Chapter 11

Residual networks

The previous chapter described how image classification performance improved as the depth of convolutional networks was extended from eight layers (AlexNet) to eighteen layers (VGG). This led to experimentation with even deeper networks. However, performance decreased again when many more layers were added.

This chapter introduces *residual blocks*. Here, each network layer computes an additive change to the current representation instead of transforming it directly. This allows deeper networks to be trained but causes an exponential increase in the activation magnitudes at initialization. To compensate for this, residual blocks employ *batch normalization*, which resets the mean and variance of the activations at each layer.

Residual blocks with batch normalization allow much deeper networks to be trained, and these networks improve performance across a variety of tasks. Architectures that combine residual blocks to tackle image classification, medical image segmentation, and human pose estimation are described.

11.1 Sequential processing

Every network we have seen so far processes the data sequentially; each layer receives the output of the previous layer and passes the result to the next (figure 11.1). For example, a three-layer network is defined by:

$$\begin{aligned}\mathbf{h}_1 &= \mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1]\\ \mathbf{h}_2 &= \mathbf{f}_2[\mathbf{h}_1, \boldsymbol{\phi}_2]\\ \mathbf{h}_3 &= \mathbf{f}_3[\mathbf{h}_2, \boldsymbol{\phi}_3]\\ \mathbf{y} &= \mathbf{f}_4[\mathbf{h}_3, \boldsymbol{\phi}_4],\end{aligned} \tag{11.1}$$

where $\mathbf{h}_1$, $\mathbf{h}_2$, and $\mathbf{h}_3$ denote the intermediate hidden layers, $\mathbf{x}$ is the network input, $\mathbf{y}$ is the output, and the functions $\mathbf{f}_k[\bullet, \boldsymbol{\phi}_k]$ perform the processing.

In a standard neural network, each layer consists of a linear transformation followed by an activation function, and the parameters $\boldsymbol{\phi}_k$ comprise the weights and biases of the

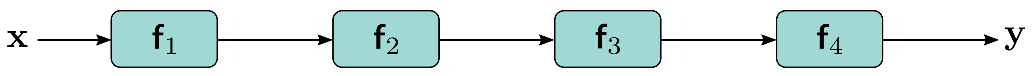

Figure 11.1 Sequential processing. Standard neural networks pass the output of each layer directly into the next layer.

linear transformation. In a convolutional network, each layer consists of a set of convolutions followed by an activation function, and the parameters comprise the convolutional kernels and biases.

Since the processing is sequential, we can equivalently think of this network as a series of nested functions:

$$\mathbf{y} = \mathbf{f}_4\left[\mathbf{f}_3\left[\mathbf{f}_2\left[\mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1], \boldsymbol{\phi}_2\right], \boldsymbol{\phi}_3\right], \boldsymbol{\phi}_4\right]. \tag{11.2}$$

11.1.1 Limitations of sequential processing

In principle, we can add as many layers as we want, and in the previous chapter we saw that adding more layers to a convolutional network does improve performance; the VGG network (figure 10.17), which has eighteen layers, outperforms AlexNet (figure 10.16), which has eight layers. However, as further layers are added, image classification performance decreases again (figure 11.2). This happens for the training set as well as the test set, which implies that the problem is training deeper networks, rather than the inability of deeper networks to generalize.

This phenomenon is not completely understood but one conjecture is that at initialization, the loss gradients change unpredictably when we modify parameters in an early network layer. With appropriate initialization, the gradient of the loss with respect to these parameters will be reasonable (i.e., no exploding or vanishing gradients). However, the derivative assumes an infinitesimal change in the parameter whereas optimization algorithms change the parameter by a finite step size. It may be that any reasonable choice of step size moves to a place with a completely different and unrelated gradient; the loss surface looks like an enormous range of tiny mountains, rather than a single large-scale structure that is easy to descend. Consequently, the algorithm does not make progress in the way that it does when the loss function gradient changes more slowly.

This conjecture is supported by empirical observations of the gradients in networks with a single input and a single output. For a shallow network, the gradient of the output with respect to the input changes relatively slowly as we change the input (figure 11.3a). However, for a deep network with 24 layers, a tiny change in the input results in a completely different gradient (figure 11.3b). This is captured by the autocorrelation function of the gradient (figure 11.3c). For shallow networks, the gradients stay highly correlated as the input changes, but for deep networks, their correlation quickly drops to zero. This is termed the *shattered gradients* phenomenon.

Appendix C.6
autocorrelation function

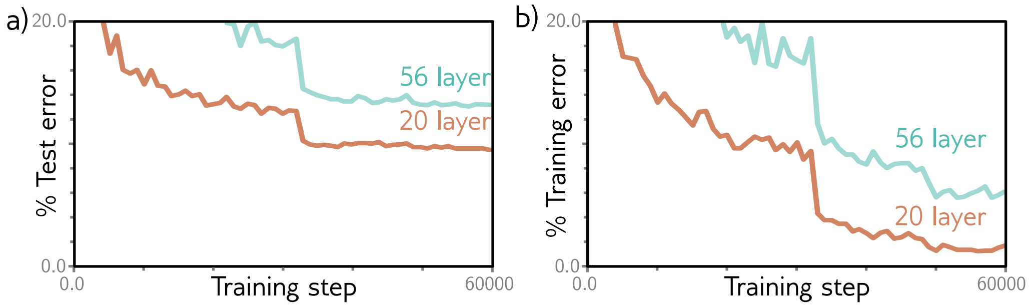

Figure 11.2 Decrease in performance when adding more convolutional layers. a) A 20-layer convolutional network outperforms a 56-layer neural network for image classification on the test set of the CIFAR-10 dataset (Krizhevsky & Hinton, 2009). b) This is also true for the training set, which suggests that the problem relates to training the original network, rather than a failure to generalize to new data. Adapted from He et al. (2016a).

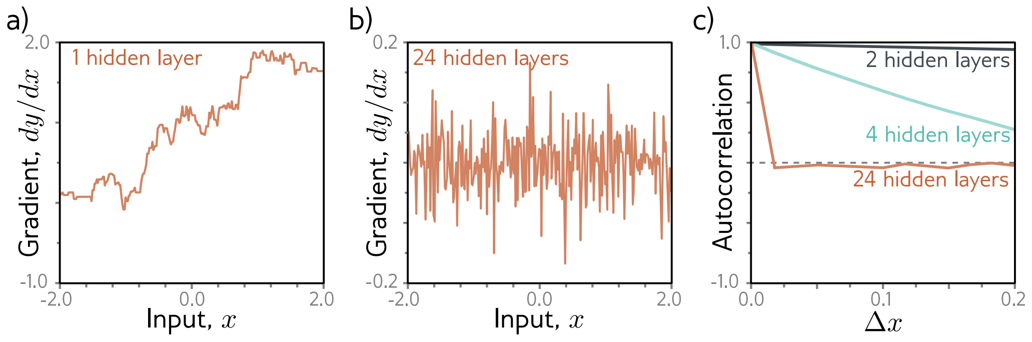

Figure 11.3 Shattered gradients. a) Consider a shallow network with 200 hidden units and Glorot initialization (He initialization without the factor of two) for both the weights and biases. The gradient $\partial y/\partial x$ of the scalar network output y with respect to the scalar input x changes relatively slowly as we change the input x. b) For a deep network with 24 layers and 200 hidden units per layer, this gradient changes very quickly and unpredictably. c) The autocorrelation function of the gradient shows that nearby gradients become unrelated (have autocorrelation close to zero) for deep networks. This *shattered gradients* phenomenon may explain why it is hard to train deep networks. Gradient descent algorithms rely on the loss surface being relatively smooth, so the gradients should be related before and after each update step. Adapted from Balduzzi et al. (2017).

