```
        # Compute the per pixel conditional probabilities
        xprob = (px * data + (1-px) * (1-data))
        # Take the product
        xprob = xprob.prod(0) * py
        print('Unnormalized Probabilities', xprob)
        # Normalize
        xprob = xprob / xprob.sum()
        print('Normalized Probabilities', xprob)
```

```
Unnormalized Probabilities
[0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
<NDArray 10 @cpu(0)>
Normalized Probabilities
[nan nan nan nan nan nan nan nan nan nan]
<NDArray 10 @cpu(0)>
```

This went horribly wrong! To find out why, let's look at the per pixel probabilities. They're typically numbers between 0.001 and 1. We are multiplying 784 of them. At this point it is worth mentioning that we are calculating these numbers on a computer, hence with a fixed range for the exponent. What happens is that we experience *numerical underflow*, i.e. multiplying all the small numbers leads to something even smaller until it is rounded down to zero. At that point we get division by zero with `nan` as a result.

To fix this we use the fact that $\log ab = \log a + \log b$, i.e. we switch to summing logarithms. This will get us unnormalized probabilities in log-space. To normalize terms we use the fact that

$$\frac{\exp(a)}{\exp(a) + \exp(b)} = \frac{\exp(a+c)}{\exp(a+c) + \exp(b+c)}$$

In particular, we can pick $c = -\max(a, b)$, which ensures that at least one of the terms in the denominator is 1.

```
In [4]: logpx = nd.log(px)
        logpxneg = nd.log(1-px)
        logpy = nd.log(py)

        def bayespost(data):
            # We need to incorporate the prior probability p(y) since p(y|x) is
            # proportional to p(x|y) p(y)
            logpost = logpy.copy()
            logpost += (logpx * data + logpxneg * (1-data)).sum(0)
            # Normalize to prevent overflow or underflow by subtracting the largest
            # value
            logpost -= nd.max(logpost)
            # Compute the softmax using logpx
            post = nd.exp(logpost).asnumpy()
            post /= np.sum(post)
            return post

        fig, figarr = plt.subplots(2, 10, figsize=(10, 3))

        # Show 10 images
        ctr = 0
        for data, label in mnist_test:
            x = data.reshape((784,1))
```

the average loss of training samples:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w},b}{\operatorname{argmin}}\ L(\mathbf{w}, b).$$

Optimization Algorithm

When the model and loss function are in a relatively simple format, the solution to the aforementioned loss minimization problem can be expressed analytically in a closed form solution, involving matrix inversion. This is very elegant, it allows for a lot of nice mathematical analysis, *but* it is also very restrictive insofar as this approach only works for a small number of cases (e.g. multilayer perceptrons and nonlinear layers are no go). Most deep learning models do not possess such analytical solutions. The value of the loss function can only be reduced by a finite update of model parameters via an incremental optimization algorithm.

The mini-batch stochastic gradient descent is widely used for deep learning to find numerical solutions. Its algorithm is simple: first, we initialize the values of the model parameters, typically at random; then we iterate over the data multiple times, so that each iteration may reduce the value of the loss function. In each iteration, we first randomly and uniformly sample a mini-batch $\mathcal{B}$ consisting of a fixed number of training data examples; we then compute the derivative (gradient) of the average loss on the mini batch with regard to the model parameters. Finally, the product of this result and a predetermined step size $\eta > 0$ is used to change the parameters in the direction of the minimum of the loss. In math we have (∂ denotes the partial derivative):

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b)$$

For quadratic losses and linear functions we can write this out explicitly as follows. Note that $\mathbf{w}$ and $\mathbf{x}$ are vectors. Here the more elegant vector notation makes the math much more readable than expressing things in terms of coefficients, say $w_1, w_2, \ldots w_d$.

$$\begin{aligned}
\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) &= w - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right), \\
b \leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \partial_b l^{(i)}(\mathbf{w}, b) &= b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right).
\end{aligned}$$

In the above equation $|\mathcal{B}|$ represents the number of samples (batch size) in each mini-batch, η is referred to as learning rate' and takes a positive number. It should be emphasized that the values of the batch size and learning rate are set somewhat manually and are typically not learned through model training. Therefore, they are referred to as *hyper-parameters*. What we usually call *tuning hyper-parameters* refers to the adjustment of these terms. In the worst case this is performed through repeated trial and error until the appropriate hyper-parameters are found. A better approach is to learn these as parts of model training. This is an advanced topic and we do not cover them here for the sake of simplicity.

on Linear Regression. There the loss is given by

$$l(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2.$$

Recall that $\mathbf{x}^{(i)}$ are the observations, $y^{(i)}$ are labels, and $(\mathbf{w}, b)$ are the weight and bias parameters respectively. To arrive at the new loss function which penalizes the size of the weight vector we need to add $\|\mathbf{w}\|^2$, but how much should we add? This is where the regularization constant (hyperparameter) λ comes in:

$$l(\mathbf{w}, b) + \frac{\lambda}{2}\|\boldsymbol{w}\|^2$$

$\lambda \geq 0$ governs the amount of regularization. For $\lambda = 0$ we recover the previous loss function, whereas for $\lambda > 0$ we ensure that $\mathbf{w}$ cannot grow too large. The astute reader might wonder why we are squaring the weight vector. This is done both for computational convenience since it leads to easy to compute derivatives, and for statistical performance, as it penalizes large weight vectors a lot more than small ones. The stochastic gradient descent updates look as follows:

$$w \leftarrow \left(1 - \frac{\eta\lambda}{|\mathcal{B}|}\right)\mathbf{w} - \frac{\eta}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} \mathbf{x}^{(i)}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right),$$

As before, we update $\mathbf{w}$ in accordance to the amount to which our estimate differs from the observation. However, we also shrink the size of $\mathbf{w}$ towards 0, i.e. the weight decays'. This is much more convenient than having to pick the number of parameters as we did for polynomials. In particular, we now have a continuous mechanism for adjusting the complexity of f. Small values of λ correspond to fairly unconstrained $\mathbf{w}$ whereas large values of λ constrain $\mathbf{w}$ considerably. Since we don't want to have large bias terms either, we often add b^2 as penalty, too.

4.5.2 High-dimensional Linear Regression

For high-dimensional regression it is difficult to pick the right' dimensions to omit. Weight-decay regularization is a much more convenient alternative. We will illustrate this below. But first we need to generate some data via

$$y = 0.05 + \sum_{i=1}^{d} 0.01 x_i + \epsilon \text{ where } \epsilon \sim \mathcal{N}(0, 0.01)$$

That is, we have additive Gaussian noise with zero mean and variance 0.01. In order to observe overfitting more easily we pick a high-dimensional problem with $d = 200$ and a deliberatly low number of training examples, e.g. 20. As before we begin with our import ritual (and data generation).

```
In [1]: import sys
        sys.path.insert(0, '..')

        %matplotlib inline
```

4.6 Dropout

In the previous chapter, we introduced one classical approach to regularize statistical models. We penalized the size (the ℓ_2 norm) of the weights, coercing them to take smaller values. In probabilistic terms we might say that this imposes a Gaussian prior on the value of the weights. But in more intuitive, functional terms, we can say that this encourages the model to spread out its weights among many features and not to depend too much on a small number of potentially spurious associations.

4.6.1 Overfitting Revisited

With great flexibility comes overfitting liability. Given many more features than examples, linear models can overfit. But when there are many more examples than features, linear models can usually be counted on not to overfit. Unfortunately this propensity to generalize well comes at a cost. For every feature, a linear model has to assign it either positive or negative weight. Linear models can't take into account nuanced interactions between features. In more formal texts, you'll see this phenomena discussed as the bias-variance tradeoff. Linear models have high bias, (they can only represent a small class of functions), but low variance (they give similar results across different random samples of the data).

Deep neural networks, however, occupy the opposite end of the bias-variance spectrum. Neural networks are so flexible because they aren't confined to looking at each feature individually. Instead, they can learn complex interactions among groups of features. For example, they might infer that Nigeria and Western Union appearing together in an email indicates spam but that Nigeria without Western Union does not connote spam.

Even for a small number of features, deep neural networks are capable of overfitting. As one demonstration of the incredible flexibility of neural networks, researchers showed that neural networks perfectly classify randomly labeled data. Let's think about what means. If the labels are assigned uniformly at random, and there are 10 classes, then no classifier can get better than 10% accuracy on holdout data. Yet even in these situations, when there is no true pattern to be learned, neural networks can perfectly fit the training labels.

4.6.2 Robustness through Perturbations

Let's think briefly about what we expect from a good statistical model. Obviously we want it to do well on unseen test data. One way we can accomplish this is by asking for what amounts to a simple' model. Simplicity can come in the form of a small number of dimensions, which is what we did when discussing fitting a function with monomial basis functions. Simplicity can also come in the form of a small norm for the basis funtions. This is what led to weight decay and ℓ_2 regularization. Yet a third way to impose some notion of simplicity is that the function should be robust under modest changes in the input. For instance, when we classify images, we would expect that alterations of a few pixels are mostly harmless.

In fact, this notion was formalized by Bishop in 1995, when he proved that Training with Input Noise is Equivalent to Tikhonov Regularization. That is, he connected the notion of having a smooth (and thus simple) function with one that is resilient to perturbations in the input. Fast forward to 2014. Given

the complexity of deep networks with many layers, enforcing smoothness just on the input misses out on what is happening in subsequent layers. The ingenious idea of Srivastava et al., 2014 was to apply Bishop's idea to the *internal* layers of the network, too, namely to inject noise into the computational path of the network while it's training.

