96),
        gluon->nn->MaxPool2D(pool_size => 3, strides => 2),
        nin_block(384, 3, 1, 1, 256),
        gluon->nn->MaxPool2D(pool_size => 3, strides =>2),
        gluon->nn->Dropout(0.5),
       # # There are 10 label classes
        nin_block(10, 3, 1, 1, 384),
        # The global average pooling layer automatically sets the window shape
        # to the height and width of the input
        gluon->nn->GlobalAvgPool2D(),
        # Transform the four-dimensional output into two-dimensional output
        # with a shape of (batch size, 10)
        gluon->nn->Flatten()
);
print $net;
```

```
Sequential(
  (0): Sequential(
    (0): Conv2D(1 -> 96, kernel_size=(11,11), stride=(4,4))
    (1): Conv2D(96 -> 96, kernel_size=(1,1), stride=(1,1))
    (2): Conv2D(96 -> 96, kernel_size=(1,1), stride=(1,1))
  )
  (1): MaxPool2D(size=(3,3), stride=(2,2), padding=(0,0), ceil_mode=0)
  (2): Sequential(
    (0): Conv2D(96 -> 256, kernel_size=(5,5), stride=(1,1), padding=(2,2))
    (1): Conv2D(256 -> 256, kernel_size=(1,1), stride=(1,1))
    (2): Conv2D(256 -> 256, kernel_size=(1,1), stride=(1,1))
  )
  (3): MaxPool2D(size=(3,3), stride=(2,2), padding=(0,0), ceil_mode=0)
  (4): Sequential(
    (0): Conv2D(256 -> 384, kernel_size=(3,3), stride=(1,1), padding=(1,1))
    (1): Conv2D(384 -> 384, kernel_size=(1,1), stride=(1,1))
    (2): Conv2D(384 -> 384, kernel_size=(1,1), stride=(1,1))
  )
  (5): MaxPool2D(size=(3,3), stride=(2,2), padding=(0,0), ceil_mode=0)
  (6): Dropout(p = 0.5)
  (7): Sequential(
    (0): Conv2D(384 -> 10, kernel_size=(3,3), stride=(1,1), padding=(1,1))
    (1): Conv2D(10 -> 10, kernel_size=(1,1), stride=(1,1))
    (2): Conv2D(10 -> 10, kernel_size=(1,1), stride=(1,1))
  )
  (8): GlobalAvgPool2D(size=(1,1), stride=(1,1), padding=(0,0), ceil_mode=1)
  (9): Flatten
)
```

Out[2]:

```
1
```

We create a data example to see the output shape of each block.

In [3]:

```
my $X = mx->nd->random->uniform(shape => [1, 1, 224, 224]);
$net->initialize();

my $i = 0;
while(defined($net->[$i]) == 1){
    $X = $net->[$i]->($X);
    print($net->[$i]->name . ' output shape: '. dump ($X->shape) . "\n");
    $i = $i+1;
}
```

```
sequential1 output shape: [1, 96, 54, 54]
pool0 output shape: [1, 96, 26, 26]
sequential2 output shape: [1, 256, 26, 26]
pool1 output shape: [1, 256, 12, 12]
sequential3 output shape: [1, 384, 12, 12]
pool2 output shape: [1, 384, 5, 5]
dropout0 output shape: [1, 384, 5, 5]
sequential4 output shape: [1, 10, 5, 5]
pool3 output shape: [1, 10, 1, 1]
flatten0 output shape: [1, 10]
```

7.3.3. Training

As before we use Fashion-MNIST to train the model. NiN's training is similar to that for AlexNet and VGG.

In [4]:

```
my ($lr, $num_epochs, $batch_size) = (0.1, 10, 128);
my ($train_iter, $test_iter) = d2l->load_data_fashion_mnist(batch_size => $batch_size,
print 1;
```

```
1
```

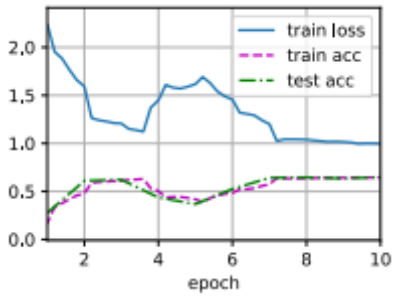
Out[4]:

```
1
```

In [ ]:

```
my $animator = d2l->train_ch6($net, $train_iter, $test_iter, $num_epochs, $lr, d2l->try
$animator->plot;
```

Expected results are:



7.3.4. Summary

NiN uses blocks consisting of a convolutional layer and multiple convolutional layers. This can be used within the convolutional stack to allow for more per-pixel nonlinearity.

NiN removes the fully-connected layers and replaces them with global average pooling (i.e., summing over all locations) after reducing the number of channels to the desired number of outputs (e.g., 10 for Fashion-MNIST).

Removing the fully-connected layers reduces overfitting. NiN has dramatically fewer parameters.

The NiN design influenced many subsequent CNN designs.