Understanding Deep Learning

Chapter 7: Gradients and Initialization

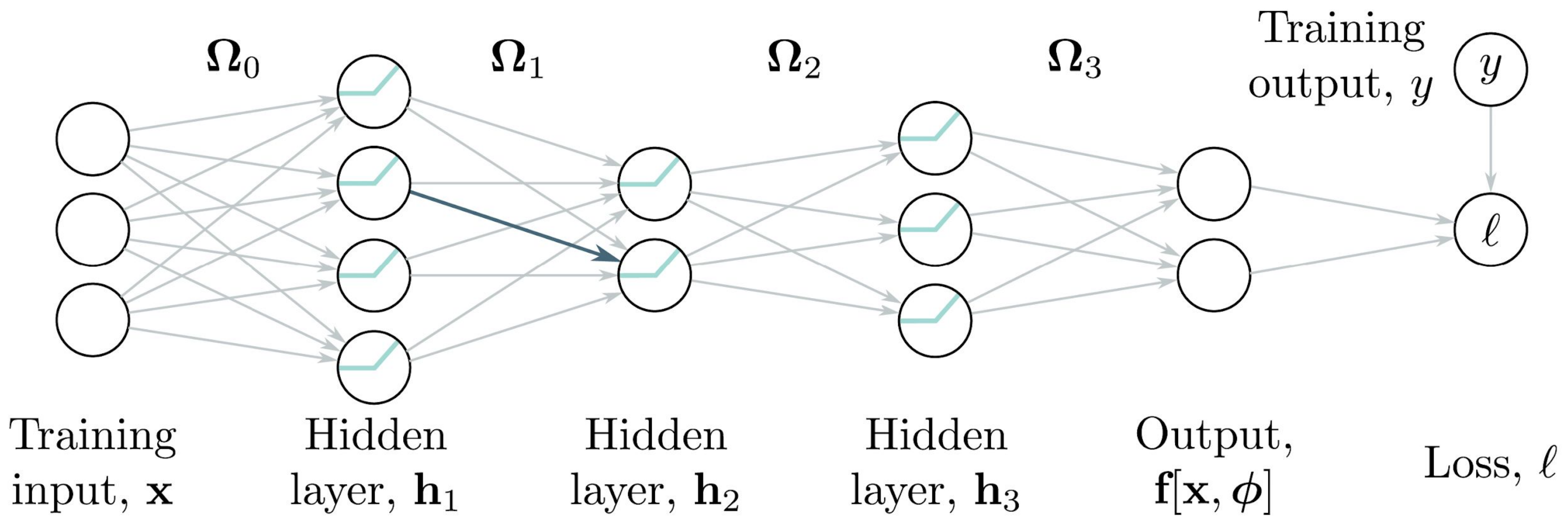
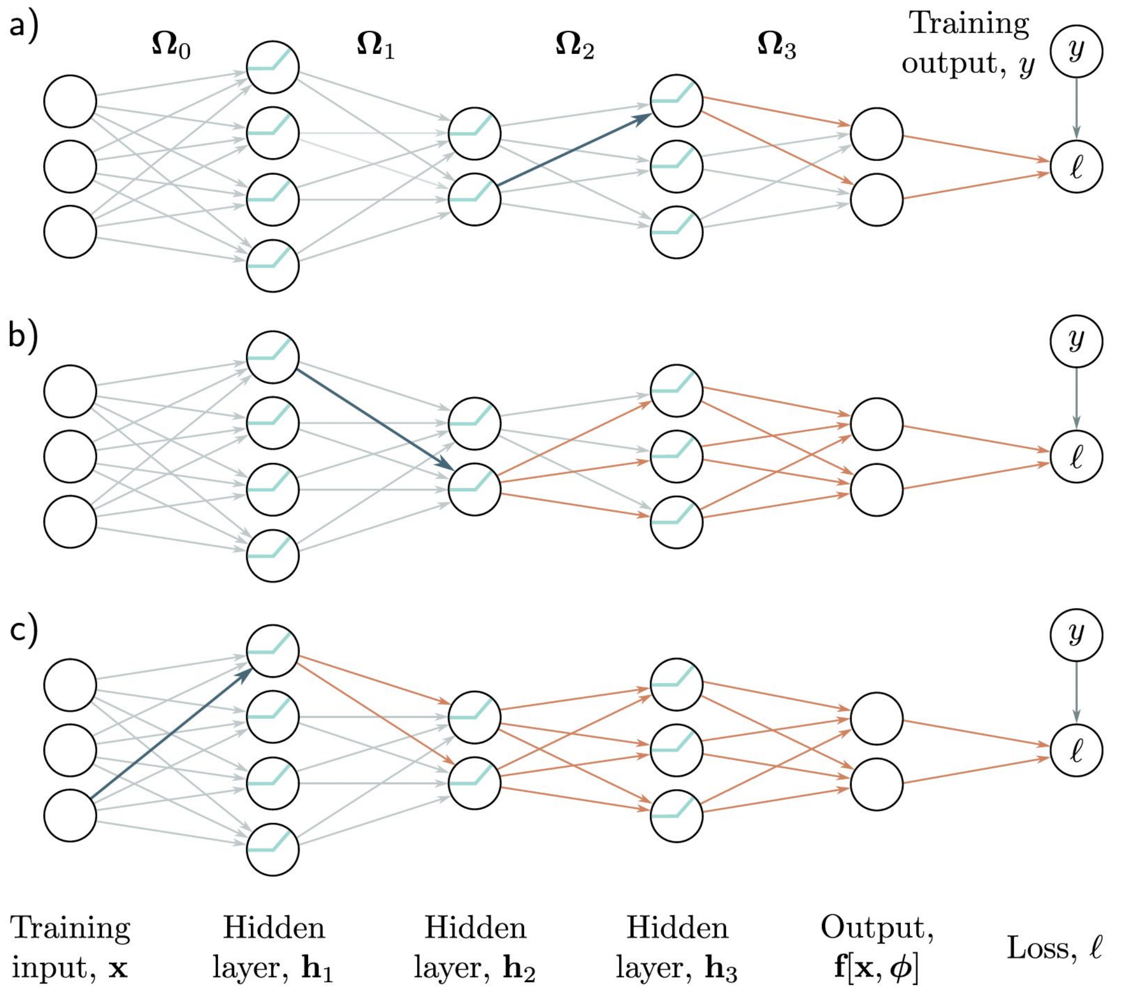

