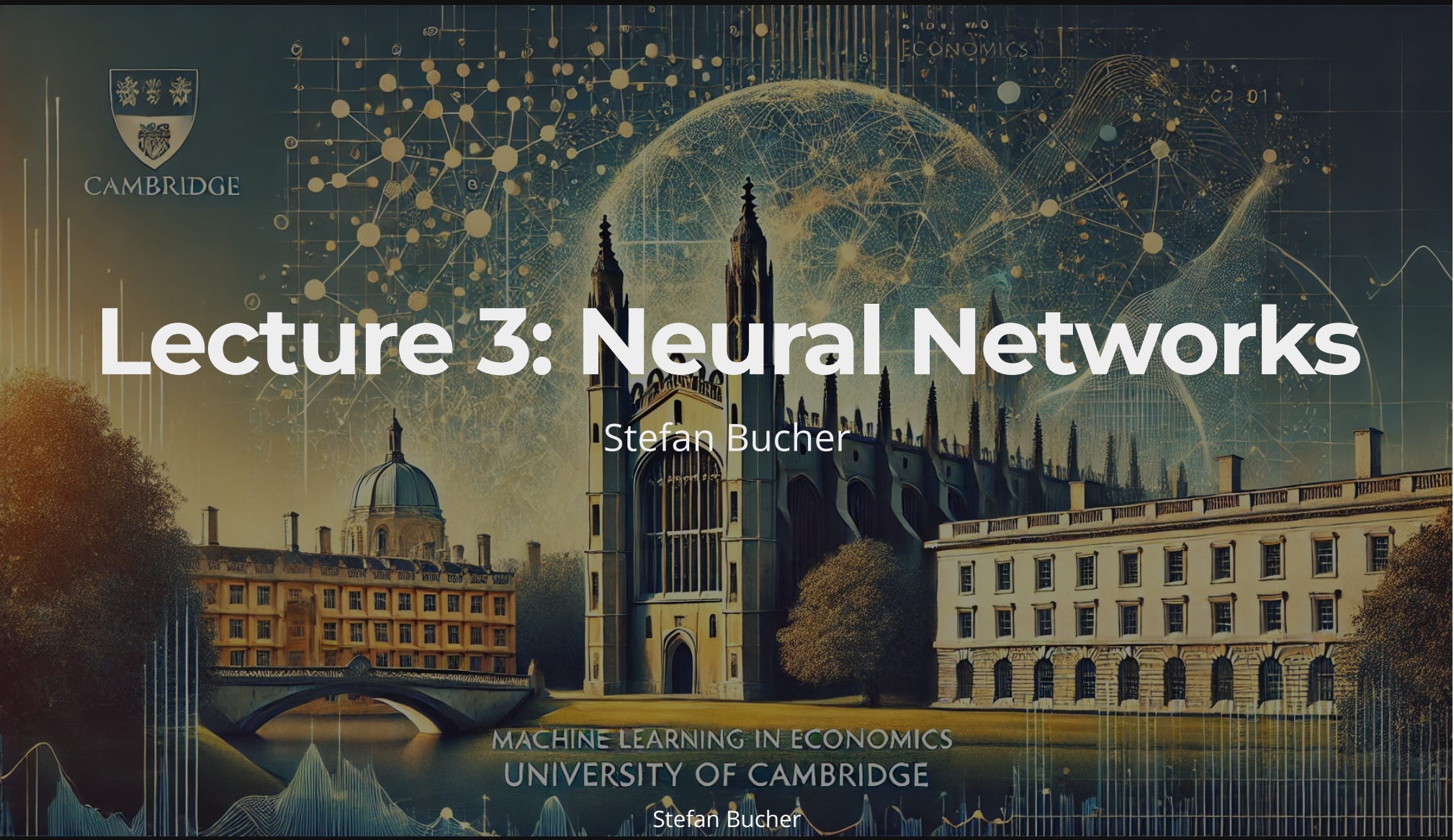


Lecture 3: Neural Networks

Stefan Bucher

Stefan Bucher

Open in Colab

Prince (2023, chaps. 3, 4, 7).[1]

. Figures taken or adapted from Prince (2023). All rights belong to the original author and publisher. These materials are intended solely for educational purposes.

Shallow Neural Networks

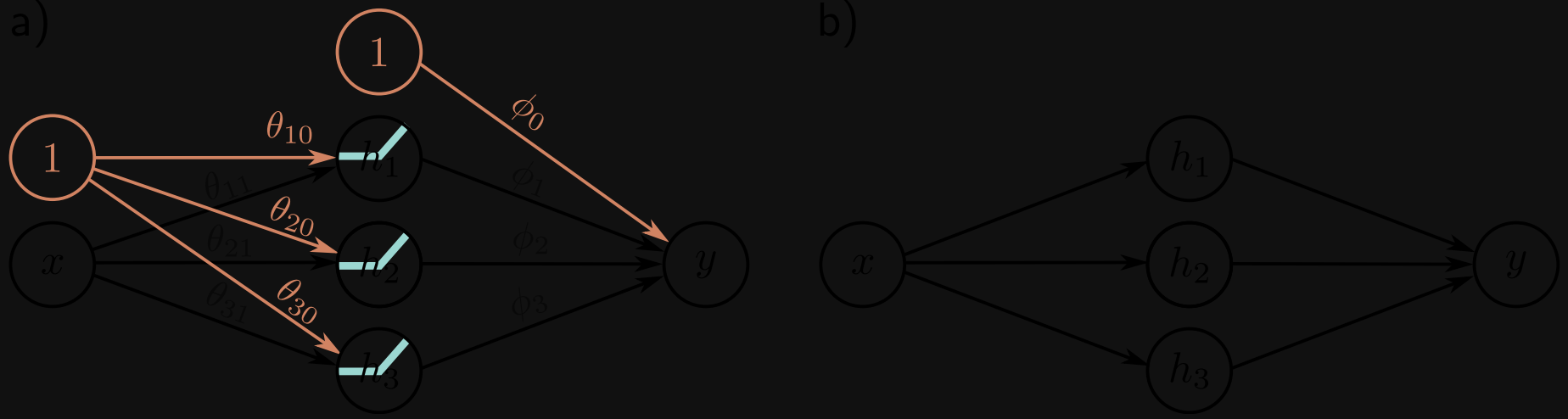
Prince (2023, chap. 3)