Shattered gradients presumably arise because changes in early network layers modify the output in an increasingly complex way as the network becomes deeper. The derivative of the output $\mathbf{y}$ with respect to the first layer $\mathbf{f}_1$ of the network in equation 11.1 is:

$$\frac{\partial \mathbf{y}}{\partial \mathbf{f}_1} = \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}. \tag{11.3}$$

When we change the parameters that determine $\mathbf{f}_1$, *all* of the derivatives in this sequence can change, since layers $\mathbf{f}_2, \mathbf{f}_3$ and $\mathbf{f}_4$ are themselves computed from $\mathbf{f}_1$. Consequently, the updated gradient at each training example may be completely different, and the loss function becomes badly behaved.[1]

11.2 Residual connections and residual blocks

Residual or *skip connections* are branches in the computational path, whereby the input to each network layer $\mathbf{f}[\bullet]$ is added back to the output (figure 11.4a). By analogy to equation 11.1, the residual network is defined as:

$$\begin{aligned}
\mathbf{h}_1 &= \mathbf{x} + \mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1] \\
\mathbf{h}_2 &= \mathbf{h}_1 + \mathbf{f}_2[\mathbf{h}_1, \boldsymbol{\phi}_2] \\
\mathbf{h}_3 &= \mathbf{h}_2 + \mathbf{f}_3[\mathbf{h}_2, \boldsymbol{\phi}_3] \\
\mathbf{y} &= \mathbf{h}_3 + \mathbf{f}_4[\mathbf{h}_3, \boldsymbol{\phi}_4],
\end{aligned} \tag{11.4}$$

where the first term on the right-hand side of each line corresponds to the residual connection. Each function $\mathbf{f}_k$ learns an additive change to the current representation. It follows that their outputs must be the same size as their inputs. Each additive combination of the input and the processed output is known as a *residual block*.

Once more, we can write this as a single function by substituting in the expressions for the intermediate quantities $\mathbf{h}_k$:

Problem 11.1

$$\begin{aligned}
\mathbf{y} = \mathbf{x} \;+\; & \mathbf{f}_1[\mathbf{x}] \\
+\; & \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big] \\
+\; & \mathbf{f}_3\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big]\Big] \\
+\; & \mathbf{f}_4\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big] + \mathbf{f}_3\Big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}] + \mathbf{f}_2\big[\mathbf{x} + \mathbf{f}_1[\mathbf{x}]\big]\Big]\Big],
\end{aligned} \tag{11.5}$$

where we have omitted the parameters $\boldsymbol{\phi}_\bullet$ for clarity. We can think of this equation as "unraveling" the network (figure 11.4b). We see that the final network output is a sum of the input and four smaller networks, corresponding to each line of the equation; one

[1]In equations 11.3 and 11.6 we overload notation to define $\mathbf{f}_k$ as the output of the function $\mathbf{f}_k[\bullet]$.

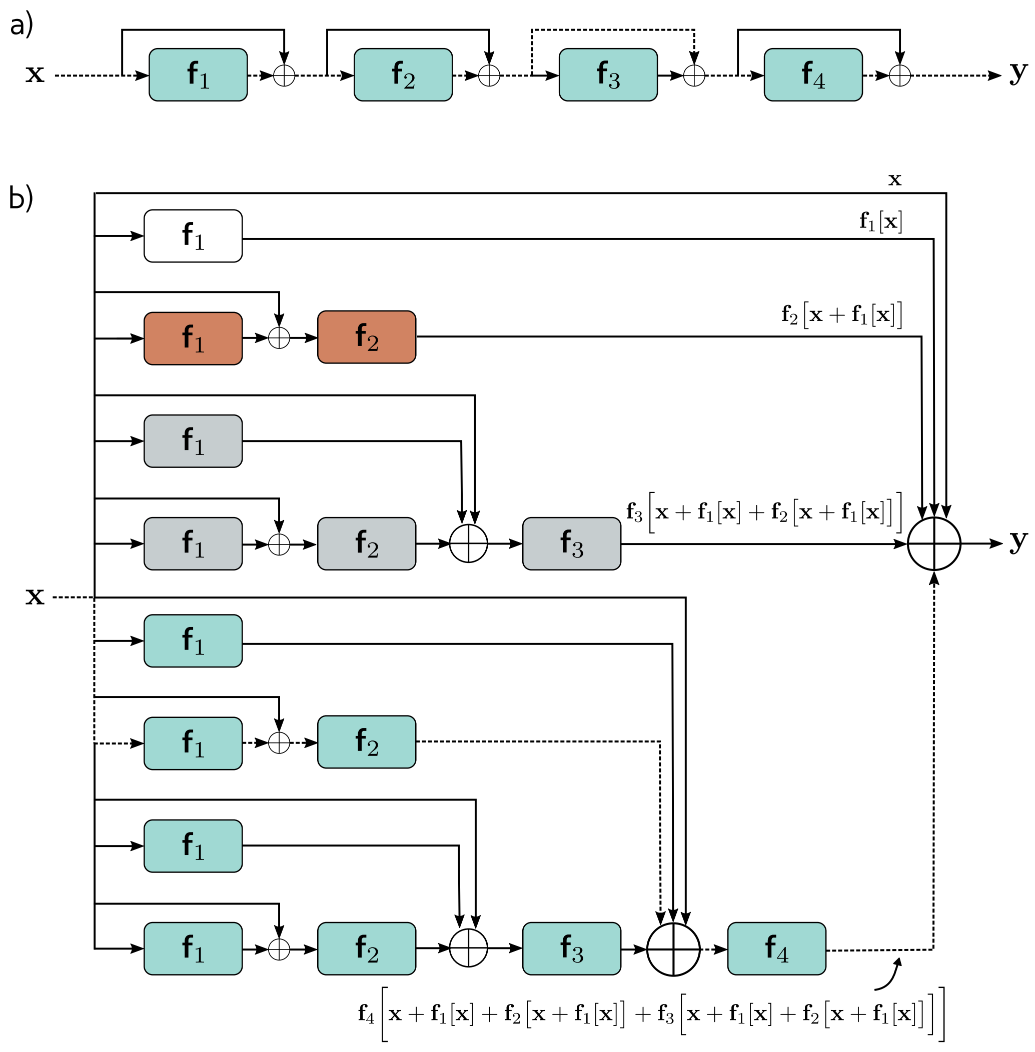
a)
x
$\mathbf{f}_1$
$\mathbf{f}_2$
$\mathbf{f}_3$
$\mathbf{f}_4$
y
b)
x
$\mathbf{f}_1[\mathbf{x}]$
$\mathbf{f}_2[\mathbf{x}+\mathbf{f}_1[\mathbf{x}]]$
$\mathbf{f}_3\left[\mathbf{x}+\mathbf{f}_1[\mathbf{x}]+\mathbf{f}_2[\mathbf{x}+\mathbf{f}_1[\mathbf{x}]]\right]$
y
$\mathbf{f}_4\left[\mathbf{x}+\mathbf{f}_1[\mathbf{x}]+\mathbf{f}_2[\mathbf{x}+\mathbf{f}_1[\mathbf{x}]]+\mathbf{f}_3\left[\mathbf{x}+\mathbf{f}_1[\mathbf{x}]+\mathbf{f}_2[\mathbf{x}+\mathbf{f}_1[\mathbf{x}]]\right]\right]$

Figure 11.4 Residual connections. a) The output of each function $\mathbf{f}_k[\mathbf{x}, \boldsymbol{\phi}_k]$ is added back to its input, which is passed via a parallel computational path called a residual or skip connection. Hence, the function computes an additive change to the representation. b) Upon expanding (unraveling) the network equations, we find that the output is the sum of the input plus four smaller networks (depicted in white, orange, gray, and cyan, respectively, and corresponding to terms in equation 11.5); we can think of this as an ensemble of networks. Moreover, the output from the cyan network is itself a transformation $\mathbf{f}_4[\bullet, \boldsymbol{\phi}_4]$ of another ensemble and so on. Alternatively, we can consider the network as a combination of 16 different paths through the computational graph. One example is the dashed path from input $\mathbf{x}$ to output $\mathbf{y}$ which is the same in both panels (a) and (b).

interpretation is that residual connections turn the original network into an ensemble of these smaller networks whose outputs are summed to compute the result.

A complementary way of thinking about this residual network is that it creates sixteen paths of different lengths from input to output. For example, the first function $\mathbf{f}_1[\mathbf{x}]$ occurs in eight of these sixteen paths, including as a direct additive term, (i.e., a path length of one) and the analogous derivative to equation 11.3 is:

Problem 11.2

Problem 11.3

$$\frac{\partial \mathbf{y}}{\partial \mathbf{f}_1} = \mathbf{I} + \frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} + \left(\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\right) + \left(\frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_1} + \frac{\partial \mathbf{f}_4}{\partial \mathbf{f}_3}\frac{\partial \mathbf{f}_3}{\partial \mathbf{f}_2}\frac{\partial \mathbf{f}_2}{\partial \mathbf{f}_1}\right), \tag{11.6}$$

where there is one term for each of the eight paths. The identity term on the right-hand side shows that changes in the parameters $\boldsymbol{\phi}_1$ in the first layer $\mathbf{f}_1[\mathbf{x}, \boldsymbol{\phi}_1]$ contribute directly to changes in the network output $\mathbf{y}$ as well as indirectly through other chains of derivatives. In general, gradients through the shorter paths will be better behaved.