$\boldsymbol{\Omega}_0$
$\boldsymbol{\Omega}_1$
$\boldsymbol{\Omega}_2$
$\boldsymbol{\Omega}_3$
Training output, y
y
ℓ
Training input, $\mathbf{x}$
Hidden layer, $\mathbf{h}_1$
Hidden layer, $\mathbf{h}_2$
Hidden layer, $\mathbf{h}_3$
Output, $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$
Loss, ℓ

Figure 7.1 Backpropagation forward pass. The goal is to compute the derivatives of the loss ℓ with respect to each of the weights (arrows) and biases (not shown). In other words, we want to know how a small change to each parameter will affect the loss. Each weight multiplies the hidden unit at its source and contributes the result to the hidden unit at its destination. Consequently, the effects of any small change to the weight will be scaled by the activation of the source hidden unit. For example, the blue weight is applied to the second hidden unit at layer 1; if the activation of this unit doubles, then the effect of a small change to the blue weight will double too. Hence, to compute the derivatives of the weights, we need to calculate and store the activations at the hidden layers. This is known as the *forward pass* since it involves running the network equations sequentially.



a)
Ω_0
Ω_1
Ω_2
Ω_3
Training output, y
y
ℓ
b)
c)
Training input, $\mathbf{x}$
Hidden layer, $\mathbf{h}_1$
Hidden layer, $\mathbf{h}_2$
Hidden layer, $\mathbf{h}_3$
Output, $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$
Loss, ℓ

Figure 7.2 Backpropagation backward pass. a) To compute how a change to a weight feeding into layer $\mathbf{h}_3$ (blue arrow) changes the loss, we need to know how the hidden unit in $\mathbf{h}_3$ changes the model output $\mathbf{f}$ and how $\mathbf{f}$ changes the loss (orange arrows). b) To compute how a small change to a weight feeding into $\mathbf{h}_2$ (blue arrow) changes the loss, we need to know (i) how the hidden unit in $\mathbf{h}_2$ changes $\mathbf{h}_3$, (ii) how $\mathbf{h}_3$ changes $\mathbf{f}$, and (iii) how $\mathbf{f}$ changes the loss (orange arrows). c) Similarly, to compute how a small change to a weight feeding into $\mathbf{h}_1$ (blue arrow) changes the loss, we need to know how $\mathbf{h}_1$ changes $\mathbf{h}_2$ and how these changes propagate through to the loss (orange arrows). The backward pass first computes derivatives at the end of the network and then works backward to exploit the inherent redundancy of these computations.

