Deep Learning Basics

An introduction to Neural Networks

Table of Contents

Resources

This lecture is heavily inspired by the opening chapters of the Understanding Deep Learning book by Simon J. D. Prince (2023).

The PDF of the book is available in its entirety through its website:
https://udlbook.github.io/udlbook/

Introduction

Deep Learning – A branch of machine learning

- In machine learning, we want to model a relationship between inputs and outputs
- Deep learning is a branch of machine learning that uses deep neural networks to this end
- We use available, historic data to learn the relationship/mapping
- The goal is to learn a mapping that yields good predictions on *new* data

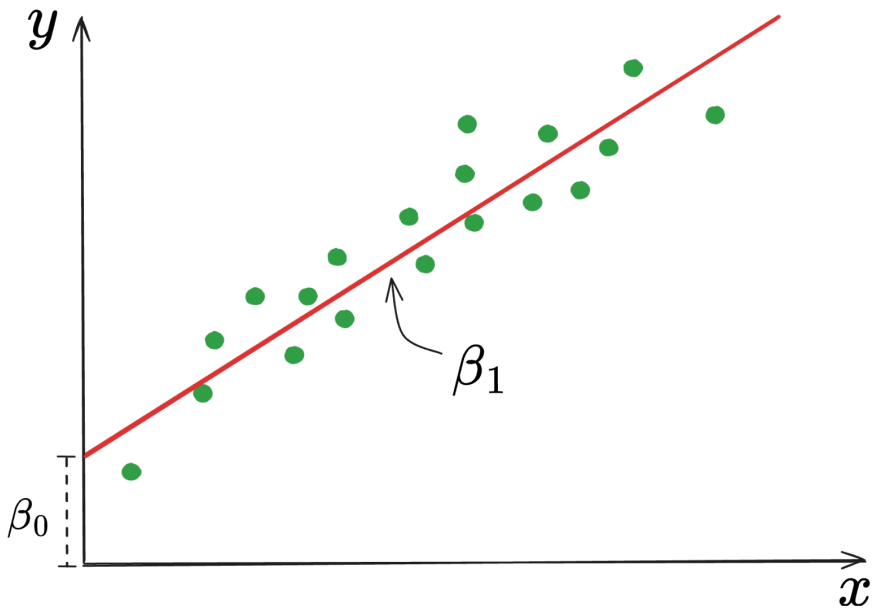
$$\underbrace{y}_{\text{output}} = f\left(\underbrace{X}_{\text{input(s)}}\right) + \epsilon$$

mapping

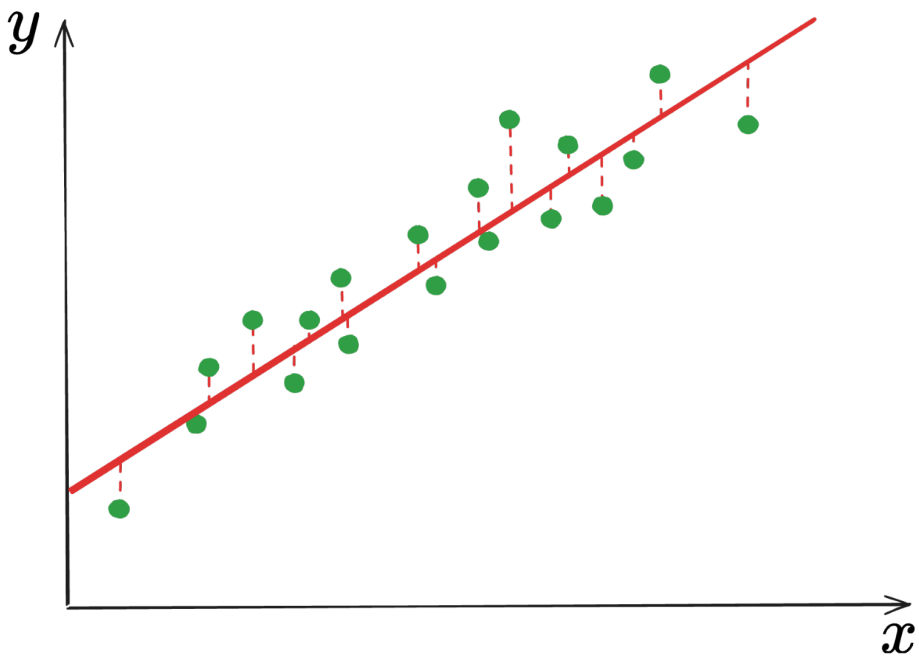
noise

Linear regression – A simple machine learning model

$$y = \underbrace{\beta_0 + \beta_1 x}_{f(x)}$$



Linear regression – Error estimates and learning model parameters



We can quantify the error as:

$$RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

The coefficients that minimize this error are given by:

$$\hat{\beta}_1 = \frac{\sum_{i=1}^{n} (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n} (x_i - \bar{x})^2}$$

$$\hat{\beta}_0 = \bar{y} - \hat{\beta}_1 \bar{x}$$

What we will look at for neural networks

- The mappings/functions that neural networks create
- How we define the error in the mapping w.r.t. available data
- How we use this definition to train the networks, i.e., learn the model parameters