Consequently, networks with residual links are less likely to suffer from problems due to long chains of derivatives. In addition, if a function $\mathbf{f}_k[\bullet]$ has a very small output (as might be encouraged by an L2 regularization term), then the input is largely unmodified. It follows that it is much easier to learn an identity mapping, and hence for the network to choose the effective depth during the training process.

11.2.1 Order of operations in residual blocks

Until this point, we have implied that the additive functions $\mathbf{f}[\mathbf{x}]$ could be any valid network layer (e.g., fully connected, or convolutional). This is technically true, but the order of operations in these functions is important. They must contain a nonlinear activation function like a ReLU or the entire network will be linear. However, in a typical network layer (figure 11.5a), the ReLU function is at the end, and so the output is non-negative. If we adopt this convention, then each residual block we will only be able to increase the input values.

Hence, it is typical to change the order of operations so that the activation function is applied first and is followed by the linear transformation (figure 11.5b). Sometimes there may be several layers of processing within the residual block (figure 11.5c) but these usually terminate with a linear transformation. Finally, we note that if we start each of these blocks with a ReLU operation, then they will do nothing if the initial network input is negative since the ReLU will clip the entire signal to zero. Hence, it's typical to start the network with a linear transformation.

11.2.2 Deeper networks with residual connections

As a rule of thumb, adding residual connections roughly doubles the depth of a network that can be practically trained before performance starts to degrade. However, we would like to be able to increase the depth further. To understand why residual connections

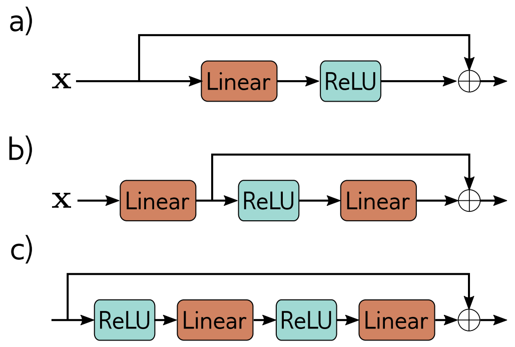

Figure 11.5 Order of operations in residual blocks. a) The usual order of linear transformation or convolution followed by a ReLU nonlinearity means that each residual block can only add non-negative quantities. b) With the reverse order, both positive and negative quantities can be added. However, we must add a linear transformation at the start of the network in case the input is all negative. c) In practice, it's common for a residual block to contain several network layers.

do not allow us to increase the depth arbitrarily, we must consider how the variance of the activations changes during the forward pass, and how the magnitude of the gradients changes during the backward pass.

11.3 Exploding gradients in residual networks

In section 7.5 we saw that initializing the network parameters is critical. Without careful initialization, the magnitudes of the intermediate values during the forward pass of backpropagation can increase or decrease exponentially. Similarly, the gradients during the backward pass can explode or vanish as we move backward through the network.

The standard approach is to initialize the network parameters so that the expected variance of the activations (in the forward pass) and gradients (in the backward pass) remains the same between layers. He initialization (section 7.5) achieves this for ReLU activations by initializing the biases $\boldsymbol{\beta}$ to zero and choosing normally distributed weights $\boldsymbol{\Omega}$ with mean zero and variance $2/D_h$ where D_h is the number of hidden units in the previous hidden layer (see figure 7.7).

Now consider a residual network. We do not have to worry about the intermediate values or gradients vanishing with network depth since there exists a path whereby each layer directly contributes to the network output (equation 11.5 and figure 11.4b). However, even if we use He initialization within the residual block, the values in the forward pass increase exponentially as we move through the network.

To see why, consider that we add the result of the processing in the residual block back to the input. Each of the two branches has some (independent) variability. Hence, the overall variance increases when we recombine them. With ReLU activations and He initialization, the expected variance is unchanged by the processing in each block. Consequently, when we recombine with the input, the variance doubles (figure 11.6a), and so grows exponentially with the number of residual blocks. This limits the possible network depth before floating point precision is exceeded in the forward pass. A similar argument applies to the gradients in the backward pass of the backpropagation algorithm.

Problem 11.4

Problem 11.5

Hence, residual networks still suffer from unstable forward propagation and exploding gradients even with He initialization. One approach that would stabilize the forward and

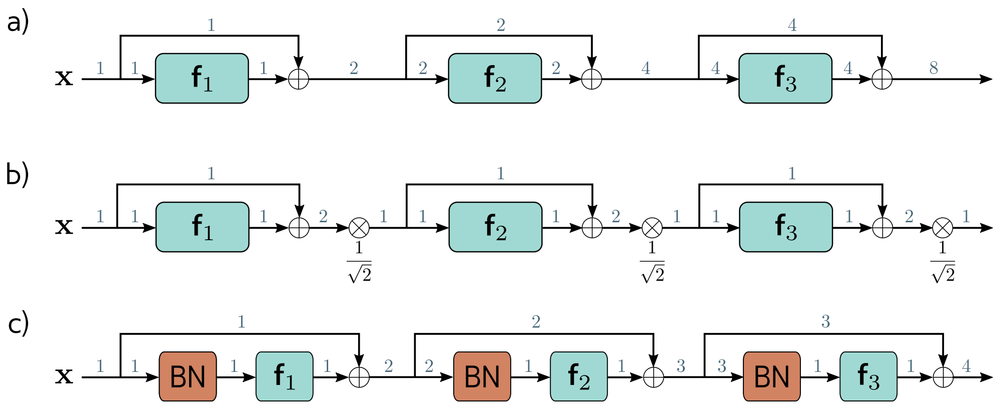

Figure 11.6 Variance in residual networks. a) He initialization ensures that the expected variance remains the same after a linear plus ReLU layer $\mathbf{f}_k$. Unfortunately, in residual networks, the output of each block is added back to the input, and so the variance doubles at each layer (blue numbers indicate variance) and grows exponentially. b) One approach would be to rescale the signal by $1/\sqrt{2}$ between each residual block. c) A second method is to use batch normalization (BN) as the first step in the residual block and set the associated offset δ to zero and scale γ to one. This transforms the input to each layer to have variance one, and with He initialization, the output variance will also be one. Now the variance increases linearly with the number of residual blocks. A side-effect is that at initialization, later layers of the network make a relatively smaller contribution.

backward passes would be to use He initialization and then multiply the combined output of each residual block by $1/\sqrt{2}$ to compensate for the doubling (figure 11.6b). However, it is more usual to use *batch normalization*.

Problem 11.6

11.4 Batch normalization

Batch normalization or *BatchNorm* shifts and rescales each activation h so that its mean and variance across the batch $\mathcal{B}$ become values which are learned during training. First, the empirical mean m_h and standard deviation s_h are computed:

$$\begin{aligned} m_h &= \frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}} h_i \\ s_h &= \sqrt{\frac{1}{|\mathcal{B}|}\sum_{i\in\mathcal{B}}(h_i - m_h)^2}. \end{aligned} \tag{11.7}$$

Then we use these quantities to normalize the batch activations to have mean zero and

Appendix B.2.3 standardization

variance one:

$$h_i \leftarrow \frac{h_i - m_h}{s_h + \epsilon} \qquad \forall i \in \mathcal{B}, \tag{11.8}$$

where ϵ is a small number that prevents division by zero if h_i is the same for every member of the batch and $s_h = 0$.

Finally, the normalized variable is scaled by γ and shifted by δ:

$$h_i \leftarrow \gamma h_i + \delta \qquad \forall i \in \mathcal{B}. \tag{11.9}$$

Problem 11.7

After this operation, the activations have mean δ and standard deviation γ across all members of the batch. Both of these quantities are parameters of the model and are learned during training.

Problem 11.8

Batch normalization is applied independently to each hidden unit, so in a standard neural network with K layers each containing D hidden units, there would be KD learned offsets δ and KD learned scales γ. In a convolutional network, the normalizing statistics are computed over both the batch and the spatial position, so if there were K layers each containing C channels, there would be KC offsets and KC scales. At test time, we do not have a batch from which we can gather statistics. To resolve this, the statistics m_h and σ_h are calculated across the whole training dataset (rather than just a batch) and frozen in the final network.