Example

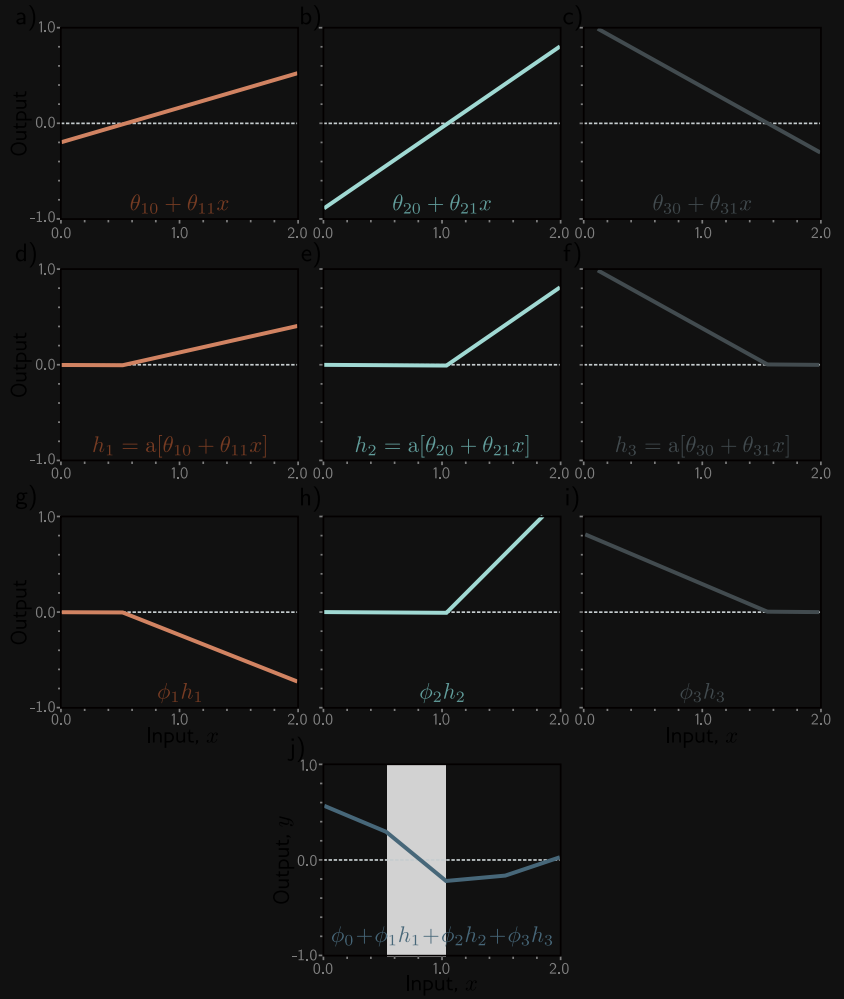
$$f[x,\phi] = \phi_0 + \phi_1 \underbrace{a(\theta_{10} + \theta_{11}x)}_{h_1} + \phi_2 \underbrace{a(\theta_{20} + \theta_{21}x)}_{h_2} + \phi_3 \underbrace{a(\theta_{30} + \theta_{31}x)}_{h_3}$$

$$a(z) = ReLU(z) = \max\{z, 0\}$$

Neural networks (shallow & deep!) with ReLU represent continuous piecewise linear functions.

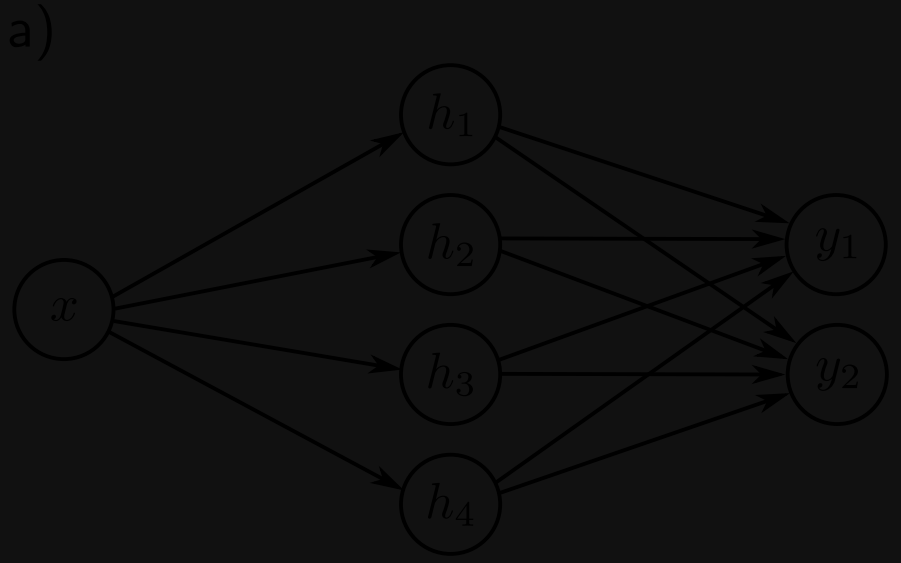


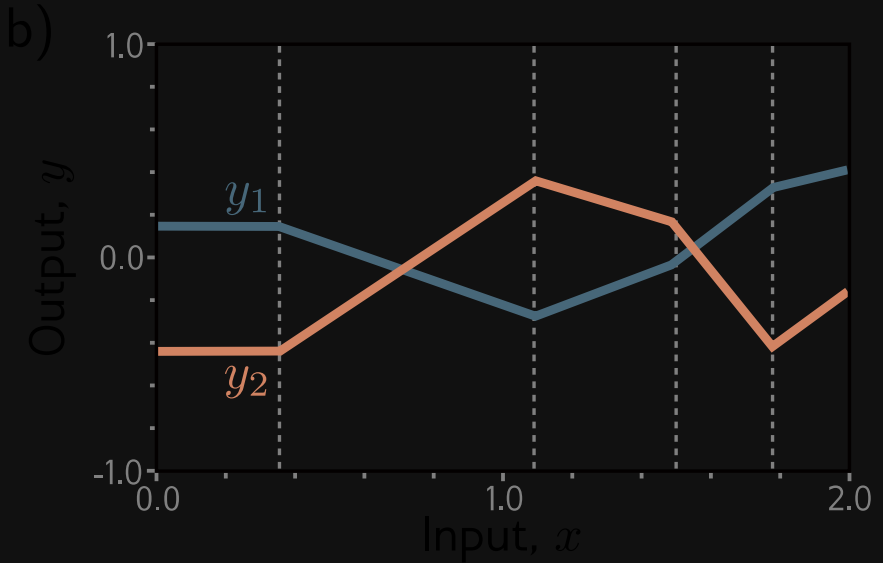
Activation Patterns



Joints where one hidden unit becomes (in)active.