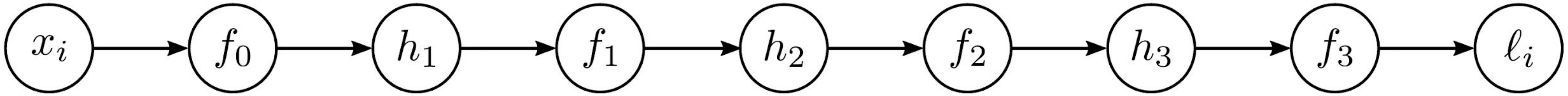

$$\begin{tikzcd} x_i \arrow[r] & f_0 \arrow[r] & h_1 \arrow[r] & f_1 \arrow[r] & h_2 \arrow[r] & f_2 \arrow[r] & h_3 \arrow[r] & f_3 \arrow[r] & \ell_i \end{tikzcd}$$

Figure 7.3 Backpropagation forward pass. We compute and store each of the intermediate variables in turn until we finally calculate the loss.

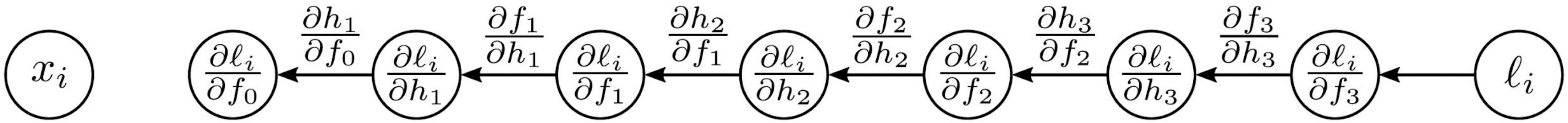
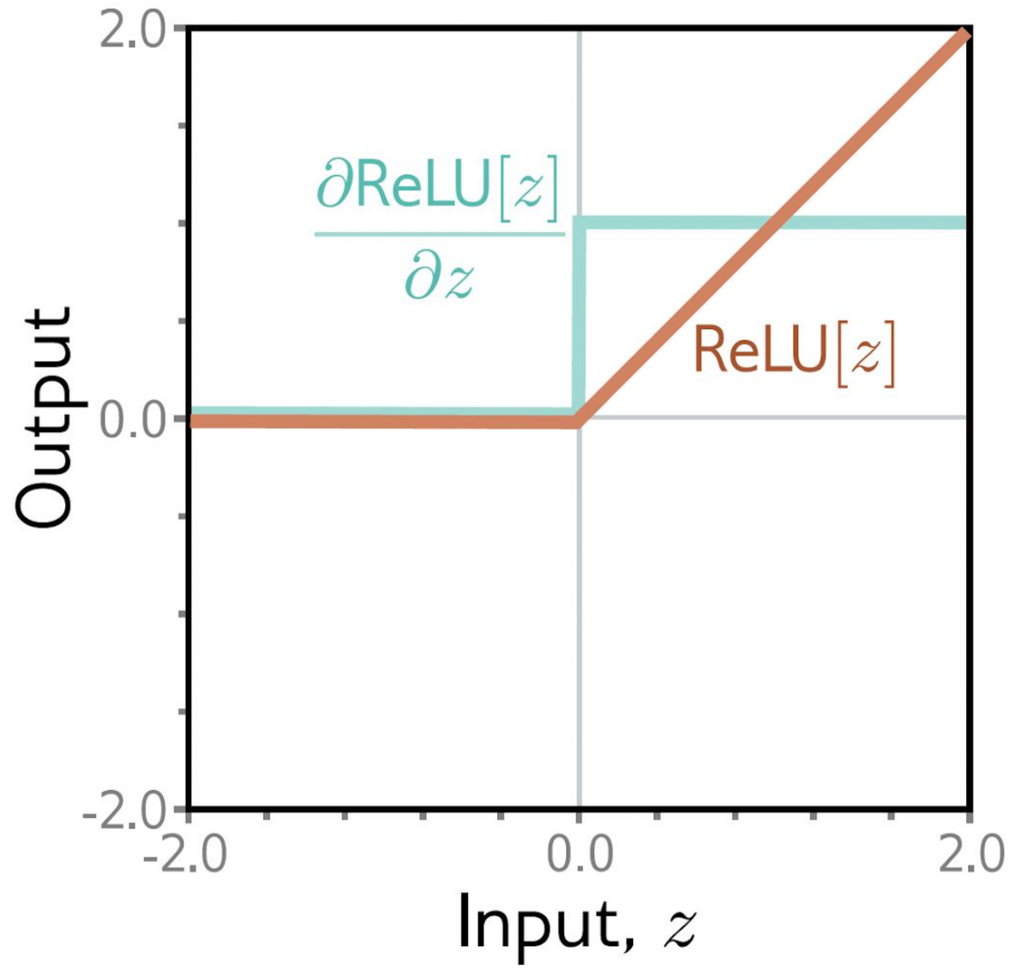