11.4.1 Costs and benefits of batch normalization

Batch normalization makes the network invariant to rescaling the weights and biases that contribute to each activation; if these are doubled, then the activations also double, the estimated standard deviation s_h doubles, and the normalization in equation 11.8 compensates for these changes. This happens separately for each hidden unit. Consequently, there will be a large family of weights and biases that all produce the same effect. Batch normalization also adds two parameters γ and δ at every hidden unit which makes the model somewhat larger. Hence, it both creates redundancy in the weight parameters and adds extra parameters to compensate for that redundancy. This is obviously inefficient, but batch normalization also provides several benefits:

Stable forward propagation: If we initialize the offsets δ to zero and the scales γ to one, then each output activation will have variance one. In a regular network, this ensures that the variance is stable during forward propagation at initialization. In a residual network, the variance must still increase as we are adding a new source of variation to the input at each layer. However, it will increase linearly with each residual block; the k^{th} layer adds one unit of variance to the existing variance of k (figure 11.6c).

This has the side-effect that later layers make a smaller change to the overall variation than earlier ones at initialization. Hence, the network is effectively less deep at the start of training, since later layers make negligible changes. As training proceeds, the network can increase the scales γ in the later layers, and so the network can easily control its own effective depth.

Higher learning rates: Empirical studies and theory both show that batch normalization makes the loss surface and its gradient change more smoothly (i.e., reduces shattered gradients). This means that we can use higher learning rates as the surface is more predictable. We saw in section 9.2 that higher learning rates can improve test performance.

Regularization: We also saw in chapter 9 that adding noise to the training process can make models improve generalization. Batch normalization injects noise because the normalization depends on the batch statistics. The activations for a given training example, are normalized by an amount that depends on the other members of the batch, and so will be slightly different at each training iteration.

11.5 Common residual architectures

Residual connections are now a standard part of deep learning pipelines. This section reviews some well-known architectures that incorporate them.

11.5.1 ResNet

Residual blocks were first used in convolutional networks for image classification. The resulting networks are known as residual networks or *ResNets* for short. In ResNets each residual block contains a batch normalization operation, then the ReLU activation function, and a convolutional layer. This is followed by the same sequence again before being added back to the input (figure 11.7a). Trial and error has shown that this order of operations works well for image classification.

Problem 11.9

For very deep networks, the number of parameters may become undesirably large. *Bottleneck residual blocks* make more efficient use of parameters using three convolutions. The first has a 1×1 kernel and reduces the number of channels. The second is a regular 3×3 kernel and the third is another 1×1 kernel to increase the number of channels back to the original amount (figure 11.7b). In this way, we can integrate information over a 3×3 pixel area using fewer parameters.

Problem 11.10

The ResNet-200 model (figure 11.8) contains 200 layers and is intended for image classification on the ImageNet database (figure 10.15). The architecture is similar to AlexNet and VGG but it uses bottleneck residual blocks instead of vanilla convolutional layers. As with AlexNet and VGG, these are periodically interspersed with decreases in spatial resolution and simultaneous increases in the number of channels. Here, the resolution is decreased by downsampling using convolutions with stride two, and the number of channels is increased either by appending zeros to the representation or using an extra 1×1 convolution. At the start of the network is a 7×7 convolutional layer, followed by a downsampling operation. At the end, there is a fully connected layer that maps the block to a vector of length 1000. This is then passed through a softmax layer to generate the class probabilities.

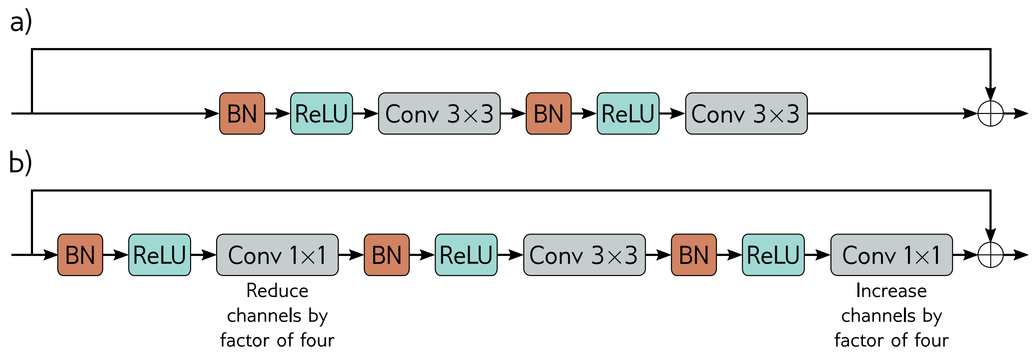

Figure 11.7 ResNet blocks. a) A standard block in the ResNet architecture contains a batch normalization operation, followed by an activation function, and then a 3×3 convolutional layer. Then, this sequence is repeated. b). A bottleneck ResNet block still integrates information over a 3×3 region but uses fewer parameters. It contains three convolutions. The first 1×1 convolution reduces the number of channels. The second 3×3 convolution is applied to the smaller representation. A final 1×1 convolution increases the number of channels again so that it can be added back to the input.

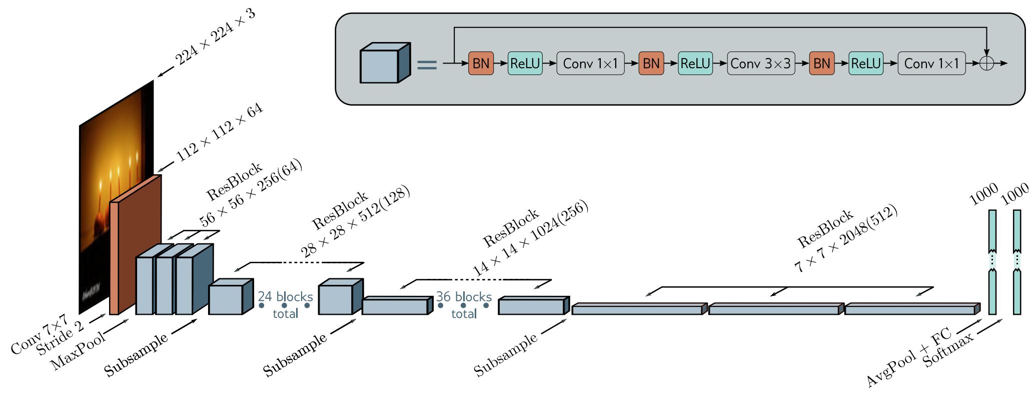

Figure 11.8 ResNet 200 model. A standard 7×7 convolutional layer with stride two is applied, followed by a MaxPool operation. A series of bottleneck residual blocks follow (number in brackets is channels after first 1×1 convolution), with periodic downsampling, and accompanying increases in the number of channels. The network concludes with average pooling across all spatial positions and a fully connected layer that maps to pre-softmax activations.

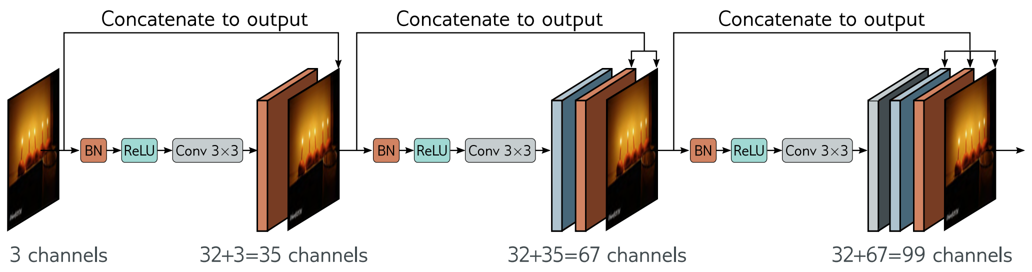

Figure 11.9 DenseNet. This architecture uses residual connections to concatenate the outputs of earlier layers to later ones. Here, the three-channel input image is processed to form a 32-channel representation. The input image is concatenated to this to give a total of 35 channels. This combined representation is processed to create another 32-channel representation and both earlier representations are concatenated to this to create a total of 67 channels and so on.