Multivariate Output

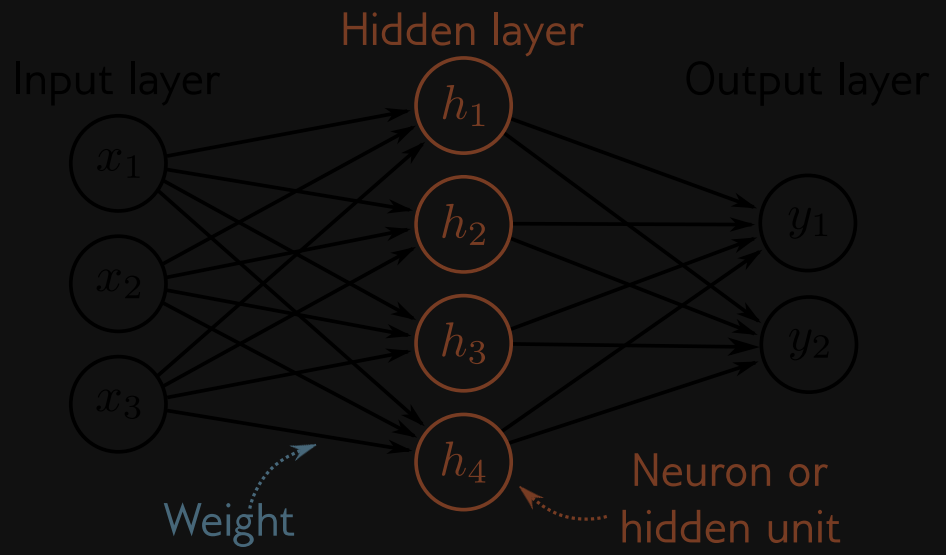




Shallow Neural Networks

Map n_{in}-dimensional input to n_{out}-dimensional output.

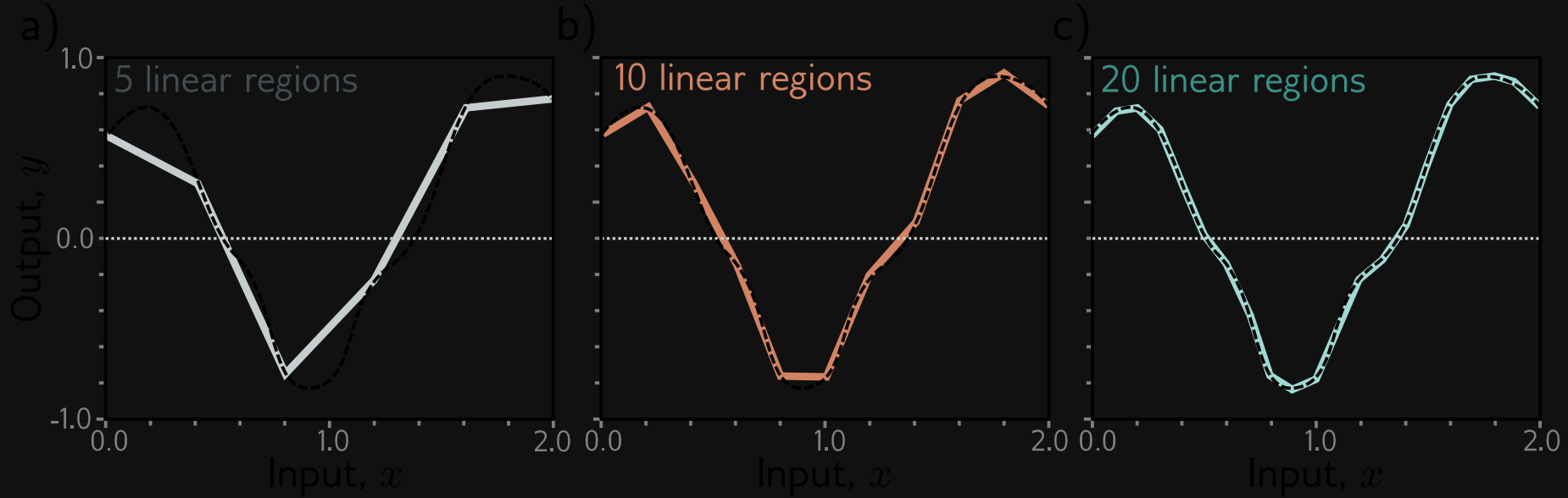
$$y_j = \phi_{j0} + \sum_{d=1}^{D} \phi_{jd} \underbrace{a(\theta_{d0} + \sum_{i=1}^{n_{in}} \theta_{di} x_i)}_{h_d}, \; j = 1, \ldots, n_{out}$$



Shallow Networks (3.3 & 3.8)

Universal Approximation Thm

∃ shallow neural network with a single layer of sufficiently many hidden units that can approximate, with arbitrary precision, any continuous function from a compact subset of $R^{n_{in}}$ to $R^{n_{out}}$.



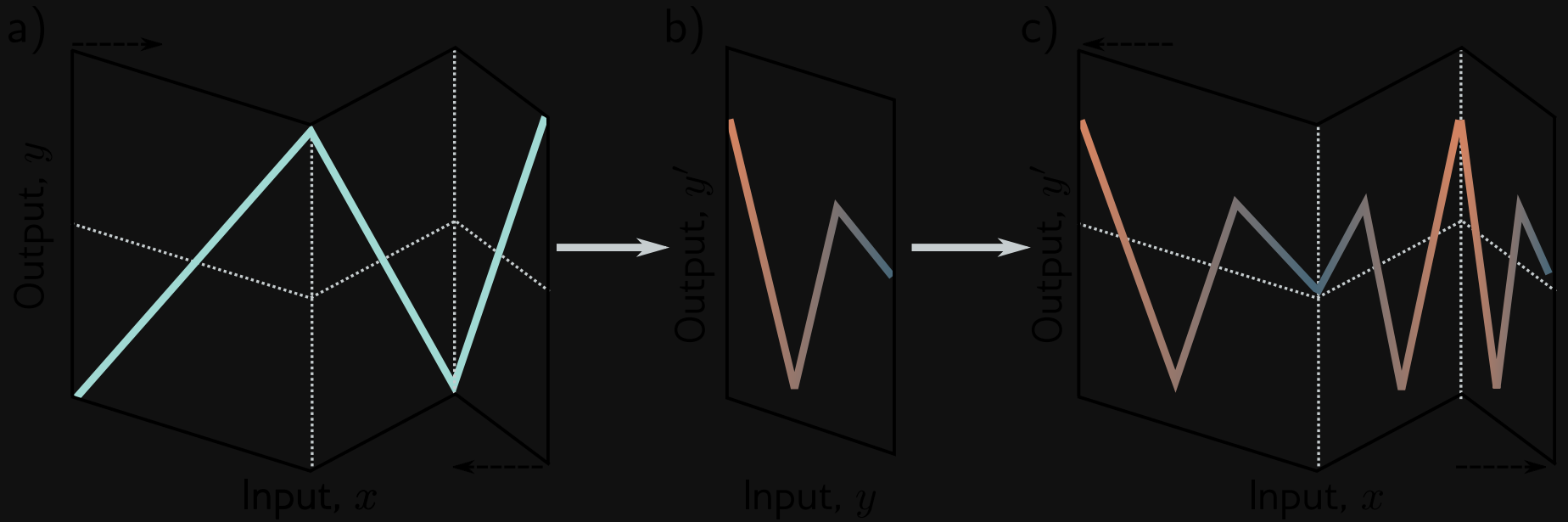
Deep Neural Networks

Prince (2023, chap. 4)