x_i
$\frac{\partial \ell_i}{\partial f_0}$
$\frac{\partial h_1}{\partial f_0}$
$\frac{\partial \ell_i}{\partial h_1}$
$\frac{\partial f_1}{\partial h_1}$
$\frac{\partial \ell_i}{\partial f_1}$
$\frac{\partial h_2}{\partial f_1}$
$\frac{\partial \ell_i}{\partial h_2}$
$\frac{\partial f_2}{\partial h_2}$
$\frac{\partial \ell_i}{\partial f_2}$
$\frac{\partial h_3}{\partial f_2}$
$\frac{\partial \ell_i}{\partial h_3}$
$\frac{\partial f_3}{\partial h_3}$
$\frac{\partial \ell_i}{\partial f_3}$
ℓ_i

Figure 7.4 Backpropagation backward pass #1. We work backward from the end of the function computing the derivatives $\partial \ell_i/\partial f_\bullet$ and $\partial \ell_i/\partial h_\bullet$ of the loss with respect to the intermediate quantities. Each derivative is computed from the previous one by multiplying by terms of the form $\partial f_k/\partial h_k$ or $\partial h_k/\partial f_{k-1}$.

$$\begin{tikzcd}
x_i & & \frac{\partial \ell_i}{\partial f_0} \arrow[dl, "\frac{\partial f_0}{\partial \beta_0}"'] \arrow[dr, "\frac{\partial f_0}{\partial \omega_0}"] & & \frac{\partial \ell_i}{\partial h_1} \arrow[ll] & & \frac{\partial \ell_i}{\partial f_1} \arrow[ll] \arrow[dl, "\frac{\partial f_1}{\partial \beta_1}"'] \arrow[dr, "\frac{\partial f_1}{\partial \omega_1}"] & & \frac{\partial \ell_i}{\partial h_2} \arrow[ll] & & \frac{\partial \ell_i}{\partial f_2} \arrow[ll] \arrow[dl, "\frac{\partial f_2}{\partial \beta_2}"'] \arrow[dr, "\frac{\partial f_2}{\partial \omega_2}"] & & \frac{\partial \ell_i}{\partial h_3} \arrow[ll] & & \frac{\partial \ell_i}{\partial f_3} \arrow[ll] \arrow[dl, "\frac{\partial f_3}{\partial \beta_3}"'] \arrow[dr, "\frac{\partial f_3}{\partial \omega_3}"] & & \ell_i \arrow[ll] \\
& \frac{\partial \ell_i}{\partial \beta_0} & & \frac{\partial \ell_i}{\partial \omega_0} & & \frac{\partial \ell_i}{\partial \beta_1} & & \frac{\partial \ell_i}{\partial \omega_1} & & \frac{\partial \ell_i}{\partial \beta_2} & & \frac{\partial \ell_i}{\partial \omega_2} & & \frac{\partial \ell_i}{\partial \beta_3} & & \frac{\partial \ell_i}{\partial \omega_3} &
\end{tikzcd}$$

Figure 7.5 Backpropagation backward pass #2. Finally, we compute the derivatives $\partial\ell_i/\partial\beta_\bullet$ and $\partial\ell_i/\partial\omega_\bullet$. Each derivative is computed by multiplying the term $\partial\ell_i/\partial f_k$ by $\partial f_k/\partial\beta_k$ or $\partial f_k/\partial\omega_k$ as appropriate.