A key challenge in this context is how to add noise without introducing undue bias. In terms of inputs $\mathbf{x}$, this is relatively easy to accomplish: simply add some noise $\epsilon \sim \mathcal{N}(0, \sigma^2)$ to it and use this data during training via $\mathbf{x}' = \mathbf{x} + \epsilon$. A key property is that in expectation $\mathbf{E}[\mathbf{x}'] = \mathbf{x}$. For intermediate layers, though, this might not be quite so desirable since the scale of the noise might not be appropriate. The alternative is to perturb coordinates as follows:

$$h' = \begin{cases} 0 & \text{with probability } p \\ \frac{h}{1-p} & \text{otherwise} \end{cases}$$

By design, the expectation remains unchanged, i.e. $\mathbf{E}[h'] = h$. This idea is at the heart of dropout where intermediate activations h are replaced by a random variable h' with matching expectation. The name dropout' arises from the notion that some neurons drop out' of the computation for the purpose of computing the final result. During training we replace intermediate activations with random variables

4.6.3 Dropout in Practice

Recall the *multilayer perceptron* with a hidden layer and 5 hidden units. Its architecture is given by

$$\begin{aligned} h &= \sigma(W_1 x + b_1) \\ o &= W_2 h + b_2 \\ \hat{y} &= \mathrm{softmax}(o) \end{aligned}$$

When we apply dropout to the hidden layer, it amounts to removing hidden units with probability p since their output is set to 0 with that probability. A possible result is the network shown below. Here h_2 and h_5 are removed. Consequently the calculation of y no longer depends on h_2 and h_5 and their respective gradient also vanishes when performing backprop. In this way, the calculation of the output layer cannot be overly dependent on any one element of $h_1, \ldots, h_5$. This is exactly what we want for regularization purposes to cope with overfitting. At test time we typically do not use dropout to obtain more conclusive results.

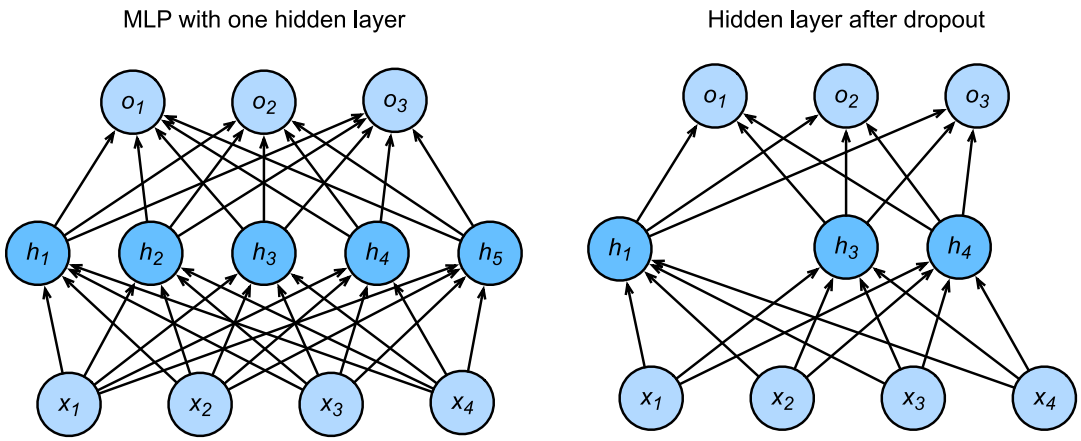


Fig. 4.4: MLP before and after dropout

4.6.4 Implementation from Scratch

To implement the dropout function we have to draw as many random variables as the input has dimensions from the uniform distribution $U[0, 1]$. According to the definition of dropout, we can implement it easily. The following `dropout` function will drop out the elements in the NDArray input `X` with the probability of `drop_prob`.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import autograd, gluon, init, nd
        from mxnet.gluon import loss as gloss, nn

        def dropout(X, drop_prob):
            assert 0 <= drop_prob <= 1
            # In this case, all elements are dropped out
            if drop_prob == 1:
                return X.zeros_like()
            mask = nd.random.uniform(0, 1, X.shape) > drop_prob
            return mask * X / (1.0-drop_prob)
```

Let us test how it works in a few examples. The dropout probability is 0, 0.5, and 1, respectively.

```
In [2]: X = nd.arange(16).reshape((2, 8))
        print(dropout(X, 0))
        print(dropout(X, 0.5))
        print(dropout(X, 1))
```

```
[[ 0.  1.  2.  3.  4.  5.  6.  7.]
 [ 8.  9. 10. 11. 12. 13. 14. 15.]]
<NDArray 2x8 @cpu(0)>
```

```
[[ 0.  0.  0.  0.  8. 10. 12.  0.]
 [16.  0. 20. 22.  0.  0.  0. 30.]]
<NDArray 2x8 @cpu(0)>

[[0. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0.]]
<NDArray 2x8 @cpu(0)>
```

Defining Model Parameters

Let's use the same dataset as used previously, namely Fashion-MNIST, described in the section Softmax Regression - Starting From Scratch. We will define a multilayer perceptron with two hidden layers. The two hidden layers both have 256 outputs.

```
In [3]: num_inputs, num_outputs, num_hiddens1, num_hiddens2 = 784, 10, 256, 256

        W1 = nd.random.normal(scale=0.01, shape=(num_inputs, num_hiddens1))
        b1 = nd.zeros(num_hiddens1)
        W2 = nd.random.normal(scale=0.01, shape=(num_hiddens1, num_hiddens2))
        b2 = nd.zeros(num_hiddens2)
        W3 = nd.random.normal(scale=0.01, shape=(num_hiddens2, num_outputs))
        b3 = nd.zeros(num_outputs)

        params = [W1, b1, W2, b2, W3, b3]
        for param in params:
            param.attach_grad()
```

Define the Model

The model defined below concatenates the fully connected layer and the activation function ReLU, using dropout for the output of each activation function. We can set the dropout probability of each layer separately. It is generally recommended to set a lower dropout probability closer to the input layer. Below we set it to 0.2 and 0.5 for the first and second hidden layer respectively. By using the `is_training` function described in the Autograd section we can ensure that dropout is only active during training.

```
In [4]: drop_prob1, drop_prob2 = 0.2, 0.5

        def net(X):
            X = X.reshape((-1, num_inputs))
            H1 = (nd.dot(X, W1) + b1).relu()
            # Use dropout only when training the model
            if autograd.is_training():
                # Add a dropout layer after the first fully connected layer
                H1 = dropout(H1, drop_prob1)
            H2 = (nd.dot(H1, W2) + b2).relu()
            if autograd.is_training():
                # Add a dropout layer after the second fully connected layer
                H2 = dropout(H2, drop_prob2)
            return nd.dot(H2, W3) + b3
```

Training and Testing

This is similar to the training and testing of multilayer perceptrons described previously.

```
In [5]: num_epochs, lr, batch_size = 10, 0.5, 256
        loss = gloss.SoftmaxCrossEntropyLoss()
        train_iter, test_iter = d2l.load_data_fashion_mnist(batch_size)
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size,
                      params, lr)
```

```
epoch 1, loss 1.1816, train acc 0.537, test acc 0.786
epoch 2, loss 0.5949, train acc 0.777, test acc 0.835
epoch 3, loss 0.4985, train acc 0.818, test acc 0.851
epoch 4, loss 0.4530, train acc 0.834, test acc 0.842
epoch 5, loss 0.4305, train acc 0.844, test acc 0.855
epoch 6, loss 0.4027, train acc 0.852, test acc 0.872
epoch 7, loss 0.3865, train acc 0.858, test acc 0.869
epoch 8, loss 0.3712, train acc 0.865, test acc 0.875
epoch 9, loss 0.3627, train acc 0.867, test acc 0.872
epoch 10, loss 0.3520, train acc 0.872, test acc 0.879
```

4.6.5 Concise Implementation

In Gluon, we only need to add the `Dropout` layer after the fully connected layer and specify the dropout probability. When training the model, the `Dropout` layer will randomly drop out the output elements of the previous layer at the specified dropout probability; the `Dropout` layer simply passes the data through during testing.

```
In [6]: net = nn.Sequential()
        net.add(nn.Dense(256, activation="relu"),
                # Add a dropout layer after the first fully connected layer
                nn.Dropout(drop_prob1),
                nn.Dense(256, activation="relu"),
                # Add a dropout layer after the second fully connected layer
                nn.Dropout(drop_prob2),
                nn.Dense(10))
        net.initialize(init.Normal(sigma=0.01))
```

Next, we will train and test the model.