Why Depth? (4.1)

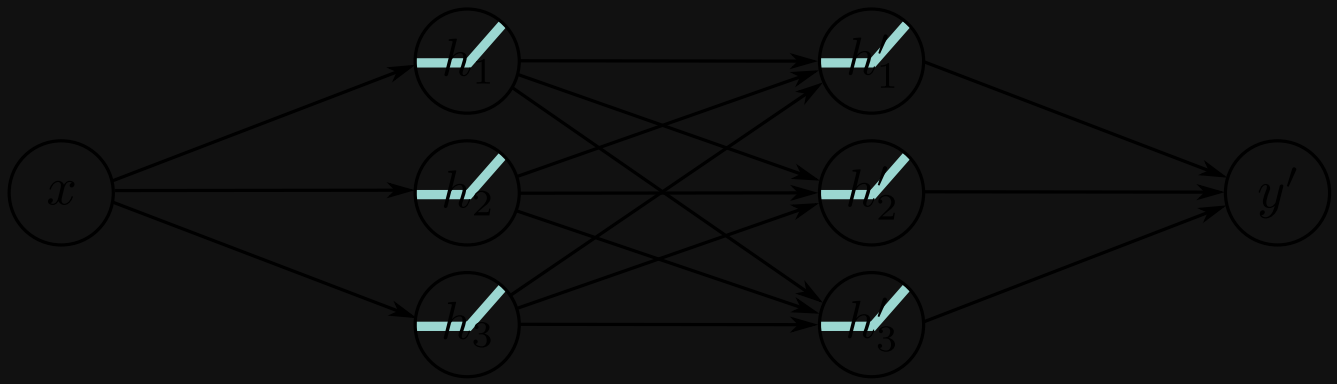
- Universal approximation theorem...
- Deep: more linear regions for same # parameters

Extra Layer is Folding

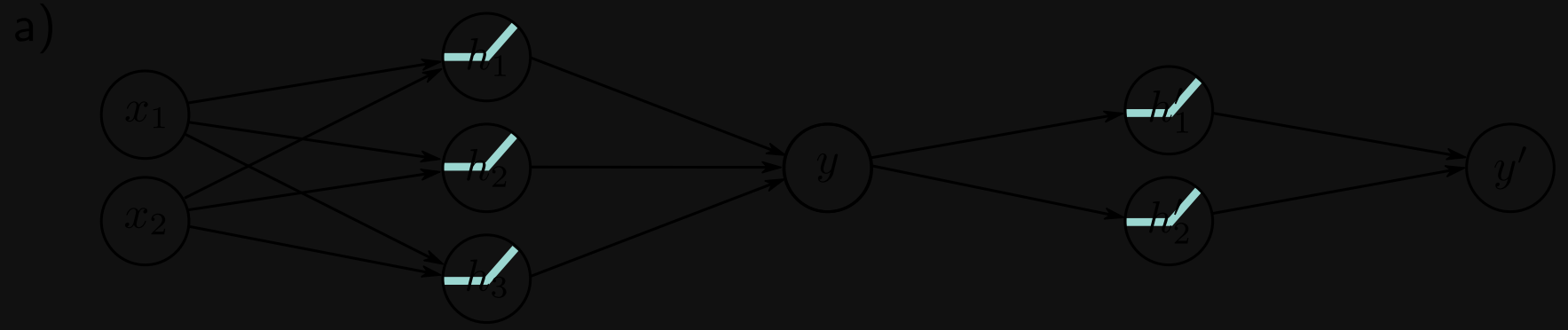


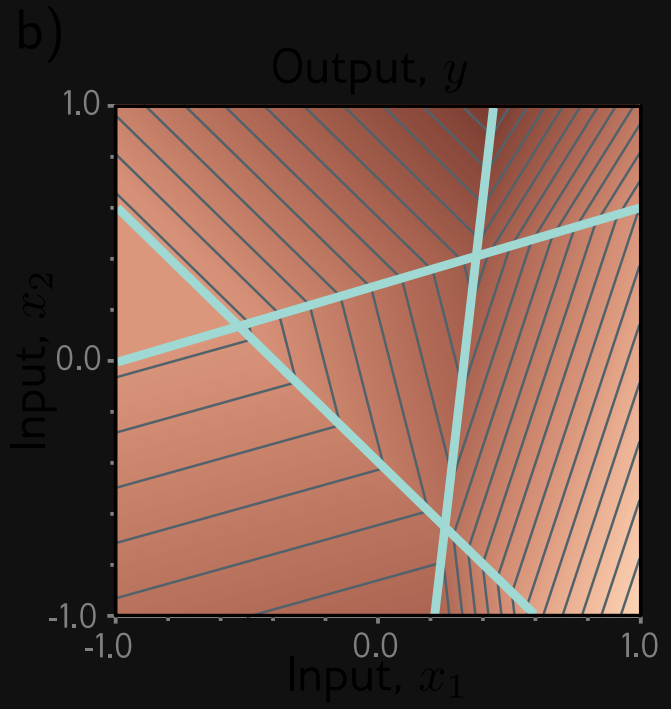
Extra Layer is Clipping (4.5)