Figure 7.6 Derivative of rectified linear unit. The rectified linear unit (orange curve) returns zero when the input is less than zero and returns the input otherwise. Its derivative (cyan curve) returns zero when the input is less than zero (since the slope here is zero) and one when the input is greater than zero (since the slope here is one).



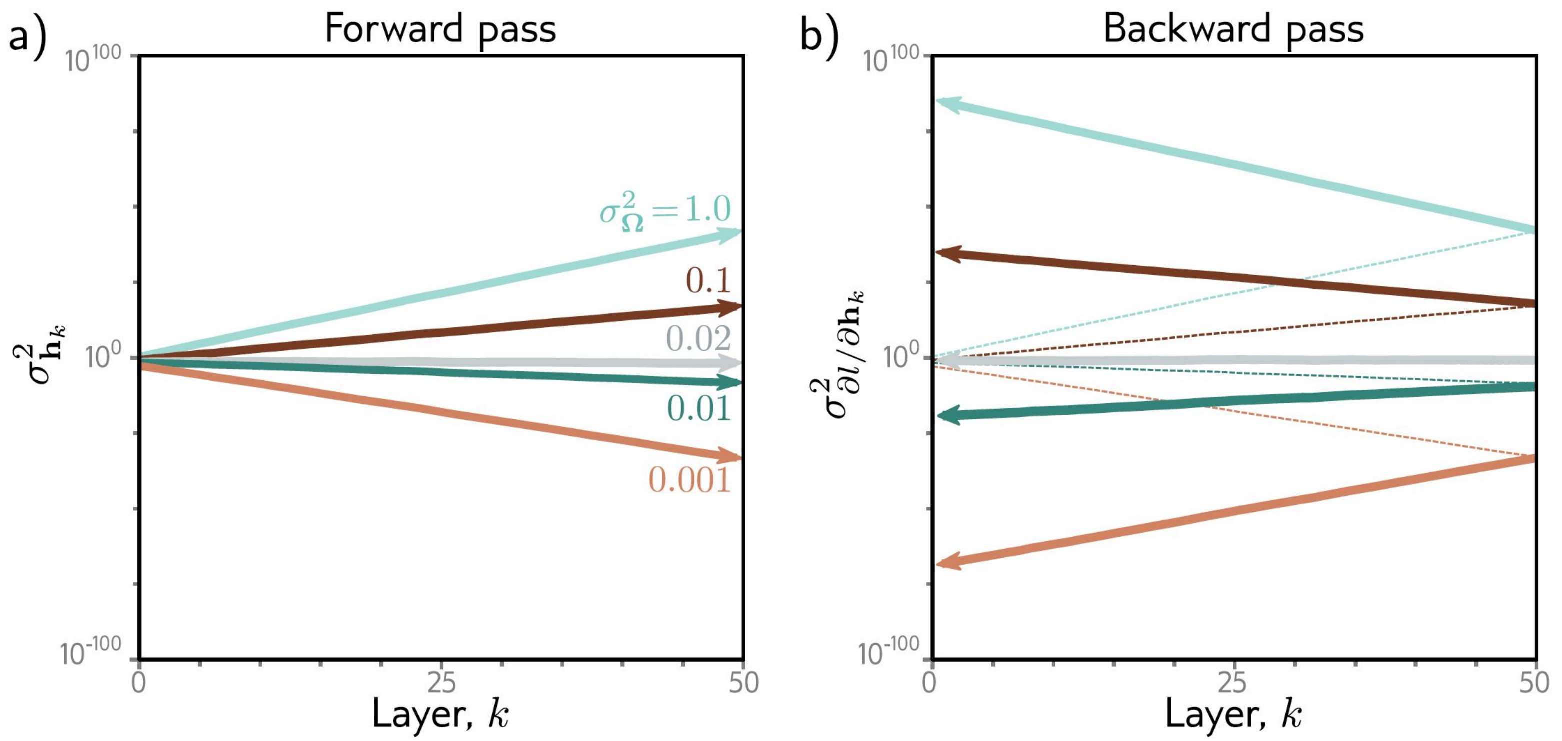


Figure 7.7 Weight initialization. Consider a deep network with 50 hidden layers and $D_h = 100$ hidden units per layer. The network has a 100-dimensional input $\mathbf{x}$ initialized from a standard normal distribution, a single fixed target $y = 0$, and a least squares loss function. The bias vectors $\boldsymbol{\beta}_k$ are initialized to zero, and the weight matrices $\boldsymbol{\Omega}_k$ are initialized with a normal distribution with mean zero and five different variances $\sigma^2_{\boldsymbol{\Omega}} \in \{0.001, 0.01, 0.02, 0.1, 1.0\}$. a) Variance of hidden unit activations computed in forward pass as a function of the network layer. For He initialization ($\sigma^2_{\boldsymbol{\Omega}} = 2/D_h = 0.02$), the variance is stable. However, for larger values, it increases rapidly, and for smaller values, it decreases rapidly (note log scale). b) The variance of the gradients in the backward pass (solid lines) continues this trend; if we initialize with a value larger than 0.02, the magnitude of the gradients increases rapidly as we pass back through the network. If we initialize with a value smaller, then the magnitude decreases. These are known as the *exploding gradient* and *vanishing gradient* problems, respectively.

```
import torch, torch.nn as nn
from torch.utils.data import TensorDataset, DataLoader
from torch.optim.lr_scheduler import StepLR

# define input size, hidden layer size, output size
D_i, D_k, D_o = 10, 40, 5