```
In [7]: trainer = gluon.Trainer(net.collect_params(), 'sgd', {'learning_rate': lr})
        d2l.train_ch3(net, train_iter, test_iter, loss, num_epochs, batch_size, None,
                      None, trainer)
```

```
epoch 1, loss 1.2293, train acc 0.534, test acc 0.765
epoch 2, loss 0.5954, train acc 0.778, test acc 0.833
epoch 3, loss 0.4940, train acc 0.819, test acc 0.845
epoch 4, loss 0.4590, train acc 0.833, test acc 0.855
epoch 5, loss 0.4212, train acc 0.847, test acc 0.867
epoch 6, loss 0.4014, train acc 0.853, test acc 0.868
epoch 7, loss 0.3867, train acc 0.858, test acc 0.870
epoch 8, loss 0.3729, train acc 0.864, test acc 0.876
epoch 9, loss 0.3585, train acc 0.868, test acc 0.876
epoch 10, loss 0.3482, train acc 0.873, test acc 0.873
```

Summary

- Beyond controlling the number of dimensions and the size of the weight vector, dropout is yet another tool to avoid overfitting. Often all three are used jointly.
- Dropout replaces an activation h with a random variable h' with expected value h and with variance given by the dropout probability p.
- Dropout is only used during training.

Exercises

1. Try out what happens if you change the dropout probabilities for layers 1 and 2. In particular, what happens if you switch the ones for both layers?
2. Increase the number of epochs and compare the results obtained when using dropout with those when not using it.
3. Compute the variance of the the activation random variables after applying dropout.
4. Why should you typically not using dropout?
5. If changes are made to the model to make it more complex, such as adding hidden layer units, will the effect of using dropout to cope with overfitting be more obvious?
6. Using the model in this section as an example, compare the effects of using dropout and weight decay. What if dropout and weight decay are used at the same time?
7. What happens if we apply dropout to the individual weights of the weight matrix rather than the activations?
8. Replace the dropout activation with a random variable that takes on values of $[0, \gamma/2, \gamma]$. Can you design something that works better than the binary dropout function? Why might you want to use it? Why not?

References

[1] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). JMLR

Scan the QR Code to Discuss

- What are the advantages and disadvantages over training on a smaller minibatch?

Scan the QR Code to Discuss

4.8 Numerical Stability and Initialization

So far we covered the tools needed to implement multilayer perceptrons, how to solve regression and classification problems, and how to control capacity. However, we took initialization of the parameters for granted, or rather simply assumed that they would not be particularly relevant. In the following we will look at them in more detail and discuss some useful heuristics.

Secondly, we were not particularly concerned with the choice of activation. Indeed, for shallow networks this is not very relevant. For deep networks, however, design choices of nonlinearity and initialization play a crucial role in making the optimization algorithm converge relatively rapidly. Failure to be mindful of these issues can lead to either exploding or vanishing gradients.

4.8.1 Vanishing and Exploding Gradients

Consider a deep network with d layers, input $\mathbf{x}$ and output $\mathbf{o}$. Each layer satisfies:

$$\mathbf{h}^{t+1} = f_t(\mathbf{h}^t) \text{ and thus } \mathbf{o} = f_d \circ \ldots \circ f_1(\mathbf{x})$$

If all activations and inputs are vectors, we can write the gradient of $\mathbf{o}$ with respect to any set of parameters $\mathbf{W}_t$ associated with the function f_t at layer t simply as

$$\partial_{\mathbf{W}_t} \mathbf{o} = \underbrace{\partial_{\mathbf{h}^{d-1}} \mathbf{h}^d}_{:=\mathbf{M}_d} \cdot \ldots \cdot \underbrace{\partial_{\mathbf{h}^t} \mathbf{h}^{t+1}}_{:=\mathbf{M}_t} \underbrace{\partial_{\mathbf{W}_t} \mathbf{h}^t}_{:=\mathbf{v}_t}.$$

In other words, it is the product of $d-t$ matrices $\mathbf{M}_d \cdot \ldots \cdot \mathbf{M}_t$ and the gradient vector $\mathbf{v}_t$. What happens is quite similar to the situation when we experienced numerical underflow when multiplying too many probabilities. At the time we were able to mitigate the problem by switching from into log-space, i.e. by shifting the problem from the mantissa to the exponent of the numerical representation. Unfortunately the problem outlined in the equation above is much more serious: initially the matrices M_t may well have a wide variety of eigenvalues. They might be small, they might be large, and in particular, their product might well be *very large* or *very small*. This is not (only) a problem of numerical representation but it means that the optimization algorithm is bound to fail. It either receives gradients with excessively large

or excessively small steps. In the former case, the parameters explode and in the latter case we end up with vanishing gradients and no meaningful progress.

Exploding Gradients

To illustrate this a bit better, we draw 100 Gaussian random matrices and multiply them with some initial matrix. For the scaling that we picked, the matrix product explodes. If this were to happen to us with a deep network, we would have no meaningful chance of making the algorithm converge.

```
In [1]: %matplotlib inline
        import mxnet as mx
        from mxnet import nd, autograd
        from matplotlib import pyplot as plt

        M = nd.random.normal(shape=(4,4))
        print('A single matrix', M)
        for i in range(100):
            M = nd.dot(M, nd.random.normal(shape=(4,4)))

        print('After multiplying 100 matrices', M)
```

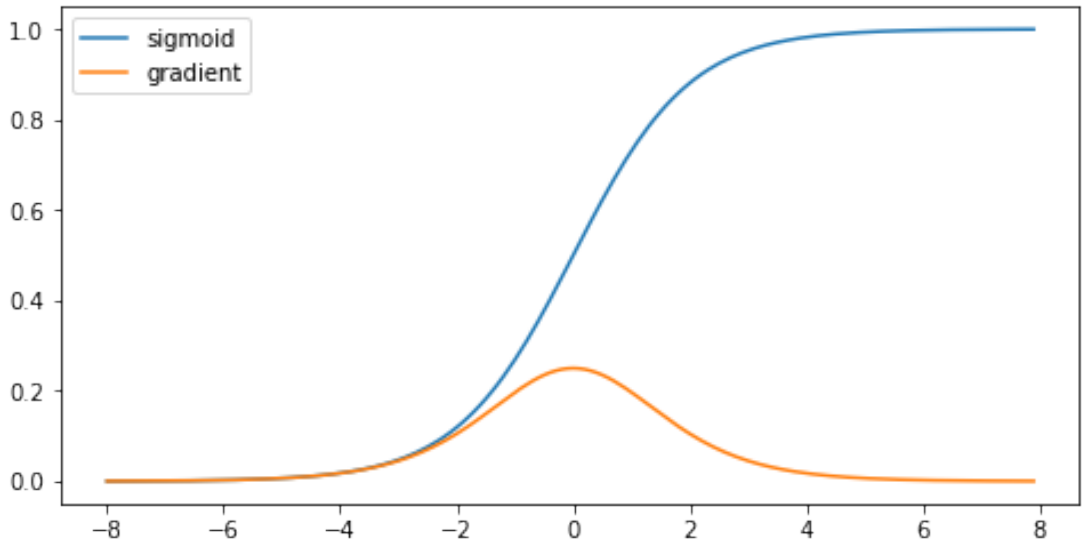
```
A single matrix
[[ 2.2122064   0.7740038   1.0434405   1.1839255 ]
 [ 1.8917114  -1.2347414  -1.771029   -0.45138445]
 [ 0.57938355 -1.856082   -1.9768796  -0.20801921]
 [ 0.2444218  -0.03716067 -0.48774993 -0.02261727]]
<NDArray 4x4 @cpu(0)>
After multiplying 100 matrices
[[ 3.1575275e+20 -5.0052276e+19  2.0565092e+21 -2.3741922e+20]
 [-4.6332600e+20  7.3445046e+19 -3.0176513e+21  3.4838066e+20]
 [-5.8487235e+20  9.2711797e+19 -3.8092853e+21  4.3977330e+20]
 [-6.2947415e+19  9.9783660e+18 -4.0997977e+20  4.7331174e+19]]
<NDArray 4x4 @cpu(0)>
```

Vanishing Gradients

The converse problem of vanishing gradients is just as bad. One of the major culprits in this context are the activation functions σ that are interleaved with the linear operations in each layer. Historically, a popular activation used to be the sigmoid function $(1 + \exp(-x))$ that was introduced in the section discussing Multilayer Perceptrons. Let us briefly review the function to see why picking it as nonlinear activation function might be problematic.

```
In [2]: x = nd.arange(-8.0, 8.0, 0.1)
        x.attach_grad()
        with autograd.record():
            y = x.sigmoid()
        y.backward()

        plt.figure(figsize=(8, 4))
        plt.plot(x.asnumpy(), y.asnumpy())
        plt.plot(x.asnumpy(), x.grad.asnumpy())
```