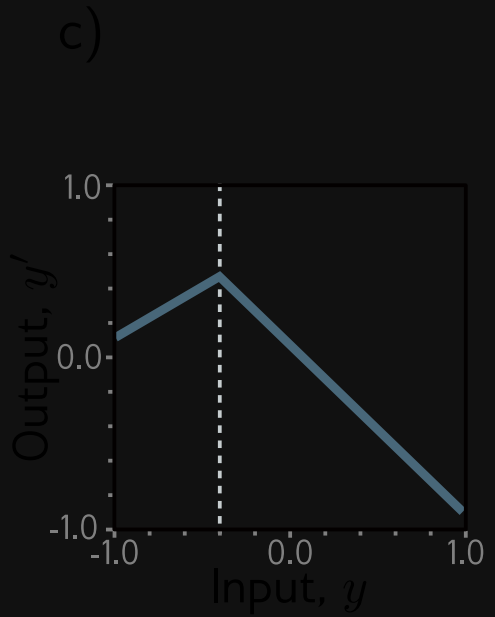
Composing as Special Case

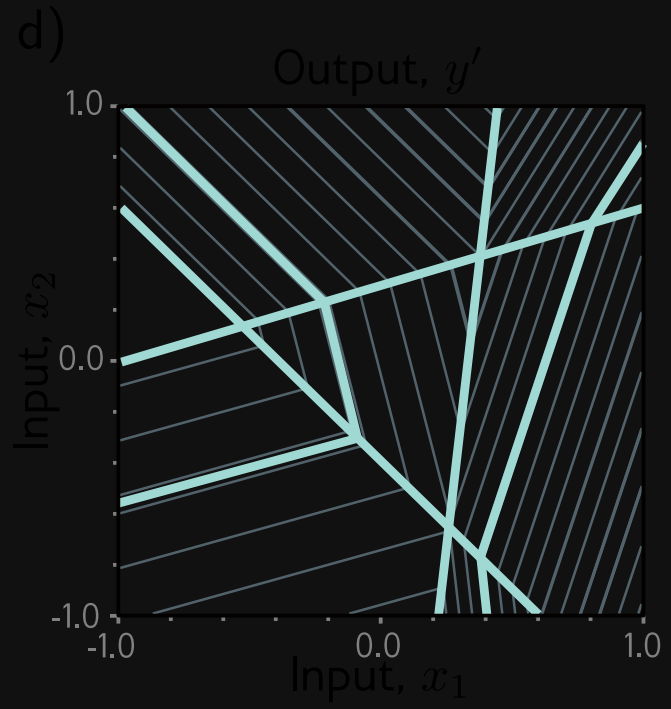


Composing in 2D



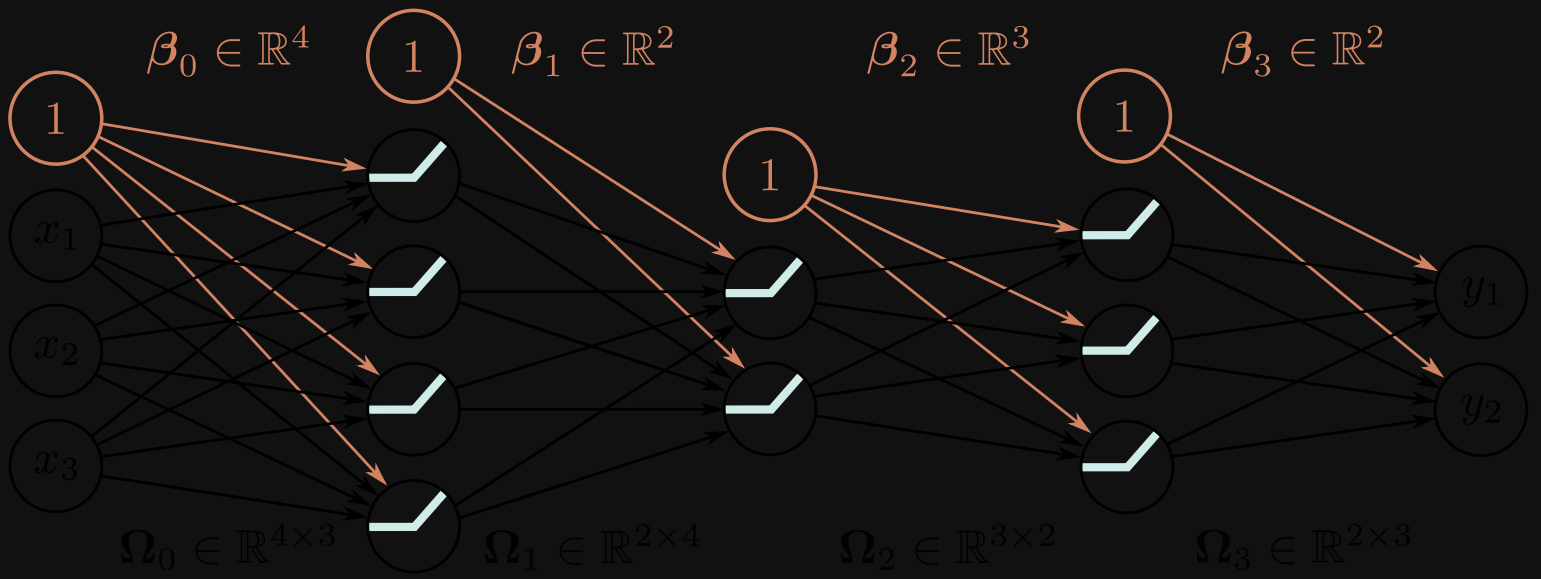




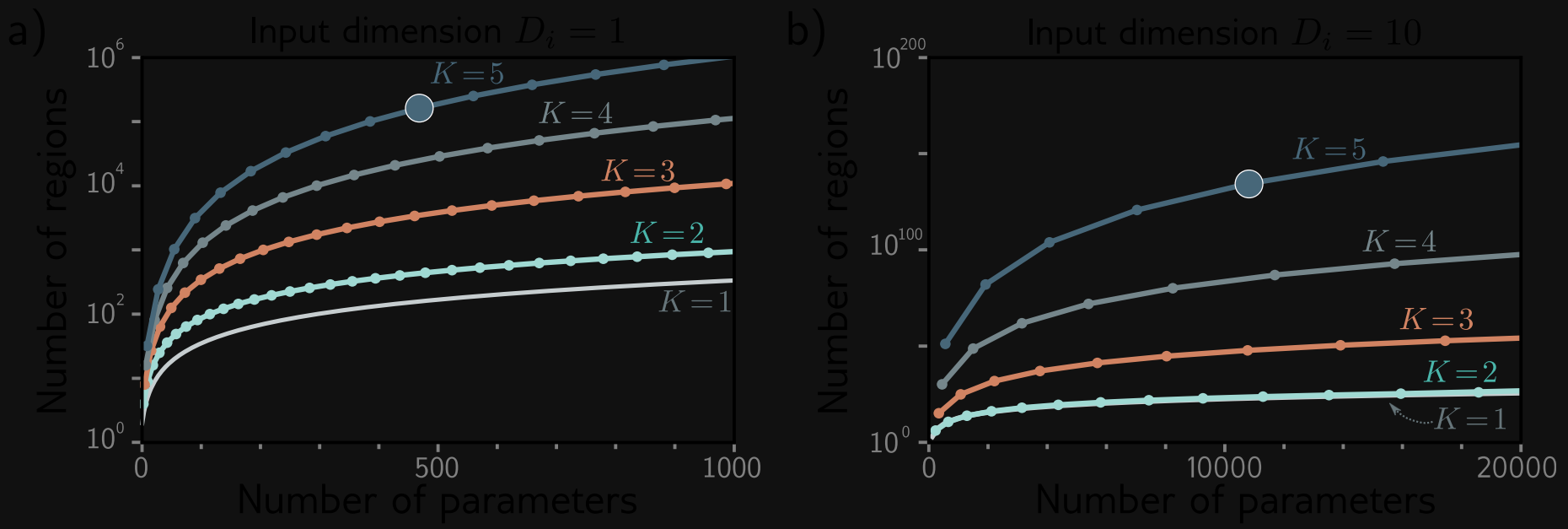


Deep Neural Network

$$\mathbf{h}_1 = \mathbf{a}(\beta_0 + \mathbf{\Omega}_0\mathbf{x})$$
$$\mathbf{h}_k = \mathbf{a}(\beta_{k-1} + \mathbf{\Omega}_{k-1}\mathbf{h}_{k-1}), \quad k = 2, \ldots, K$$
$$\mathbf{y} = \beta_K + \mathbf{\Omega}_K\mathbf{h}_K$$