The ResNet-200 model achieved a remarkable 4.8% error rate for the correct class being in the top five and 20.1% for identifying the correct class correctly. This compares favorably with AlexNet (16.4%, 38.1%) and VGG (6.8%, 23.7%) and was one of the first networks to exceed human performance (5.1% for being in the top five guesses). However, this model was conceived in 2016 and is now far from state-of-the-art. At the time of writing, the best-performing model on this task has a 9.0% error for identifying the class correctly (see figure 10.22). This and all the other top-performing models for image classification are now based on transformers (see chapter 12).

11.5.2 DenseNet

Residual blocks receive the output from the previous layer, modify it, and add it back to the original input. An alternative is to concatenate the modified and original signals. This increases the representation size (in terms of channels for a convolutional network), but an optional subsequent linear transformation can map back to the original size (a 1×1 convolution for a convolutional network). This allows the model to add the representations together, take a weighted sum, or combine them in a more complex way.

The DenseNet architecture uses concatenation so that the input to a layer comprises the concatenated outputs from *all* previous layers (figure 11.9). These are processed to create a new representation that is itself concatenated with the previous representation and passed to the next layer. This concatenation means that there is a direct contribution from earlier layers to the output, and so the loss surface behaves reasonably.

Problem 11.11

In practice, this can only be sustained for a few layers because the number of channels (and hence the number of parameters required to process them) becomes increasingly large. This problem can be alleviated by applying a 1×1 convolution to reduce the number of channels before the next 3×3 convolution is applied. In a convolutional network, the input is periodically downsampled; concatenation of the representation across the downsampling makes no sense since the representations are of different sizes. Consequently, the chain of concatenation is broken at this point and a smaller representation starts a new chain. In addition, another bottleneck 1×1 convolution can be applied when the downsampling occurs to further control the representation size.

This network performs competitively with ResNet models on image classification benchmarks (see figure 10.22); indeed for a comparable parameter count, the DenseNet model can perform better. This is presumably because it has the option of reusing processing from earlier layers more flexibly.

11.5.3 U-Nets and hourglass networks

Section 10.5.3 described a network for semantic segmentation that had an encoder-decoder or hourglass structure. The encoder repeatedly downsamples the image until the receptive fields cover large areas and information is integrated from across the image. Then the decoder upsamples it again until it is the same size as the original image. The final output is a probability over possible object classes at each pixel. One drawback of this architecture is that the low-resolution representation in the middle of the network has to remember the high-resolution details of the original image to make the final result accurate. This is not necessary if residual connections add or concatenate the representations from the encoder to their partner in the decoder.

The *U-Net* (figure 11.10) is an encoder-decoder architecture where the earlier representations are concatenated to the later ones. It uses "valid" convolutions, so the spatial size of the representation decreases by two pixels each time a 3×3 convolutional layer is applied. This means that the upsampled version is smaller than its counterpart in the encoder, which must be cropped before concatenation. Note that the U-Net is completely convolutional, so after training it can be run on an image of *any size*.

Problem 11.12

The U-Net has been used extensively for segmenting medical images (figure 11.11), but this and similar architectures have also found use in computer vision. *Hourglass networks* are similar, but apply further convolutional layers in the skip connections and add the result back to the decoder side rather than concatenating it. Several of these networks can be stacked and trained together, creating a *stacked hourglass network* that alternates between considering the image at local and global levels. This network has been used for human pose estimation (figure 11.12). The system is trained to predict one "heatmap" for each joint, and the final joint position is estimated by taking the maximum of each heatmap.

11.6 Why do nets with residual connections perform so well?

Residual networks allow much deeper networks to be trained; in fact, it's possible to extend the ResNet architecture to 1000 layers and still train effectively. The improvement in image classification performance was initially attributed to the additional depth of the network but two pieces of evidence have emerged that contradict this viewpoint.

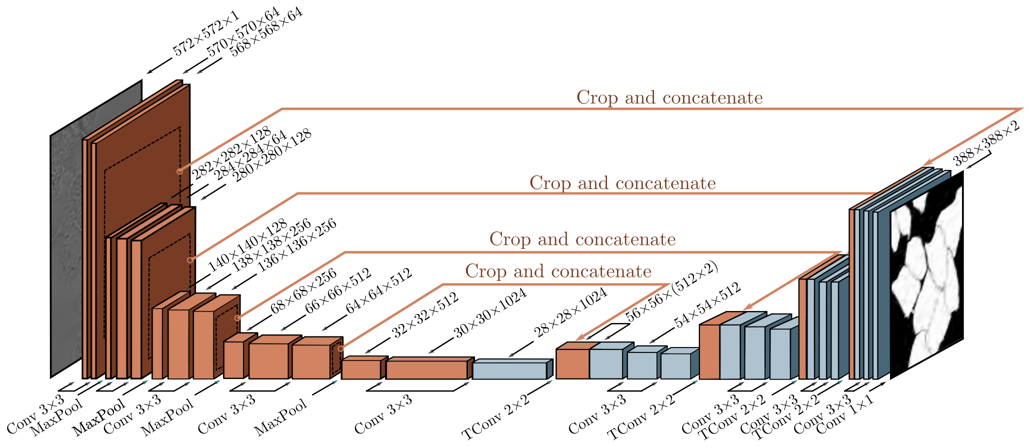

Figure 11.10 U-Net for segmenting HeLa cells. The network follows an encoder-decoder structure, in which the representation is downsampled (orange blocks) and then re-upsampled again (blue blocks). Residual connections append the last representation at each scale on the left-hand side to the first representation at the same scale on the right-hand side (orange arrows). The U-Net uses "valid" convolutions, so the size decreases slightly with each layer even without downsampling. This means that the representations from the left-hand side must be cropped (dashed squares) before appending to the right-hand side. Adapted from Ronneberger et al. (2015).

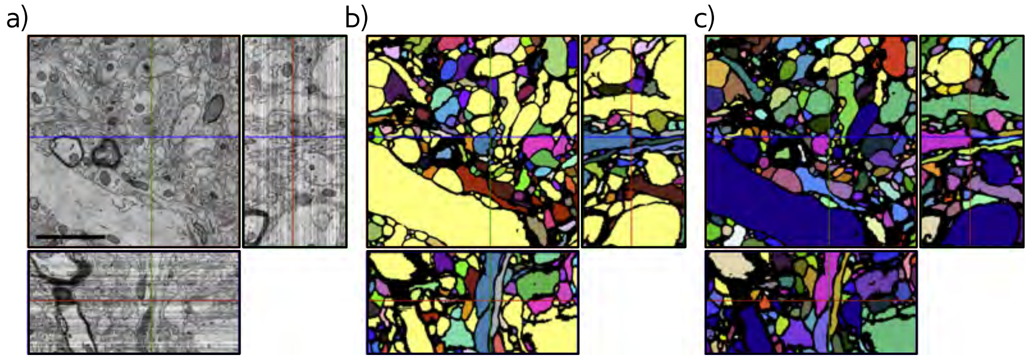

Figure 11.11 Segmentation using U-Net in 3D. a) Three slices through a 3D volume of mouse cortex taken by scanning electron microscope. b) A single U-Net is used to classify voxels as being inside or outside neurites. Connected regions are identified with different colors. c) For a better result, an ensemble of five U-Nets is trained and a voxel is only classified as belonging to the cell if all five networks agree. Adapted from Falk et al. (2019).