```
plt.legend(['sigmoid', 'gradient'])
plt.show()
```



As we can see, the gradient of the sigmoid vanishes for very large or very small arguments. Due to the chain rule, this means that unless we are in the Goldilocks zone where the activations are in the range of, say $[-4, 4]$, the gradients of the overall product may vanish. When we have many layers this is likely to happen for *some* layer. Before ReLu $\max(0, x)$ was proposed, this problem used to be the bane of deep network training. As a consequence ReLu has become the default choice when designing activation functions in deep networks.

Symmetry

A last problem in deep network design is the symmetry inherent in their parametrization. Assume that we have a deep network with one hidden layer with two units, say h_1 and h_2. In this case, we could flip the weights $\mathbf{W}_1$ of the first layer and likewise the outputs of the second layer and we would obtain the same function. More generally, we have permutation symmetry between the hidden units of each layer. This is more than just a theoretical nuisance. Assume that we initialize the parameters of some layer as $\mathbf{W}_l = 0$ or even just assume that all entries of $\mathbf{W}_l$ are identical. In this case the gradients for all dimensions are identical and we will never be able to use the expressive power inherent in a given layer. In fact, the hidden layer behaves as if it had only a single unit.

4.8.2 Parameter Initialization

One way of addressing, or at least mitigating the issues raised above is through careful initialization of the weight vectors. This way we can ensure that at least initially the gradients do not vanish and that they are

within a reasonable scale where the network weights do not diverge. Additional care during optimization and suitable regularization ensures that things never get too bad. Let's get started.

Default Initialization

In the previous sections, e.g. in Concise Implementation of Linear Regression, we used `net.initialize(init.Normal(sigma=0.01))` as a way to pick normally distributed random numbers as initial values for the weights. If the initialization method is not specified, such as `net.initialize()`, MXNet will use the default random initialization method: each element of the weight parameter is randomly sampled with an uniform distribution $U[-0.07, 0.07]$ and the bias parameters are all set to 0. Both choices tend to work quite well in practice for moderate problem sizes.

Xavier Initialization

Let's look at the scale distribution of the activations of the hidden units h_i for some layer. They are given by

$$h_i = \sum_{j=1}^{n_{\text{in}}} W_{ij} x_j$$

The weights W_{ij} are all drawn independently from the same distribution. Let's furthermore assume that this distribution has zero mean and variance σ^2 (this doesn't mean that the distribution has to be Gaussian, just that mean and variance need to exist). We don't really have much control over the inputs into the layer x_j but let's proceed with the somewhat unrealistic assumption that they also have zero mean and variance γ^2 and that they're independent of $\mathbf{W}$. In this case we can compute mean and variance of h_i as follows:

$$\begin{aligned}
\mathbf{E}[h_i] &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij} x_j] = 0 \\
\mathbf{E}[h_i^2] &= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2 x_j^2] \\
&= \sum_{j=1}^{n_{\text{in}}} \mathbf{E}[W_{ij}^2] \mathbf{E}[x_j^2] \\
&= n_{\text{in}} \sigma^2 \gamma^2
\end{aligned}$$

One way to keep the variance fixed is to set $n_{\text{in}} \sigma^2 = 1$. Now consider backpropagation. There we face a similar problem, albeit with gradients being propagated from the top layers. That is, instead of $\mathbf{W}\mathbf{w}$ we need to deal with $\mathbf{W}^\top \mathbf{g}$, where $\mathbf{g}$ is the incoming gradient from the layer above. Using the same reasoning as for forward propagation we see that the gradients' variance can blow up unless $n_{\text{out}} \sigma^2 = 1$. This leaves us in a dilemma: we cannot possibly satisfy both conditions simultaneously. Instead, we

simply try to satisfy

$$\frac{1}{2}(n_{\text{in}} + n_{\text{out}})\sigma^2 = 1 \text{ or equivalently } \sigma = \sqrt{\frac{2}{n_{\text{in}} + n_{\text{out}}}}$$

This is the reasoning underlying the eponymous Xavier initialization, proposed by Xavier Glorot and Yoshua Bengio in 2010. It works well enough in practice. For Gaussian random variables the Xavier initialization picks a normal distribution with zero mean and variance $\sigma^2 = 2/(n_{\text{in}} + n_{\text{out}})$. For uniformly distributed random variables $U[-a, a]$ note that their variance is given by $a^2/3$. Plugging $a^2/3$ into the condition on σ^2 yields that we should initialize uniformly with $U\left[-\sqrt{6/(n_{\text{in}} + n_{\text{out}})}, \sqrt{6/(n_{\text{in}} + n_{\text{out}})}\right]$.

Beyond

The reasoning above barely scratches the surface. In fact, MXNet has an entire `mxnet.initializer` module with over a dozen different heuristics. They can be used, e.g. when parameters are tied (i.e. when parameters of in different parts the network are shared), for superresolution, sequence models, and related problems. We recommend the reader to review what is offered as part of this module.

Summary

- Vanishing and exploding gradients are common issues in very deep networks, unless great care is taking to ensure that gradients and parameters remain well controlled.
- Initialization heuristics are needed to ensure that at least the initial gradients are neither too large nor too small.
- The ReLu addresses one of the vanishing gradient problems, namely that gradients vanish for very large inputs. This can accelerate convergence significantly.
- Random initialization is key to ensure that symmetry is broken before optimization.

Exercises

1. Can you design other cases of symmetry breaking besides the permutation symmetry?
2. Can we initialize all weight parameters in linear regression or in softmax regression to the same value?
3. Look up analytic bounds on the eigenvalues of the product of two matrices. What does this tell you about ensuring that gradients are well conditioned?
4. If we know that some terms diverge, can we fix this after the fact? Look at the paper on LARS by You, Gitman and Ginsburg, 2017 for inspiration.

most common use cases for convolutional networks. That is, we will discuss the two-dimensional case (image height and width). We begin with the cross-correlation operator that we introduced in the previous section. Strictly speaking, convolutional networks are a slight misnomer (but for notation only), since the operations are typically expressed as cross correlations.

6.2.1 The Cross-Correlation Operator

In a convolutional layer, an input array and a correlation kernel array output an array through a cross-correlation operation. Let's see how this works for two dimensions. As shown below, the input is a two-dimensional array with a height of 3 and width of 3. We mark the shape of the array as 3×3 or (3, 3). The height and width of the kernel array are both 2. This array is also called a kernel or filter in convolution computations. The shape of the kernel window (also known as the convolution window) depends on the height and width of the kernel, which is 2×2.

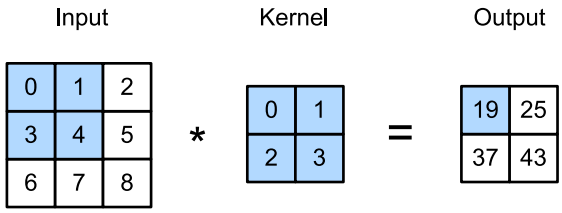


Fig. 6.1: Two-dimensional cross-correlation operation. The shaded portions are the first output element and the input and kernel array elements used in its computation: $0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19$.

In the two-dimensional cross-correlation operation, the convolution window starts from the top-left of the input array, and slides in the input array from left to right and top to bottom. When the convolution window slides to a certain position, the input subarray in the window and kernel array are multiplied and summed by element to get the element at the corresponding location in the output array. The output array has a height of 2 and width of 2, and the four elements are derived from a two-dimensional cross-correlation operation:

$$
\begin{aligned}
0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3 = 19,\\
1 \times 0 + 2 \times 1 + 4 \times 2 + 5 \times 3 = 25,\\
3 \times 0 + 4 \times 1 + 6 \times 2 + 7 \times 3 = 37,\\
4 \times 0 + 5 \times 1 + 7 \times 2 + 8 \times 3 = 43.
\end{aligned}
$$

Note that the output size is *smaller* than the input. In particular, the output size is given by the input size $H \times W$ minus the size of the convolutional kernel $h \times w$ via $(H - h + 1) \times (W - w + 1)$. This is the case since we need enough space to shift' the convolutional kernel across the image (later we will see how to keep the size unchanged by padding the image with zeros around its boundary such that there's enough space to shift the kernel). Next, we implement the above process in the `corr2d` function. It accepts the input array `X` with the kernel array `K` and outputs the array `Y`.

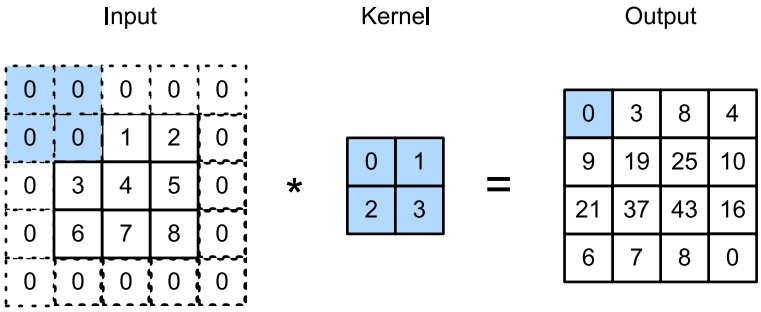


Fig. 6.2: Two-dimensional cross-correlation with padding. The shaded portions are the input and kernel array elements used by the first output element: $0 \times 0 + 0 \times 1 + 0 \times 2 + 0 \times 3 = 0$.

In general, if a total of p_h rows are padded on both sides of the height and a total of p_w columns are padded on both sides of width, the output shape will be

$$(n_h - k_h + p_h + 1) \times (n_w - k_w + p_w + 1),$$

This means that the height and width of the output will increase by p_h and p_w respectively.

In many cases, we will want to set $p_h = k_h - 1$ and $p_w = k_w - 1$ to give the input and output the same height and width. This will make it easier to predict the output shape of each layer when constructing the network. Assuming that k_h is odd here, we will pad $p_h/2$ rows on both sides of the height. If k_h is even, one possibility is to pad $\lceil p_h/2 \rceil$ rows on the top of the input and $\lfloor p_h/2 \rfloor$ rows on the bottom. We will pad both sides of the width in the same way.

Convolutional neural networks often use convolution kernels with odd height and width values, such as 1, 3, 5, and 7, so the number of padding rows or columns on both sides are the same. For any two-dimensional array `X`, assume that the element in its `i`th row and `j`th column is `X[i,j]`. When the number of padding rows or columns on both sides are the same so that the input and output have the same height and width, we know that the output `Y[i,j]` is calculated by cross-correlation of the input and convolution kernel with the window centered on `X[i,j]`.

In the following example we create a two-dimensional convolutional layer with a height and width of 3, and then assume that the padding number on both sides of the input height and width is 1. Given an input with a height and width of 8, we find that the height and width of the output is also 8.

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

        # We define a convenience function to calculate the convolutional layer. This
        # function initializes the convolutional layer weights and performs
        # corresponding dimensionality elevations and reductions on the input and
        # output
        def comp_conv2d(conv2d, X):
            conv2d.initialize()
            # (1,1) indicates that the batch size and the number of channels
```

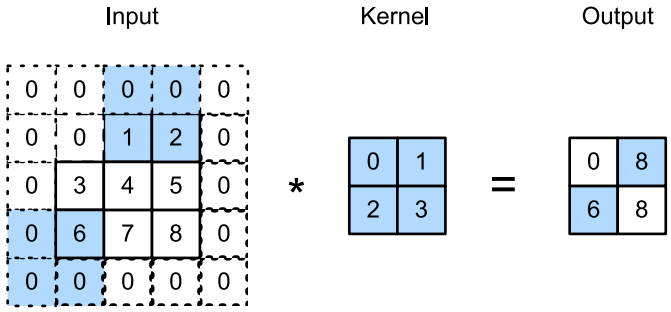


Fig. 6.3: Cross-correlation with strides of 3 and 2 for height and width respectively. The shaded portions are the output element and the input and core array elements used in its computation: $0 \times 0 + 0 \times 1 + 1 \times 2 + 2 \times 3 = 8$, $0 \times 0 + 6 \times 1 + 0 \times 2 + 0 \times 3 = 6$.