Depth Efficiency



Backpropagation

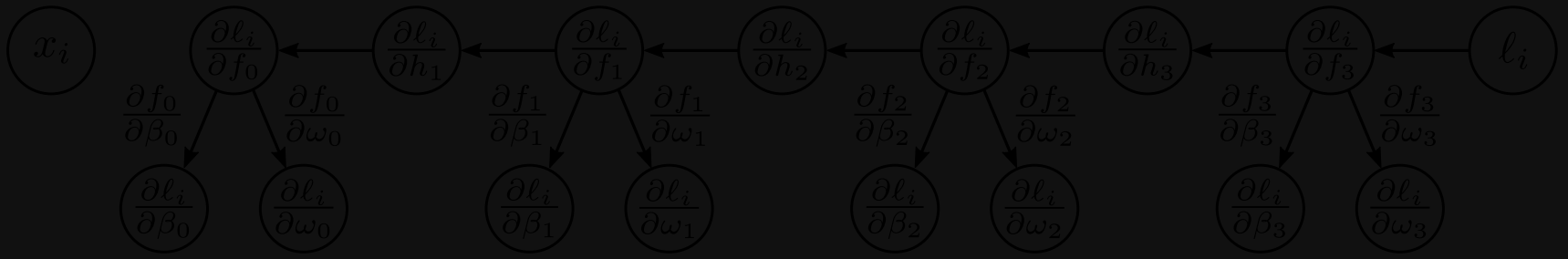
Prince (2023, chaps. 7.1–7.4)

Computing Gradients with Chain Rule

Stochastic Gradient Descent

$$\phi_{t+1} \leftarrow \phi_t - \alpha \sum_{i \in B_t} \nabla_\phi l_i(\phi_t)$$

requires computing $\frac{\partial l_i}{\partial \beta_k}$ and $\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k}$ at each layer $k \in \{0, \ldots, K\}$.

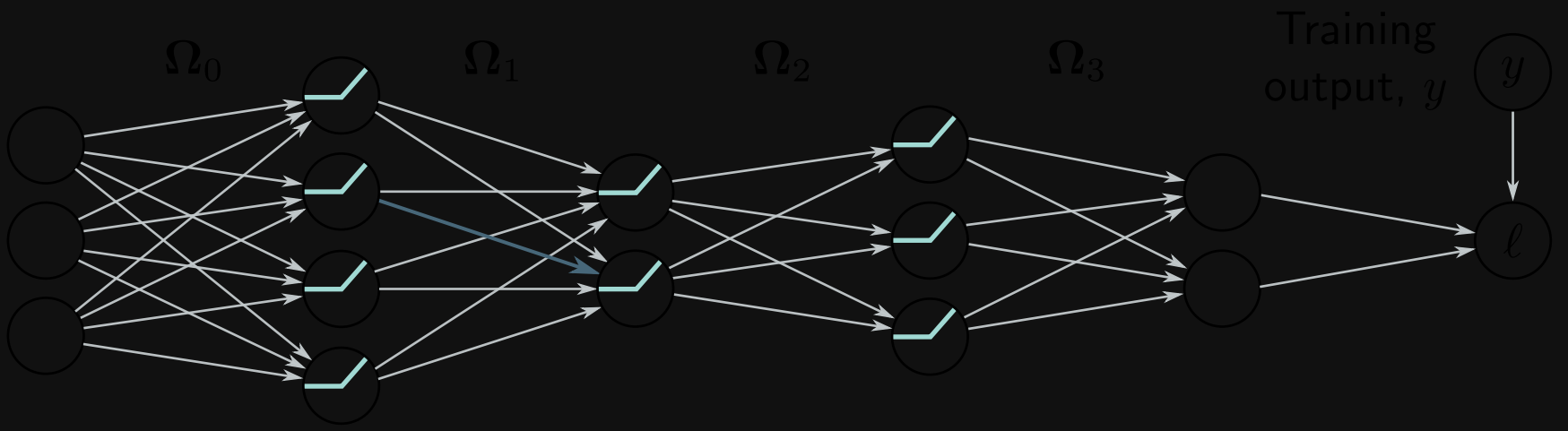


$$\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k} = \frac{\partial l_i}{\partial \mathbf{f}_k} \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\Omega}_k} = \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{f}_k} \frac{\partial \mathbf{f}_{k+1}}{\partial \mathbf{h}_{k+1}} \frac{\partial l_i}{\partial \mathbf{f}_{k+1}} \mathbf{h}_k^T$$

where activation $\mathbf{h}_{k+1}$ results from pre-activation $\mathbf{f}_k$.

Forward Pass

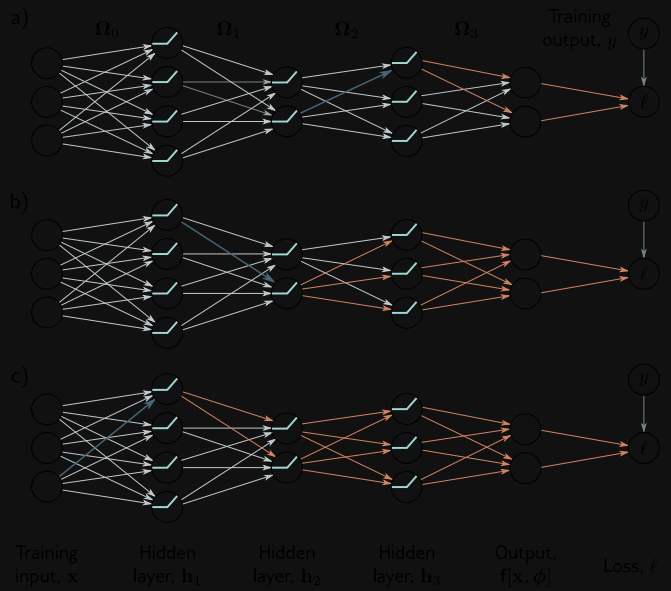
Gradients of $l_i = l(\mathbf{f}[\mathbf{x}_i, \phi], \mathbf{y}_i)$ w.r.t. weights depend on activations $\mathbf{h}_k$, which we compute first.