# create model with two hidden layers
model = nn.Sequential(
    nn.Linear(D_i, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_k),
    nn.ReLU(),
    nn.Linear(D_k, D_o))

# He initialization of weights
def weights_init(layer_in):
    if isinstance(layer_in, nn.Linear):
        nn.init.kaiming_uniform(layer_in.weight)
        layer_in.bias.data.fill_(0.0)
model.apply(weights_init)

# choose least squares loss function
criterion = nn.MSELoss()
# construct SGD optimizer and initialize learning rate and momentum
optimizer = torch.optim.SGD(model.parameters(), lr = 0.1, momentum=0.9)
# object that decreases learning rate by half every 10 epochs
scheduler = StepLR(optimizer, step_size=10, gamma=0.5)

# create 100 random data points and store in data loader class
x = torch.randn(100, D_i)
y = torch.randn(100, D_o)
data_loader = DataLoader(TensorDataset(x,y), batch_size=10, shuffle=True)

# loop over the dataset 100 times
for epoch in range(100):
    epoch_loss = 0.0
    # loop over batches
    for i, data in enumerate(data_loader):
        # retrieve inputs and labels for this batch
        x_batch, y_batch = data
        # zero the parameter gradients
        optimizer.zero_grad()
        # forward pass
        pred = model(x_batch)
        loss = criterion(pred, y_batch)
        # backward pass
        loss.backward()
        # SGD update
        optimizer.step()
        # update statistics
        epoch_loss += loss.item()
    # print error
    print(f'Epoch {epoch:5d}, loss {epoch_loss:.3f}')
    # tell scheduler to consider updating learning rate
    scheduler.step()
```

Figure 7.8 Sample code for training two-layer network on random data.

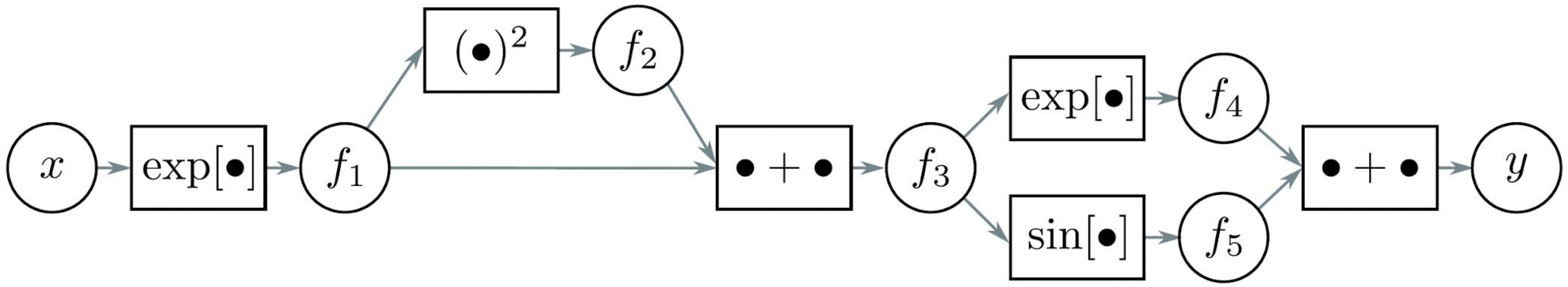


Figure 7.9 Computational graph for problem 7.12 and problem 7.13. Adapted from Domke (2010).