In general, when the stride for the height is s_h and the stride for the width is s_w, the output shape is

$$\lfloor (n_h - k_h + p_h + s_h)/s_h \rfloor \times \lfloor (n_w - k_w + p_w + s_w)/s_w \rfloor.$$

If we set $p_h = k_h - 1$ and $p_w = k_w - 1$, then the output shape will be simplified to $\lfloor (n_h + s_h - 1)/s_h \rfloor \times \lfloor (n_w + s_w - 1)/s_w \rfloor$. Going a step further, if the input height and width are divisible by the strides on the height and width, then the output shape will be $(n_h/s_h) \times (n_w/s_w)$.

Below, we set the strides on both the height and width to 2, thus halving the input height and width.

```
In [3]: conv2d = nn.Conv2D(1, kernel_size=3, padding=1, strides=2)
        comp_conv2d(conv2d, X).shape
```

```
Out[3]: (4, 4)
```

Next, we will look at a slightly more complicated example.

```
In [4]: conv2d = nn.Conv2D(1, kernel_size=(3, 5), padding=(0, 1), strides=(3, 4))
        comp_conv2d(conv2d, X).shape
```

```
Out[4]: (2, 2)
```

For the sake of brevity, when the padding number on both sides of the input height and width are p_h and p_w respectively, we call the padding (p_h, p_w). Specifically, when $p_h = p_w = p$, the padding is p. When the strides on the height and width are s_h and s_w, respectively, we call the stride (s_h, s_w). Specifically, when $s_h = s_w = s$, the stride is s. By default, the padding is 0 and the stride is 1. In practice we rarely use inhomogeneous strides or padding, i.e. we usually have $p_h = p_w$ and $s_h = s_w$.

Summary

- Padding can increase the height and width of the output. This is often used to give the output the same height and width as the input.

shape $c_i \times k_h \times k_w$. Since the input and convolution kernel each have c_i channels, we can perform a cross-correlation operation on the two-dimensional array of the input and the two-dimensional kernel array of the convolution kernel on each channel, and then add the c_i cross-correlated two-dimensional outputs by channel to get a two-dimensional array. This is the output of a two-dimensional cross-correlation operation between the multi-channel input data and the multi-input channel convolution kernel.

The figure below shows an example of a two-dimensional cross-correlation computation with two input channels. On each channel, the two-dimensional input array and the two-dimensional kernel array are cross-correlated, and then added together by channel to obtain the output. The shaded portions are the first output element as well as the input and kernel array elements used in its computation: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

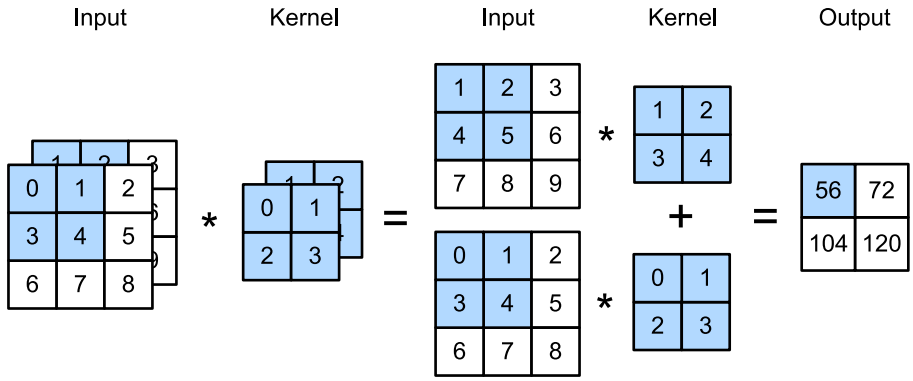


Fig. 6.4: Cross-correlation computation with 2 input channels. The shaded portions are the first output element as well as the input and kernel array elements used in its computation: $(1 \times 1 + 2 \times 2 + 4 \times 3 + 5 \times 4) + (0 \times 0 + 1 \times 1 + 3 \times 2 + 4 \times 3) = 56$.

Let's implement cross-correlation operations with multiple input channels. We simply need to perform a cross-correlation operation for each channel, and then add them up using the `add_n` function.

```
In [1]: import sys
        sys.path.insert(0, '..')

        import d2l
        from mxnet import nd

        def corr2d_multi_in(X, K):
            # First, traverse along the 0th dimension (channel dimension) of X and K.
            # Then, add them together by using * to turn the result list into a
            # positional argument of the add_n function
            return nd.add_n(*[d2l.corr2d(x, k) for x, k in zip(X, K)])
```

We can construct the input array `X` and the kernel array `K` of the above diagram to validate the output of the cross-correlation operation.

```
In [2]: X = nd.array([[[0, 1, 2], [3, 4, 5], [6, 7, 8]],
                      [[1, 2, 3], [4, 5, 6], [7, 8, 9]]])
```

6.4.3 1×1 Convolutional Layer

At first a 1×1 convolution, i.e. $k_h = k_w = 1$, doesn't seem to make much sense. After all, a convolution correlates adjacent pixels. A 1×1 convolution obviously doesn't. Nonetheless, it is a popular choice when designing complex and deep networks. Let's see in some detail what it actually does.

Because the minimum window is used, the 1×1 convolution loses the ability of the convolutional layer to recognize patterns composed of adjacent elements in the height and width dimensions. The main computation of the 1×1 convolution occurs on the channel dimension. The figure below shows the cross-correlation computation using the 1×1 convolution kernel with 3 input channels and 2 output channels. It is worth noting that the inputs and outputs have the same height and width. Each element in the output is derived from a linear combination of elements in the same position in the height and width of the input between different channels. Assuming that the channel dimension is considered a feature dimension and that the elements in the height and width dimensions are considered data examples, then the 1×1 convolutional layer is equivalent to the fully connected layer.

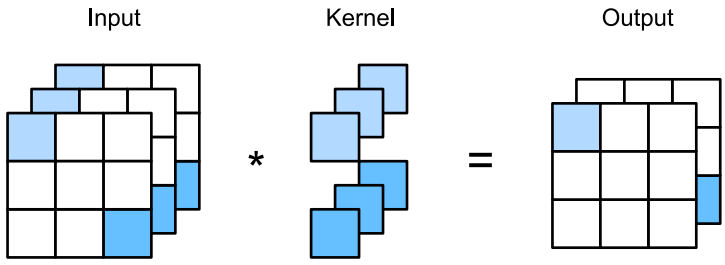


Fig. 6.5: The cross-correlation computation uses the 1×1 convolution kernel with 3 input channels and 2 output channels. The inputs and outputs have the same height and width.

Let's check whether this works in practice: we implement the 1×1 convolution using a fully connected layer. The only thing is that we need to make some adjustments to the data shape before and after the matrix multiplication.

```
In [6]: def corr2d_multi_in_out_1x1(X, K):
            c_i, h, w = X.shape
            c_o = K.shape[0]
            X = X.reshape((c_i, h * w))
            K = K.reshape((c_o, c_i))
            Y = nd.dot(K, X)  # Matrix multiplication in the fully connected layer
            return Y.reshape((c_o, h, w))
```

When performing 1×1 convolution, the above function is equivalent to the previously implemented cross-correlation function `corr2d_multi_in_out`. Let's check this with some reference data.

```
In [7]: X = nd.random.uniform(shape=(3, 3, 3))
        K = nd.random.uniform(shape=(2, 3, 1, 1))

        Y1 = corr2d_multi_in_out_1x1(X, K)
        Y2 = corr2d_multi_in_out(X, K)
```

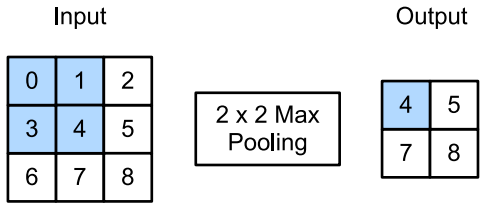


Fig. 6.6: Maximum pooling with a pooling window shape of 2×2. The shaded portions represent the first output element and the input element used for its computation: $\max(0, 1, 3, 4) = 4$

The output array in the figure above has a height of 2 and a width of 2. The four elements are derived from the maximum value of max:

$$\max(0, 1, 3, 4) = 4,$$
$$\max(1, 2, 4, 5) = 5,$$
$$\max(3, 4, 6, 7) = 7,$$
$$\max(4, 5, 7, 8) = 8.$$

Average pooling works like maximum pooling, only with the maximum operator replaced by the average operator. The pooling layer with a pooling window shape of $p \times q$ is called the $p \times q$ pooling layer. The pooling operation is called $p \times q$ pooling.

Let us return to the object edge detection example mentioned at the beginning of this section. Now we will use the output of the convolutional layer as the input for 2×2 maximum pooling. Set the convolutional layer input as `X` and the pooling layer output as `Y`. Whether or not the values of `X[i, j]` and `X[i, j+1]` are different, or `X[i, j+1]` and `X[i, j+2]` are different, the pooling layer outputs all include `Y[i, j]=1`. That is to say, using the 2×2 maximum pooling layer, we can still detect if the pattern recognized by the convolutional layer moves no more than one element in height and width.