$$\mathbf{h}_k = \mathbf{a}(\mathbf{f}_{k-1})$$
$$\mathbf{f}_k = \beta_k + \mathbf{\Omega}_k \mathbf{h}_k$$

Backward Pass

Then compute gradients backwards using chain rule.



$$\frac{\partial l_i}{\partial \mathbf{f}_k} = \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{f}_k}\frac{\partial \mathbf{f}_{k+1}}{\partial \mathbf{h}_{k+1}}\frac{\partial l_i}{\partial \mathbf{f}_{k+1}} = \mathbb{I}[\mathbf{f}_k > 0] \odot \boldsymbol{\Omega}_{k+1}^T \frac{\partial l_i}{\partial \mathbf{f}_{k+1}}$$

$$\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k} = \frac{\partial \mathbf{f}_k}{\partial \boldsymbol{\Omega}_k}\frac{\partial l_i}{\partial \mathbf{f}_k} = \frac{\partial l_i}{\partial \mathbf{f}_k}\mathbf{h}_k^T$$

Backpropagation Algorithm

Forward pass computes activations

$$\begin{aligned}
\mathbf{f}_0 &= \beta_0 + \boldsymbol{\Omega}_0 \mathbf{x}_i \\
\mathbf{h}_k &= \mathbf{a}(\mathbf{f}_{k-1}) \qquad & k \in \{1, \ldots, K\} \\
\mathbf{f}_k &= \beta_k + \boldsymbol{\Omega}_k \mathbf{h}_k \qquad & k \in \{1, \ldots, K\}
\end{aligned}$$

Backward pass computes gradients

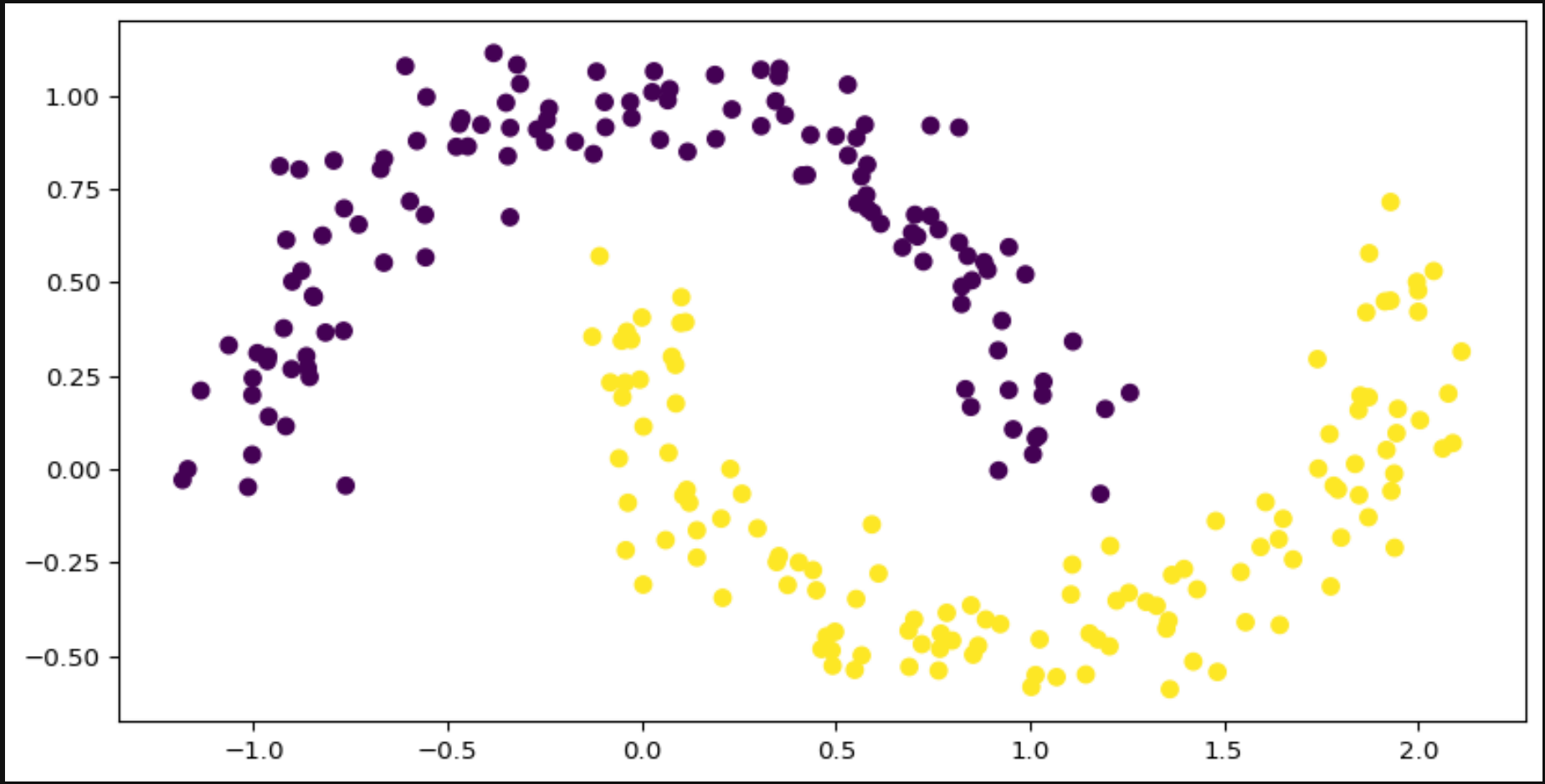
$$\begin{aligned}
\frac{\partial l_i}{\partial \beta_k} &= \frac{\partial l_i}{\partial \mathbf{f}_k} & k \in \{K, K-1, \ldots, 0\} \\
\frac{\partial l_i}{\partial \boldsymbol{\Omega}_k} &= \frac{\partial l_i}{\partial \mathbf{f}_k} \mathbf{h}_k^T & k \in \{K, K-1, \ldots, 0\} \\
\frac{\partial l_i}{\partial \mathbf{f}_{k-1}} &= \mathbb{I}[\mathbf{f}_{k-1} > 0] \odot \left(\boldsymbol{\Omega}_k^T \frac{\partial l_i}{\partial \mathbf{f}_k} \right) & k \in \{K, K-1, \ldots, 1\}
\end{aligned}$$

where $l_i = l(\mathbf{f}[\mathbf{x}_i, \phi], \mathbf{y}_i)$ and $\mathbf{h}_0 = \mathbf{x}_i$ (i.e. do for each training example in batch).[1]

. This is done in parallel across a batch, so we get tensors instead of matrices.