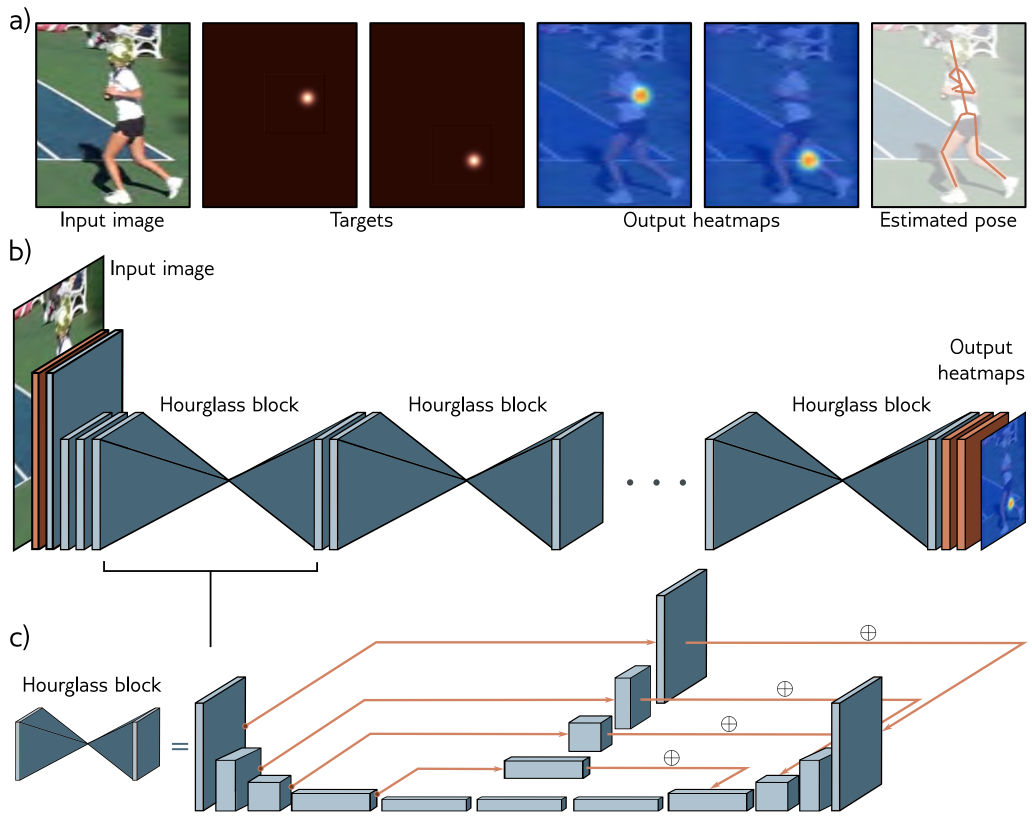

Figure 11.12 Stacked hourglass networks for pose estimation. a) The network input is an image containing a person and the output is a set of heatmaps, where there is one heatmap for each joint. This is formulated as a regression problem where the targets are images with small highlighted regions at the ground-truth joint positions. Each joint position can be extracted by taking the peak of the estimated heatmap and these can be combined to reconstruct the 2D pose. b) The architecture consists of initial convolutional and residual layers followed by a series of hourglass blocks. c) Each hourglass block consists of an encoder-decoder network similar to the U-Net except that the convolutions use zero-padding, some further processing is done in the residual links, and these links add this processed representation rather than concatenate it. Each blue cuboid is itself a bottleneck residual block (figure 11.7b). Adapted from Newell et al. (2016).

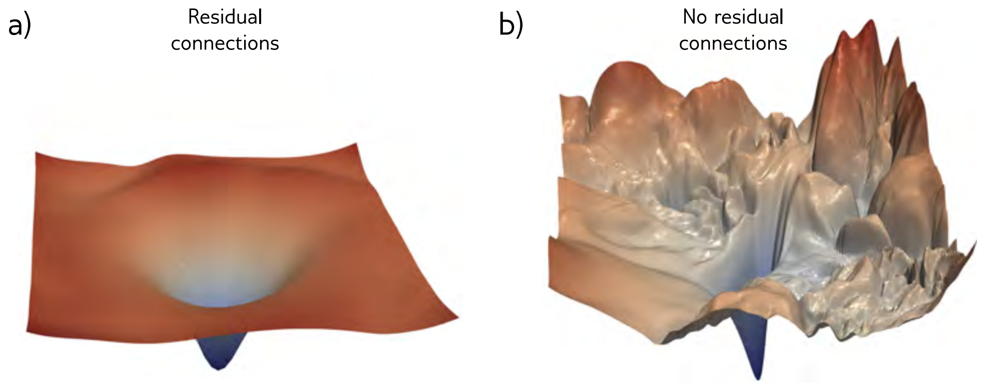

Figure 11.13 Visualizing neural network loss surfaces. Each plot shows the loss surface in two random directions in parameter space around the minimum found by SGD for an image classification task on the CIFAR-10 dataset. These directions are normalized to facilitate side-by-side comparison. a) Residual net with 56 layers. b) Results from the same network without skip connections. The surface is clearly smoother with the skip connections. This facilitates learning and makes the final network performance more robust to small errors in the parameters so that it will likely generalize better. Adapted from Li et al. (2018a).

First, shallower, wider residual networks sometimes outperform deeper, narrower ones with a comparable parameter count. In other words, better performance can sometimes be achieved with a network with fewer layers but more channels per layer. Second, there is evidence that the gradients during training do not propagate effectively through very long paths in the unraveled network (figure 11.4b). In effect, a very deep network may be acting more like a combination of shallower networks.

The current view is that residual connections add some value of their own, as well as allowing training of deeper networks. This perspective is supported by the fact that the loss surfaces of residual networks around a minimum tend to be smoother and more predictable than those for the same network when the skip connections are removed (figure 11.13). This may make it easier to learn a good solution that generalizes well.

11.7 Summary

Increasing network depth indefinitely causes both training and test performance to decrease for image classification tasks. It is conjectured that this is because the gradient of the loss with respect to the parameters early in the network changes quickly and unpredictably relative to the update step size. Residual connections add the processed representation back to their own input. This means that each layer contributes directly to the output as well as indirectly, so propagating gradients through many layers is not mandatory and the surface becomes smoother.

Residual networks do not suffer from vanishing gradients but introduce an exponential increase in the variance of the activations during forward propagation through the network, and corresponding problems with exploding gradients. This is usually handled by adding batch normalization layers to the network. These normalize by the empirical mean and variance of the batch and then shift and rescale using parameters that are learned. If these parameters are initialized judiciously, then very deep networks can be trained. There is evidence that both residual links and batch normalization make the loss surface smoother, which permits larger learning rates. Moreover, the variability in the batch statistics adds a source of regularization.

Residual blocks have been incorporated into convolutional networks. They allow deeper networks to be trained with commensurate increases in image classification performance. Variations of residual networks include the DenseNet architecture, which concatenates outputs of all prior layers to feed into the current layer, and U-Nets, which incorporate residual connections into encoder-decoder models.

Notes

Residual connections: Residual connections were introduced by He et al. (2016a) who built a network with 152 layers, which was eight times larger than VGG (figure 10.17), and which achieved state-of-the-art performance on the ImageNet classification task. Each residual block consisted of a convolutional layer followed by batch normalization, a ReLU activation, a second convolutional layer, and second batch normalization. A second ReLU function was then applied after this block was added back to the main representation. This architecture was termed *ResNet v1*. He et al. (2016b) investigated different variations of residual architectures, in which either (i) processing could also be applied along the skip connection or (ii) after the two branches had recombined. They concluded that neither of these was necessary leading to the architecture in figure 11.7, which is sometimes termed a *pre-activation residual block* and is the backbone of *ResNet v2*. They trained a network with 200 layers that improved further on the ImageNet classification task (see figure 11.8). Since this time, new methods for regularization, optimization, and data augmentation have been developed and Wightman et al. (2021) exploit these to present a more modern training pipeline for the ResNet architecture.

Why residual connections help: Residual networks certainly allow deeper networks to be trained. Presumably, this is related to reducing shattered gradients (Balduzzi et al., 2017) at the start of training and the smoother loss surface near the minima as depicted in figure 11.13 (Li et al., 2018a). Residual connections alone (i.e., without batch normalization) increase the trainable depth of a network by roughly a factor of two (Sankararaman et al., 2020). With batch normalization, very deep networks can be trained, but it is unclear that depth is critical for performance. Zagoruyko & Komodakis (2016) showed that wide residual networks with only 16 layers outperformed all residual networks of the time for image classification. Orhan & Pitkow (2017) propose a different explanation for why residual connections improve learning in terms of eliminating singularities (places on the loss surface where the Hessian is degenerate).

Related architectures: Residual connections are a special case of *highway networks* (Srivastava et al., 2015) which also split the computation into two branches and additively recombine. Highway networks use a gating function that weights the inputs to the two branches in a way that depends on the data itself, whereas residual networks send the data down both branches in a straightforward manner. Xie et al. (2017) introduced the ResNeXt architecture, which places a residual connection around multiple parallel convolutional branches.