As shown below, we implement the forward computation of the pooling layer in the `pool2d` function. This function is very similar to the `corr2d` function in the section on *convolutions*. The only difference lies in the computation of the output `Y`.

```
In [1]: from mxnet import nd
        from mxnet.gluon import nn

        def pool2d(X, pool_size, mode='max'):
            p_h, p_w = pool_size
            Y = nd.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
            for i in range(Y.shape[0]):
                for j in range(Y.shape[1]):
                    if mode == 'max':
                        Y[i, j] = X[i: i + p_h, j: j + p_w].max()
                    elif mode == 'avg':
                        Y[i, j] = X[i: i + p_h, j: j + p_w].mean()
            return Y
```

We can construct the input array `X` in the above diagram to validate the output of the two-dimensional

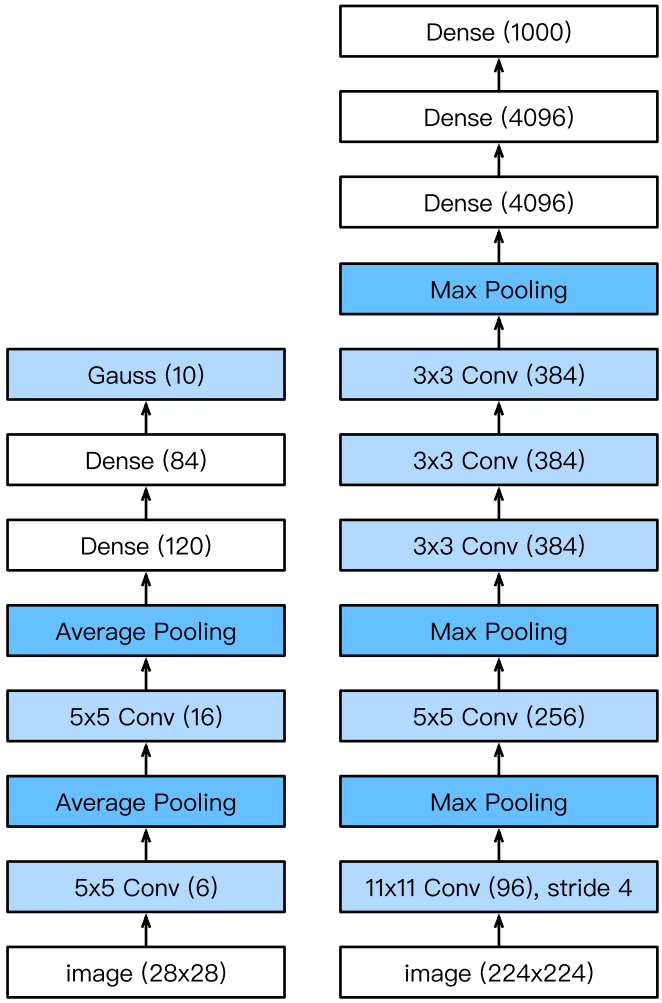


Fig. 6.10: LeNet (left) and AlexNet (right)

The design philosophies of AlexNet and LeNet are very similar, but there are also significant differences. First, AlexNet is much deeper than the comparatively small LeNet5. AlexNet consists of eight layers, five convolutional layers, two fully connected hidden layers, and one fully connected output layer. Second, AlexNet used the ReLu instead of the sigmoid as its activation function. This improved convergence during training significantly. Let's delve into the details below.

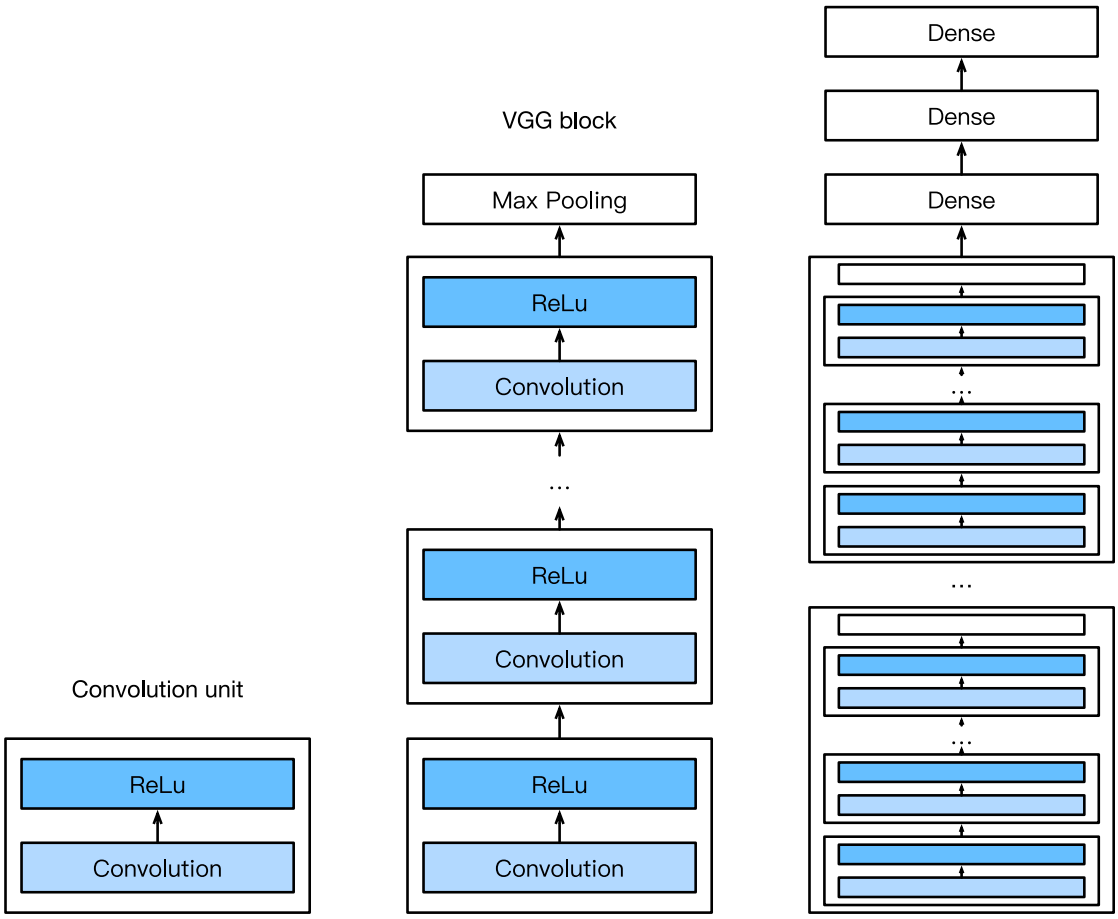


Fig. 6.11: Designing a network from building blocks

The VGG network proposed by Simonyan and Ziserman has 5 convolutional blocks, among which the former two use a single convolutional layer, while the latter three use a double convolutional layer. The first block has 64 output channels, and the latter blocks double the number of output channels, until that number reaches 512. Since this network uses 8 convolutional layers and 3 fully connected layers, it is often called VGG-11.

```
In [2]: conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
```

Now, we will implement VGG-11. This is a simple matter of executing a for loop over `conv_arch`.

```
In [3]: def vgg(conv_arch):
            net = nn.Sequential()
            # The convolutional layer part
            for (num_convs, num_channels) in conv_arch:
                net.add(vgg_block(num_convs, num_channels))
```

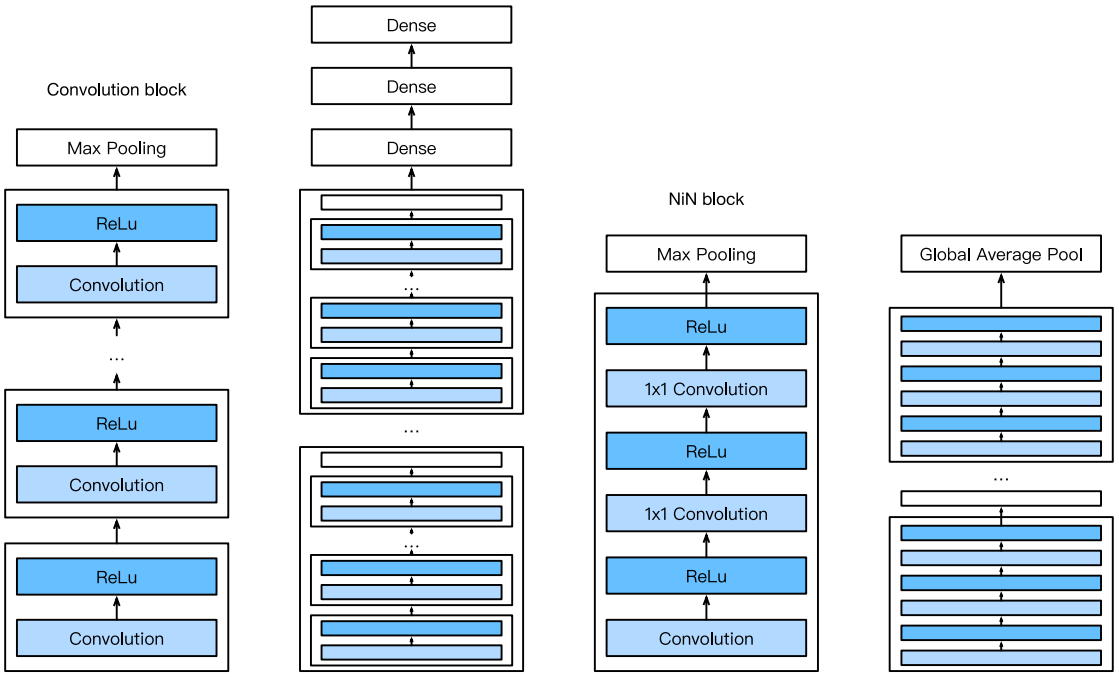


Fig. 6.12: The figure on the left shows the network structure of AlexNet and VGG, and the figure on the right shows the network structure of NiN.