PyTorch Implementation

Moons Dataset



Neural Network Implementation

```python
import torch
import torch.nn as nn

class MyFirstNet(nn.Module):
  def __init__(self):
    super(MyFirstNet, self).__init__()
    self.layers = nn.Sequential(
        nn.Linear(2, 16),
        nn.ReLU(),
        nn.Linear(16, 2),
    )

  def forward(self, x):
     return self.layers(x)

  def predict(self, x):
    output = self.forward(x)
    return torch.argmax(output, 1)

  def train(self, X, y):
    loss_function = nn.CrossEntropyLoss()
    optimizer = torch.optim.SGD(model.parameters(), lr=1e-2)

    epochs = 15000
```

```
MyFirstNet(
  (layers): Sequential(
    (0): Linear(in_features=2, out_features=16, bias=True)
    (1): ReLU()
    (2): Linear(in_features=16, out_features=2, bias=True)
  )
```

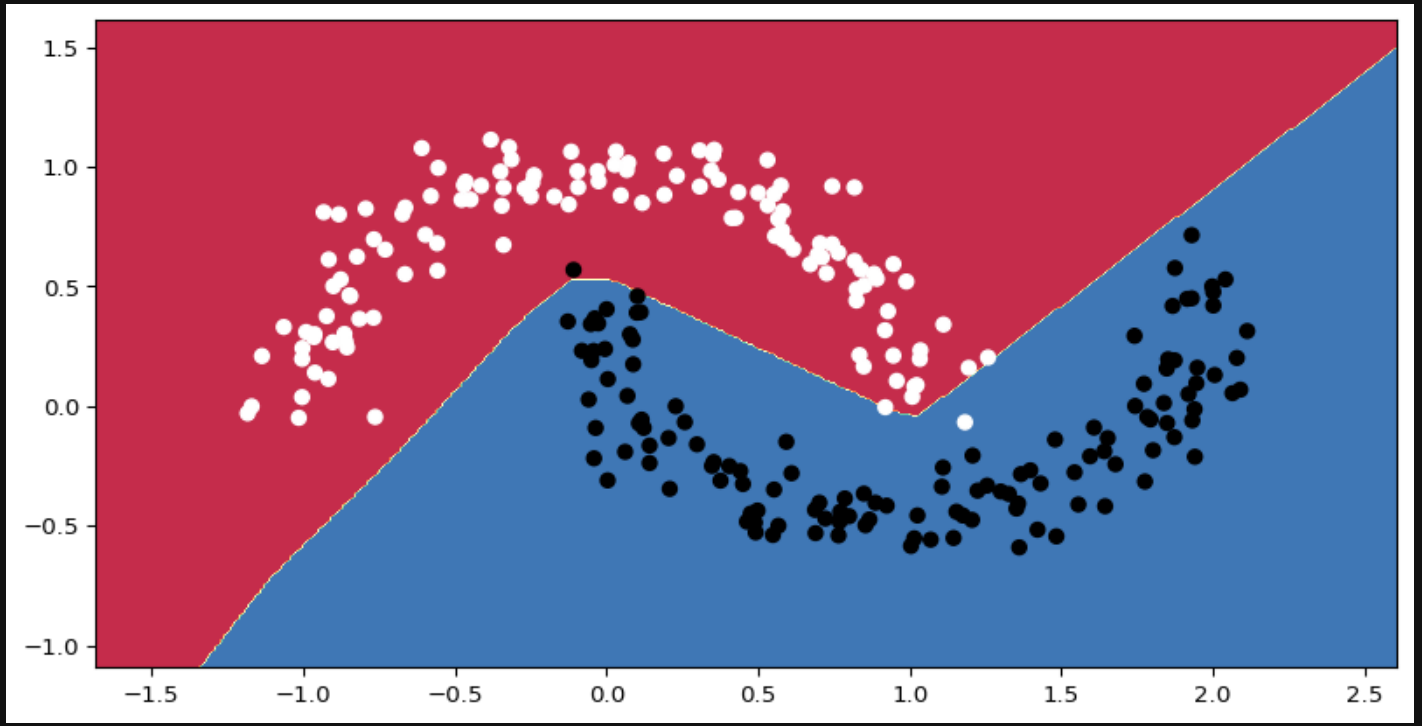
```
)
Sample input:
 tensor([[ 0.8508,  0.5047],
        [ 0.6876, -0.4327],
        [ 1.8481, -0.0702],
        [ 0.3535, -0.2334],
        [ 0.9281,  0.3966]], device='mps:0')

Network output:
 tensor([[ 0.2646, -0.0653],
        [ 0.0444,  0.1359],
        [ 0.2497, -0.0940],
```

Training

```
1 losses = model.train(X, y)
```

```
Epoch 0 loss is 0.7242650985717773
Epoch 1000 loss is 0.27691155672073364
Epoch 2000 loss is 0.24738825857639313
Epoch 3000 loss is 0.23405785858631134
Epoch 4000 loss is 0.22105270624160767
Epoch 5000 loss is 0.20637723803520203
Epoch 6000 loss is 0.18947520852088928
Epoch 7000 loss is 0.17046067118644714
Epoch 8000 loss is 0.15018796920776367
Epoch 9000 loss is 0.13044017553329468
Epoch 10000 loss is 0.11250948905944824
Epoch 11000 loss is 0.09699149429798126
Epoch 12000 loss is 0.0840226486325264
Epoch 13000 loss is 0.0734294205904007
Epoch 14000 loss is 0.06483542174100876
```


1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
2.5

Another Example

```
 1 import torch, torch.nn as nn
 2 from torch.utils.data import TensorDataset, DataLoader
 3 from torch.optim.lr_scheduler import StepLR
 4
 5 # input size, hidden layer size, output size
 6 D_i, D_k, D_o = 10, 40, 5
 7 # create model with two hidden layers
 8 model = nn.Sequential(
 9   nn.Linear(D_i, D_k),
10   nn.ReLU(),
11   nn.Linear(D_k, D_k),
12   nn.ReLU(),
13   nn.Linear(D_k, D_o))
14
15 # He initialization of weights
16 def weights_init(layer_in):
17   if isinstance(layer_in, nn.Linear):
18     nn.init.kaiming_normal_(layer_in.weight)
19     layer_in.bias.data.fill_(0.0)
20 model.apply(weights_init)
21 # choose least squares loss function
22 criterion = nn.MSELoss()
23 # construct SGD optimizer and initialize learning rate and momentum
24 optimizer = torch.optim.SGD(model.parameters(), lr = 0.1, momentum=0.9)
```

```
Epoch    99, loss 1.964
```

References

Prince, Simon J. D. 2023. *Understanding Deep Learning*. Cambridge, Massachusetts: The MIT Press.