Residual networks as ensembles: Veit et al. (2016) first characterized residual networks as ensembles of shorter networks and depicted the "unraveled network" interpretation (figure 11.4b). They provide evidence that this interpretation is valid by showing that deleting layers in a trained network (and hence a subset of paths) only has a modest effect on performance. Conversely, removing a layer has catastrophic consequences for a purely sequential network like VGG. They also looked at the gradient magnitudes along paths of different lengths and showed that the gradient vanishes in longer paths. In a residual network consisting of 54 blocks, almost all of the gradient updates during training were from paths of length 5 to 17 blocks long, even though these only constitute 0.45% of the total paths. In other words, the effect of adding more blocks seems to be mainly to add more parallel shorter paths rather than to create a network that is truly deeper.

Regularization for residual networks: L2 regularization has a fundamentally different effect in vanilla networks and residual networks without BatchNorm. In the former, it encourages the output of the layer to be a constant function determined by the bias. In the latter, it encourages the output of the residual block to compute the identity.

Several regularization methods have been developed that are targeted specifically at residual architectures. ResDrop (Yamada et al., 2016), stochastic depth (Huang et al., 2016), and RandomDrop (Yamada et al., 2019) all regularize residual networks by randomly dropping residual blocks during the training process. In the latter case, the propensity for dropping a block is determined by a Bernouilli variable, whose parameter is linearly decreased during training. At test time, the residual blocks are added back in with their expected probability. These methods are effectively versions of Dropout, in which all the hidden units in a block are simultaneously dropped in concert. In the multiple paths view of residual networks (figure 11.4b), they simply remove some of the paths at each training step. Wu et al. (2018b) developed BlockDrop which analyzes an existing network and decides which residual blocks to use at runtime with the goal of improving the efficiency of inference.

Separate regularization procedures have been developed for networks like ResNeXt which have multiple paths inside the residual block. Shake-shake (Gastaldi, 2017a,b) randomly re-weights these paths differently during the forward and backward passes. In the forward pass, this can be viewed as synthesizing random data, and in the backward pass as injecting another form of noise into the training method. ShakeDrop (Yamada et al., 2019) also draws a Bernoulli variable that decides whether each block will be subject to Shake-Shake or behave like a normal residual unit on this training iteration.

Batch normalization: Batch normalization was introduced by Ioffe & Szegedy (2015) outside of the context of residual networks. They showed empirically that it allowed higher learning rates, increased speed of convergence, and made sigmoid activation functions more practical (since the distribution of outputs is controlled, and so examples are less likely to fall in the saturated extremes of the sigmoid). Balduzzi et al. (2017) investigated the activation of hidden units in later layers of deep networks with ReLU functions at initialization. They showed that many such hidden units were always active or always inactive regardless of the input, but that BatchNorm reduced this tendency.

Although batch normalization helps stabilize the forward propagation of signals through a network, Yang et al. (2019) showed that it causes gradient explosion in ReLU networks without skip connections, with each layer increasing the magnitude of the gradients by $\sqrt{\pi/(\pi-1)} \approx 1.21$. This argument is summarized by Luther (2020). Since a residual network can be seen as a combination of paths of different lengths (figure 11.4), this effect must be present in residual networks as well. Presumably, however, the benefit of removing the 2^K increases in magnitude in

Problem 11.13

the forward pass of a network with K layers outweighs the harm done by increasing the gradients by 1.21^K in the backward pass, and so overall BatchNorm makes training more stable.

Variations of batch normalization: Several variants of BatchNorm have been proposed (figure 11.14). BatchNorm normalizes each channel separately based on statistics gathered across the batch. *Ghost batch normalization* or *GhostNorm* (Hoffer et al., 2017) uses only part of the batch to compute the normalization statistics, which makes them noisier and increases the amount of regularization when the batch size is very large (figure 11.14b).

When the batch size is very small, or the fluctuations within a batch are very large (as is often the case in natural language processing tasks), the statistics in BatchNorm may become unreliable. Ioffe (2017) proposed *batch renormalization*, which keeps a running average of the batch statistics and modifies the normalization of any batch to ensure that it is more representative. A second problem with batch normalization is that it is unsuitable for use in recurrent neural networks (networks for processing sequence, in which the last output is fed back as an additional input as we move through the sequence (see figure 12.19). This is because the statistics must be stored at each step in the sequence, and it's not clear what to do if a test sequence is longer than the training sequences. A third problem is that batch normalization needs access to the whole batch, but this may not be easily available if the training is distributed across several machines using data parallelism.

Layer normalization or *LayerNorm* (Ba et al., 2016) avoids using the batch statistics by normalizing each data example separately, using statistics gathered across the channels and spatial position (figure 11.14c). However, there is still a separate learned scale γ and offset δ per channel. *Group normalization* or *GroupNorm* (Wu & He, 2018) is similar to LayerNorm but divides the channels into groups and computes the statistics for each group separately, computing statistics across the within-group channels and spatial position (figure 11.14d). Again, there are still separate scale and offset parameters per channel. *Instance normalization* or *InstanceNorm* (Ulyanov et al., 2016) takes this to the extreme where the number of groups is the same as the number of channels, and so each channel is normalized separately (figure 11.14e), using statistics gathered across spatial position alone. Salimans & Kingma (2016) investigated normalizing the network weights rather than the activations, but this has been less empirically successful. Teye et al. (2018) introduced *Monte Carlo batch normalization*, which can provide meaningful estimates of uncertainty in the predictions of neural networks. A recent comparison of the properties of different normalization schemes can be found in Lubana et al. (2021).

Why BatchNorm helps: BatchNorm helps control the initial gradients in a residual network (figure 11.6c). However, the mechanism by which BatchNorm improves performance more generally is not well understood. The stated goal of Ioffe & Szegedy (2015) was to reduce problems caused by *internal covariate shift*, which is the change in the distribution of inputs to a layer caused by updating preceding layers during the backpropagation update. However, Santurkar et al. (2018) provided evidence against this view by artificially inducing covariate shift and showing that networks with and without BatchNorm performed equally well.

Motivated by this, they searched for another explanation as to why BatchNorm should improve performance. They showed empirically for the VGG network that adding batch normalization decreases the variation in both the loss and its gradient as we move in the gradient direction. In other words, the loss surface is both smoother and changes more slowly, which is why larger learning rates are possible. They also provide theoretical proofs for both these phenomena and show that for any parameter initialization, the distance to the nearest optimum is less for networks with batch normalization. Bjorck et al. (2018) also argue that BatchNorm improves the properties of the loss landscape and allows larger learning rates.

Other explanations of why BatchNorm improves performance include that it decreases the importance of tuning the learning rate (Ioffe & Szegedy, 2015; Arora et al., 2018). Indeed Li & Arora (2019) show that it's possible to use an exponentially increasing learning rate schedule with batch normalization. Ultimately, this is because batch normalization makes the network

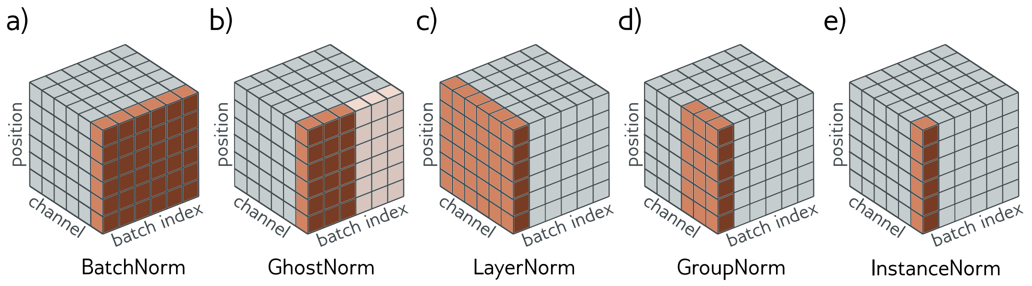

Figure 11.14 Normalization schemes. BatchNorm modifies each channel separately, but modifies each member of the batch in the same way based on statistics gathered across the batch and spatial position. Ghost BatchNorm computes these statistics from only part of the batch to make them more variable. LayerNorm computes statistics for each member of the batch separately, based on statistics gathered across the channels and spatial position. It retains a separate learned scaling factor for each channel. GroupNorm normalizes within each group of channels, and also retains a separate scale and offset parameter for each channel. InstanceNorm normalizes within each channel separately computing the statistics across spatial position only. Adapted from Wu & He (2018).

invariant to the scales of the weight matrices. An intuitive visualization of this effect is provided by Huszár (2019).

Hoffer et al. (2017) identified that BatchNorm has a regularizing effect due to fluctuations in statistics due to the random composition of the batch. They proposed manipulating this directly by using a *ghost batch size*, in which the mean and standard deviation statistics are computed over a subset of the batch. In this way, large batches can be used, without losing the regularizing effect of the additional noise when the batch size is smaller. Luo et al. (2018) investigate the regularization effects of batch normalization.