The NiN block is the basic block in NiN. It concatenates a convolutional layer and two 1×1 convolutional layers that act as fully connected layers (with ReLu in between). The convolution width of the first layer is typically set by the user. The subsequent widths are fixed to 1×1.

```
In [1]: import sys
        sys.path.insert(0, '..')

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

        def nin_block(num_channels, kernel_size, strides, padding):
            blk = nn.Sequential()
            blk.add(nn.Conv2D(num_channels, kernel_size, strides, padding,
 ↪  activation='relu'),
                    nn.Conv2D(num_channels, kernel_size=1, activation='relu'),
                    nn.Conv2D(num_channels, kernel_size=1, activation='relu'))
            return blk
```

Scan the QR Code to Discuss

6.10 Networks with Parallel Concatenations (GoogLeNet)

During the ImageNet Challenge in 2014, a new architecture emerged that outperformed the rest. Szegedy et al., 2014 proposed a structure that combined the strengths of the NiN and repeated blocks paradigms. At its heart was the rather pragmatic answer to the question as to which size of convolution is ideal for processing. After all, we have a smorgasbord of choices, 1×1 or 3×3, 5×5 or even larger. And it isn't always clear which one is the best. As it turns out, the answer is that a combination of all the above works best. Over the next few years, researchers made several improvements to GoogLeNet. In this section, we will introduce the first version of this model series in a slightly simplified form - we omit the peculiarities that were added to stabilize training, due to the availability of better training algorithms.

6.10.1 Inception Blocks

The basic convolutional block in GoogLeNet is called an Inception block, named after the movie of the same name. This basic block is more complex in structure than the NiN block described in the previous section.

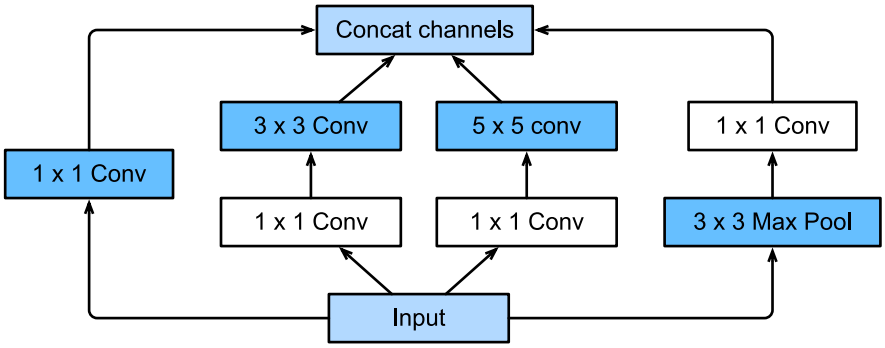


Fig. 6.13: Structure of the Inception block.

As can be seen in the figure above, there are four parallel paths in the Inception block. The first three paths use convolutional layers with window sizes of 1×1, 3×3, and 5×5 to extract information from different

6.10.2 GoogLeNet Model

GoogLeNet uses an initial long range feature convolution, a stack of a total of 9 inception blocks and global average pooling to generate its estimates. Maximum pooling between inception blocks reduced the dimensionality. The first part is identical to AlexNet and LeNet, the stack of blocks is inherited from VGG and the global average pooling that avoids a stack of fully connected layers at the end. The architecture is depicted below.

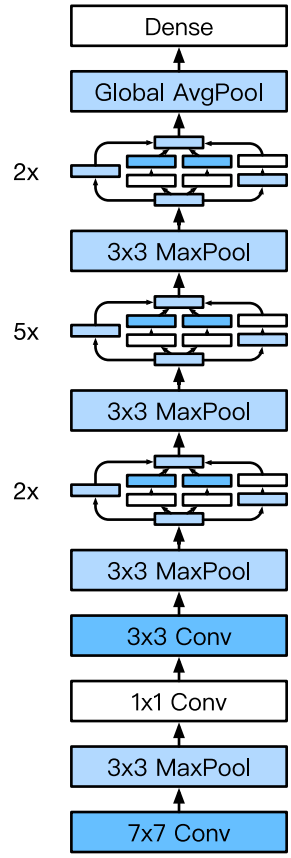


Fig. 6.14: Full GoogLeNet Model

Let's build the network piece by piece. The first block uses a 64-channel 7Œ7 convolutional layer.

```
In [2]: b1 = nn.Sequential()
        b1.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3, activation='relu'),
               nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

The second block uses two convolutional layers: first, a 64-channel 1×1 convolutional layer, then a 3×3 convolutional layer that triples the number of channels. This corresponds to the second path in the

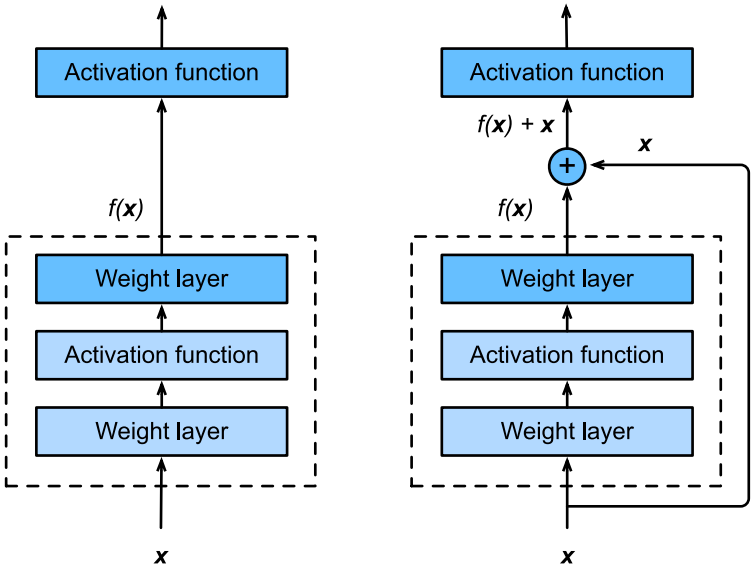


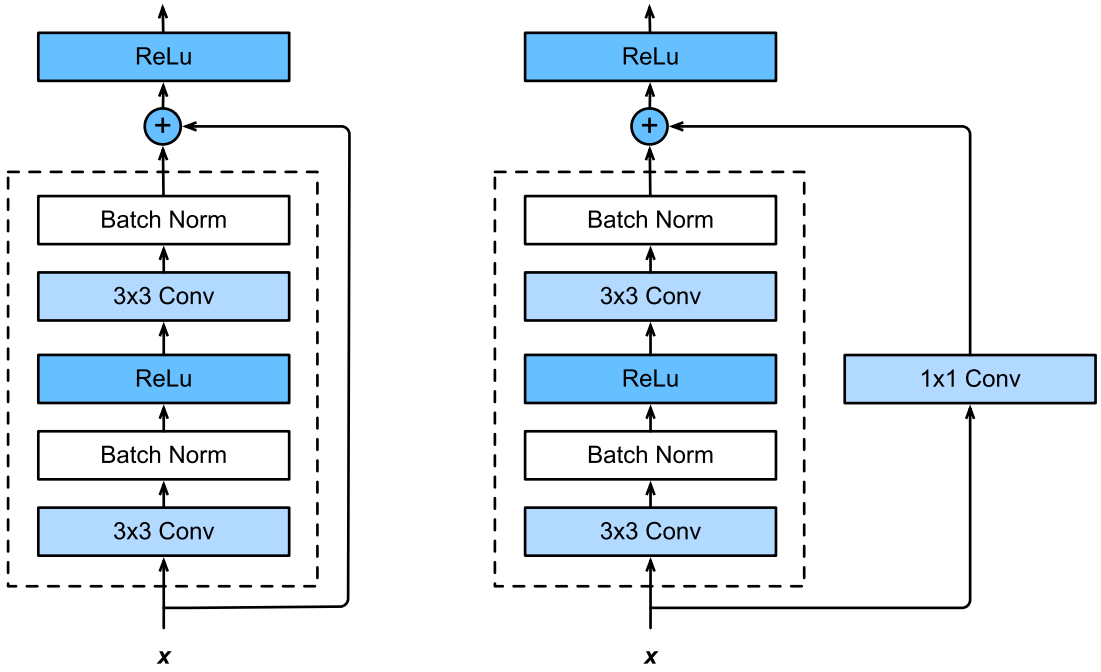
Fig. 6.16: The difference between a regular block (left) and a residual block (right). In the latter case, we can short-circuit the convolutions.

ResNet follows VGG's full 3×3 convolutional layer design. The residual block has two 3×3 convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers be of the same shape as the input, so that they can be added together. If we want to change the number of channels or the the stride, we need to introduce an additional 1×1 convolutional layer to transform the input into the desired shape for the addition operation. Let's have a look at the code below.

```
In [1]: import sys
        sys.path.insert(0, '..')

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

        # This class has been saved in the d2l package for future use
        class Residual(nn.Block):
            def __init__(self, num_channels, use_1x1conv=False, strides=1, **kwargs):
                super(Residual, self).__init__(**kwargs)
                self.conv1 = nn.Conv2D(num_channels, kernel_size=3, padding=1,
                                       strides=strides)
                self.conv2 = nn.Conv2D(num_channels, kernel_size=3, padding=1)
                if use_1x1conv:
                    self.conv3 = nn.Conv2D(num_channels, kernel_size=1,
                                           strides=strides)
```

```
        else:
            self.conv3 = None
        self.bn1 = nn.BatchNorm()
        self.bn2 = nn.BatchNorm()

    def forward(self, X):
        Y = nd.relu(self.bn1(self.conv1(X)))
        Y = self.bn2(self.conv2(Y))
        if self.conv3:
            X = self.conv3(X)
        return nd.relu(Y + X)
```

This code generates two types of networks: one where we add the input to the output before applying the ReLu nonlinearity, and whenever `use_1x1conv=True`, one where we adjust channels and resolution by means of a 1×1 convolution before adding. The diagram below illustrates this:



Now let us look at a situation where the input and output are of the same shape.

```
In [2]: blk = Residual(3)
        blk.initialize()
        X = nd.random.uniform(shape=(4, 3, 6, 6))
        blk(X).shape
```

```
Out[2]: (4, 3, 6, 6)
```

We also have the option to halve the output height and width while increasing the number of output channels.

```
In [3]: blk = Residual(6, use_1x1conv=True, strides=2)
```

```
        blk.initialize()
        blk(X).shape
```

```
Out[3]: (4, 6, 3, 3)
```

6.12.3 ResNet Model

The first two layers of ResNet are the same as those of the GoogLeNet we described before: the 7×7 convolutional layer with 64 output channels and a stride of 2 is followed by the 3×3 maximum pooling layer with a stride of 2. The difference is the batch normalization layer added after each convolutional layer in ResNet.

```
In [4]: net = nn.Sequential()
        net.add(nn.Conv2D(64, kernel_size=7, strides=2, padding=3),
                nn.BatchNorm(), nn.Activation('relu'),
                nn.MaxPool2D(pool_size=3, strides=2, padding=1))
```

GoogLeNet uses four blocks made up of Inception blocks. However, ResNet uses four modules made up of residual blocks, each of which uses several residual blocks with the same number of output channels. The number of channels in the first module is the same as the number of input channels. Since a maximum pooling layer with a stride of 2 has already been used, it is not necessary to reduce the height and width. In the first residual block for each of the subsequent modules, the number of channels is doubled compared with that of the previous module, and the height and width are halved.

Now, we implement this module. Note that special processing has been performed on the first module.

```
In [5]: def resnet_block(num_channels, num_residuals, first_block=False):
            blk = nn.Sequential()
            for i in range(num_residuals):
                if i == 0 and not first_block:
                    blk.add(Residual(num_channels, use_1x1conv=True, strides=2))
                else:
                    blk.add(Residual(num_channels))
            return blk
```

Then, we add all the residual blocks to ResNet. Here, two residual blocks are used for each module.

```
In [6]: net.add(resnet_block(64, 2, first_block=True),
                resnet_block(128, 2),
                resnet_block(256, 2),
                resnet_block(512, 2))
```

Finally, just like GoogLeNet, we add a global average pooling layer, followed by the fully connected layer output.

```
In [7]: net.add(nn.GlobalAvgPool2D(), nn.Dense(10))
```

There are 4 convolutional layers in each module (excluding the 1×1 convolutional layer). Together with the first convolutional layer and the final fully connected layer, there are 18 layers in total. Therefore, this model is commonly known as ResNet-18. By configuring different numbers of channels and residual blocks in the module, we can create different ResNet models, such as the deeper 152-layer ResNet-152. Although the main architecture of ResNet is similar to that of GoogLeNet, ResNet's structure is simpler

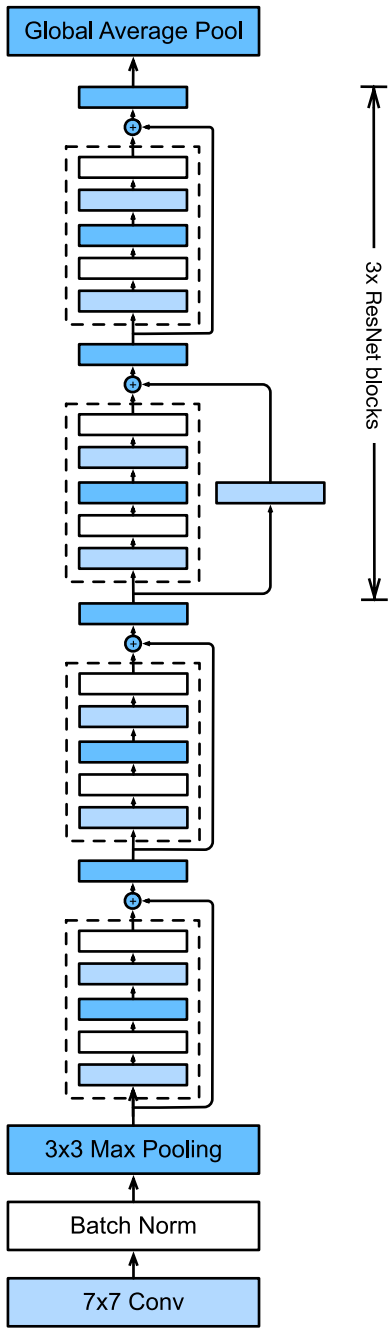


Fig. 6.17: ResNet 18

Before training ResNet, let us observe how the input shape changes between different modules in ResNet. As in all previous architectures, the resolution decreases while the number of channels increases up until the point where a global average pooling layer aggregates all features.

```
In [8]: X = nd.random.uniform(shape=(1, 1, 224, 224))
        net.initialize()
        for layer in net:
            X = layer(X)
            print(layer.name, 'output shape:\t', X.shape)
```

```
conv5 output shape:      (1, 64, 112, 112)
batchnorm4 output shape:         (1, 64, 112, 112)
relu0 output shape:      (1, 64, 112, 112)
pool0 output shape:      (1, 64, 56, 56)
sequential1 output shape:        (1, 64, 56, 56)
sequential2 output shape:        (1, 128, 28, 28)
sequential3 output shape:        (1, 256, 14, 14)
sequential4 output shape:        (1, 512, 7, 7)
pool1 output shape:      (1, 512, 1, 1)
dense0 output shape:     (1, 10)
```

6.12.4 Data Acquisition and Training

We train ResNet on the Fashion-MNIST data set, just like before. The only thing that has changed is the learning rate that decreased again, due to the more complex architecture.

```
In [9]: lr, num_epochs, batch_size, ctx = 0.05, 5, 256, d2l.try_gpu()
        net.initialize(force_reinit=True, 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, resize=96)
        d2l.train_ch5(net, train_iter, test_iter, batch_size, trainer, ctx,
                      num_epochs)
```

```
training on gpu(0)
epoch 1, loss 0.4950, train acc 0.826, test acc 0.875, time 61.4 sec
epoch 2, loss 0.2667, train acc 0.902, test acc 0.902, time 57.0 sec
epoch 3, loss 0.2035, train acc 0.926, test acc 0.894, time 56.9 sec
epoch 4, loss 0.1587, train acc 0.943, test acc 0.906, time 57.0 sec
epoch 5, loss 0.1205, train acc 0.957, test acc 0.904, time 57.0 sec
```

Summary

- Residual blocks allow for a parametrization relative to the identity function $f(\mathbf{x}) = \mathbf{x}$.
- Adding residual blocks increases the function complexity in a well-defined manner.
- We can train an effective deep neural network by having residual blocks pass through cross-layer data channels.
- ResNet had a major influence on the design of subsequent deep neural networks, both for convolutional and sequential nature.

6.13.1 Function Decomposition

The key point is that it decomposes the function into increasingly higher order terms. In a similar vein, ResNet decomposes functions into

$$f(\mathbf{x}) = \mathbf{x} + g(\mathbf{x})$$

That is, ResNet decomposes f into a simple linear term and a more complex nonlinear one. What if we want to go beyond two terms? A solution was proposed by Huang et al, 2016 in the form of DenseNet, an architecture that reported record performance on the ImageNet dataset.

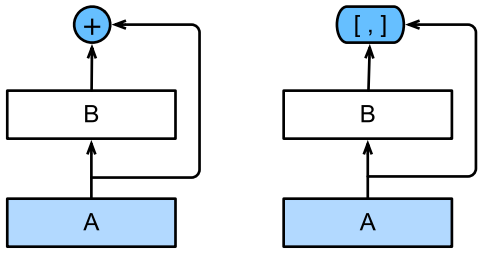


Fig. 6.18: The main difference between ResNet (left) and DenseNet (right) in cross-layer connections: use of addition and use of concatenation.

The key difference between ResNet and DenseNet is that in the latter case outputs are *concatenated* rather than added. As a result we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions.

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), f_3(\mathbf{x}, f_1(\mathbf{x}), f_2(\mathbf{x}, f_1(\mathbf{x})), \ldots]$$

In the end, all these functions are combined in an MLP to reduce the number of features again. In terms of implementation this is quite simple - rather than adding terms, we concatenate them. The name DenseNet arises from the fact that the dependency graph between variables becomes quite dense. The last layer of such a chain is densely connected to all previous layers. The main components that compose a DenseNet are dense blocks and transition layers. The former defines how the inputs and outputs are concatenated, while the latter controls the number of channels so that it is not too large.

layer name	output size	18-layer	34-layer	50-layer	101-layer	152-layer
conv1	112×112	7×7, 64, stride 2				
conv2_x	56×56	3×3 max pool, stride 2				
		$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 64 \\ 3\times3, 64 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 64 \\ 3\times3, 64 \\ 1\times1, 256 \end{bmatrix}\times3$
conv3_x	28×28	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 128 \\ 3\times3, 128 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times4$	$\begin{bmatrix} 1\times1, 128 \\ 3\times3, 128 \\ 1\times1, 512 \end{bmatrix}\times8$
conv4_x	14×14	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 256 \\ 3\times3, 256 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times6$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times23$	$\begin{bmatrix} 1\times1, 256 \\ 3\times3, 256 \\ 1\times1, 1024 \end{bmatrix}\times36$
conv5_x	7×7	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times2$	$\begin{bmatrix} 3\times3, 512 \\ 3\times3, 512 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$	$\begin{bmatrix} 1\times1, 512 \\ 3\times3, 512 \\ 1\times1, 2048 \end{bmatrix}\times3$
	1×1	average pool, 1000-d fc, softmax				
FLOPs		1.8×10^9	3.6×10^9	3.8×10^9	7.6×10^9	11.3×10^9