Alternatives to batch normalization: Although BatchNorm is widely used, it is not strictly necessary to train deep residual nets; there are other ways of making the loss surface tractable. Balduzzi et al. (2017) proposed the rescaling by $\sqrt{1/2}$ in figure 11.6b; they argued that it prevents gradient explosion, but does not resolve the problem of shattered gradients.

Other work has investigated rescaling the output of the function in the residual block before addition back to the input. For example, De & Smith (2020) introduce SkipInit, in which a learnable scalar multiplier is placed at the end of each residual branch. This helps as long as this multiplier is initialized to less than $\sqrt{1/K}$, where K is the number of residual blocks. In practice, they suggest initializing this to zero. Similarly, Hayou et al. (2021) introduce Stable ResNet, which rescales the output of the function in the k^{th} residual block (before addition to the main branch) by a constant λ_k. They prove that in the limit of infinite width, the expected gradient norm of the weights in the first layer is lower bounded by the sum of squares of the scalings λ_k. They investigate setting these to a constant $\sqrt{1/K}$, where K is the number of residual blocks and show that it is possible to train networks with up to 1000 blocks.

Zhang et al. (2019a) introduce a method called *FixUp*, in which every layer is initialized using He normalization, but the last linear/convolutional layer of every residual block is set to zero. This means that the initial forward pass is stable (since each residual block contributes nothing) and the gradients do not explode in the backward pass (for the same reason). They also rescale the branches so that the magnitude of the total expected change in the parameters is constant regardless of the number of residual blocks. Although these methods allow training of deep residual networks, they do not generally achieve the same test performance as when using BatchNorm. This is probably because they do not benefit from the regularization induced by the noisy batch statistics. De & Smith (2020) modify their method to induce regularization via dropout which helps close this gap.

DenseNet and U-Net: DenseNet was first introduced by Huang et al. (2017b), U-Net was developed by Ronneberger et al. (2015), and stacked hourglass networks by Newell et al. (2016). Of these three architectures, U-Net has been most extensively adapted. Çiçek et al. (2016) introduced 3D U-Net and Milletari et al. (2016) introduced VNet, both of which extend U-Net to process 3D data. Zhou et al. (2018) combine the ideas of DenseNet and U-Net in an architecture that both downsamples and re-upsamples the image, but also repeatedly uses intermediate representations. U-Nets are commonly used in medical image segmentation and a review of this work can be found in Siddique et al. (2021). However, they have been applied to other areas including depth estimation (Alhashim & Wonka, 2018), semantic segmentation (Iglovikov & Shvets, 2018), inpainting (Zeng et al., 2019), pansharpening (Yao et al., 2018), and image-to-image translation (Isola et al., 2017). U-Nets are also a key component in diffusion models (chapter 18).

Problems

Problem 11.1 Derive equation 11.5 from equations 11.4 by substituting in the expression for $\mathbf{h}_1, \mathbf{h}_2$, and $\mathbf{h}_3$.

Problem 11.2 The network in figure 11.4a has four residual blocks. When we unravel this network, we get one path of length zero, four paths of length one, six paths of length two, four paths of length three, and one path of length four. How many paths of each length would there be if there were (i) three residual blocks and (ii) five residual blocks? Deduce the rule for K residual blocks.

Problem 11.3 Show that the derivative of the network in equation 11.5 with respect to the first layer $\mathbf{f}_1[\mathbf{x}]$ is given by equation 11.6.

Problem 11.4 Consider a residual block, where the block contents comprise a standard linear transformation plus ReLU layer and we have used He initialization. Explain why the distributions of the activations in the block and the skip connection are independent.

Problem 11.5 Write native Python code that shows how the variance of the activations in the forward pass and gradients in the backward pass increases in a residual network as a function of network depth (i.e., similar to figure 7.7). The network should take $D = 100$-dimensional inputs $\mathbf{x}$, which are drawn from a standard normal distribution. The first layer of the network is a linear transform consisting of a 100×100 weight matrix and a 100×1 bias vector. This is followed by 50 residual blocks of the type shown in figure 11.5b and finally another linear transform that maps to a single output f. All parameters are initialized with He initialization. The targets y are also drawn from a standard normal distribution and a least squares loss function is used.

Problem 11.6 Show that adding a normalization factor of $1/\sqrt{2}$ after each residual block in your code for problem 11.5 stabilizes the variance of the intermediate values in the forward pass.

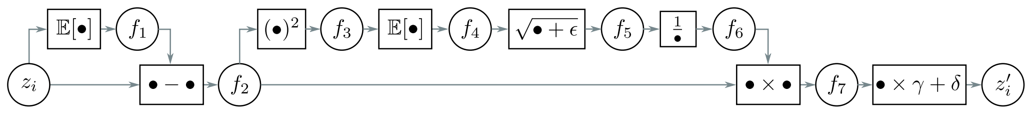

Figure 11.15 Computational graph for batch normalization (see problem 11.10).

Problem 11.7 The forward pass for batch normalization given a batch of scalar values $\{z_i\}_{i=1}^{I}$ consists of the following operations (figure 11.15):

$$\begin{aligned} f_1 &= \mathbb{E}[z_i] & f_5 &= \sqrt{f_4 + \epsilon} \\ f_{2i} &= x_i - f_1 & f_6 &= 1/f_5 \\ f_{3i} &= f_{2i}^2 & f_{7i} &= f_{2i} \times f_6 \\ f_4 &= \mathbb{E}[f_{3i}] & z'_i &= f_{7i} \times \gamma + \delta, \end{aligned} \tag{11.10}$$

where $\mathbb{E}[z_i] = \frac{1}{I}\sum_i z_i$. Write Python code to implement the forward pass.

Now derive the algorithm for the backward pass. Work backward through the computational graph computing the derivatives to generate a set of operations that computes $\partial z'_i/\partial z_i$ for every element in the batch. Write Python code to implement the backward pass.

Problem 11.8 Consider a fully connected neural network with one input, one output, and ten hidden layers, each of which contains twenty hidden units. How many parameters does this network have? How many parameters will it have if we place a batch normalization operation between each linear transformation and ReLU?

Problem 11.9 Consider applying a L2 regularization penalty to the weights in the convolutional layers in figure 11.7a, but not to the scaling parameters of the subsequent BatchNorm layers. What do you expect will happen as training proceeds?

Problem 11.10 Consider a convolutional residual block that contains a batch normalization operation, followed by a ReLU activation function, and then a 3×3 convolutional layer. If the input has 512 channels, then how many parameters are needed to define this block? Now consider a bottleneck residual block that contains three batch normalization / activation / ReLU sequences. The first is a 1×1 convolutional layer that reduces the number of channels from 512 to 128. The second is a 3×3 convolutional layer with the same number of input and output channels. The third is a 1×1 convolutional layer that increases the number of channels from 128 to 512 (see figure 11.7b). How many parameters are needed to define this block?

Problem 11.11 The DenseNet architecture (figure 11.9) can be described by the equations:

$$\begin{aligned} \mathbf{h}_1 &= \mathbf{f}_1[\mathbf{x}] \\ \mathbf{h}_2 &= \mathbf{f}_2[\mathbf{concat}[\mathbf{x}, \mathbf{h}_1]] \\ \mathbf{h}_3 &= \mathbf{f}_3[\mathbf{concat}[\mathbf{x}, \mathbf{h}_1, \mathbf{h}_2]] \\ \mathbf{y} &= \mathbf{concat}[\mathbf{x}, \mathbf{h}_1, \mathbf{h}_2, \mathbf{h}_3] \end{aligned} \tag{11.11}$$

Draw this network in the unraveled style (figure 11.4).

Problem 11.12 The U-Net is completely convolutional and can be run with any sized image after training. In principle, we could also train with a collection of arbitrary sized images. Why do you think this is not done in practice?

Problem 11.13 Figure 7.7 shows that the variance of the activations during forward propagation and the variance of the gradients during backward propagation in vanilla ReLU networks is stabilized by He initialization. Repeat this experiment, but with a BatchNorm layer with $\gamma = 1$ and $\delta = 0$ after each ReLU activation function. Write code to show that the gradients now increase as we move backward through the network at a rate of approximately 1.21 per layer.