Shallow Neural Networks

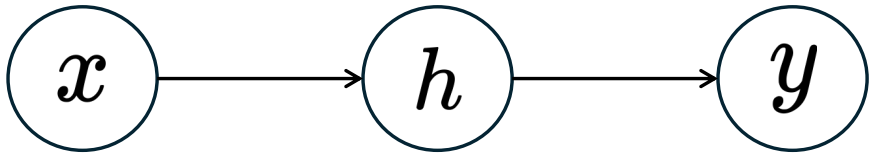
Understanding the basic building blocks

Moving from linear regression to shallow neural networks

Shallow neural networks are machine learning models that:

1. Compute linear transformations of the inputs
2. Pass the results of these transformations through an activation function
3. Forms outputs by taking a linear combination of these ‘activations’

$$y = a[\beta_0 + \beta_1 x]$$



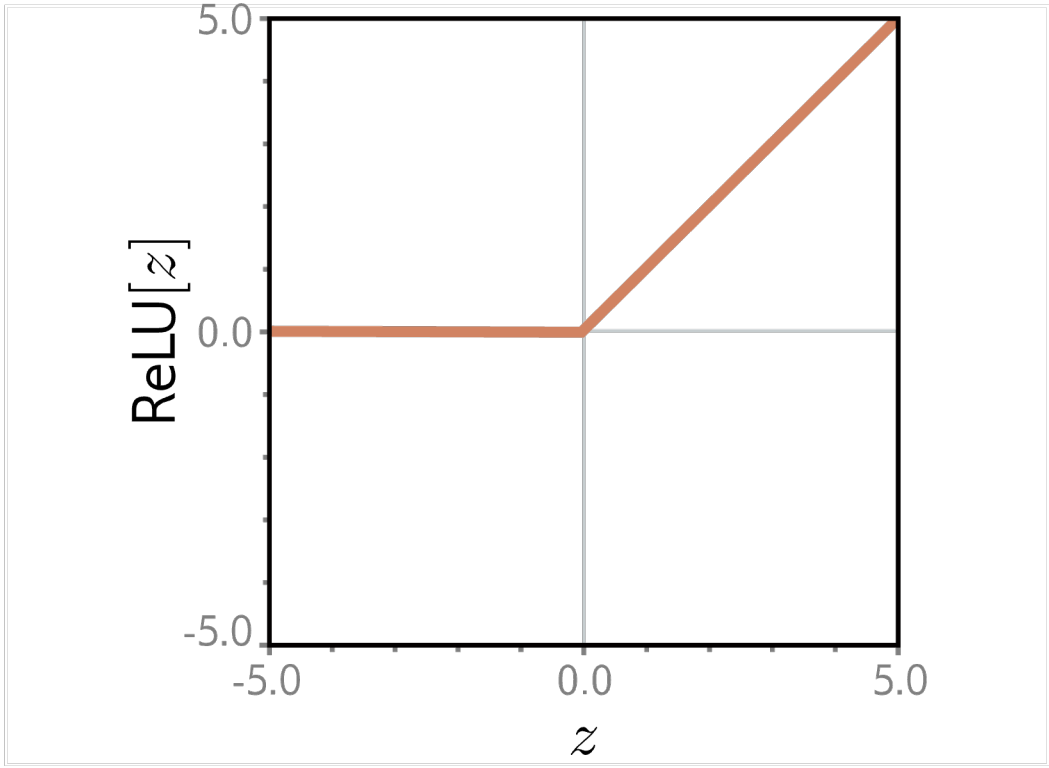
Activation functions – Introducing non-linearity

The activation functions are non-linear functions that allow the model to capture non-linear relationships in the output data.

There are many options, but the most common choice is the *rectified linear unit (ReLU)*:

$$a[z] = ReLU[z] = \begin{cases} 0 & z < 0, \\ z, & z \geq 0. \end{cases}$$

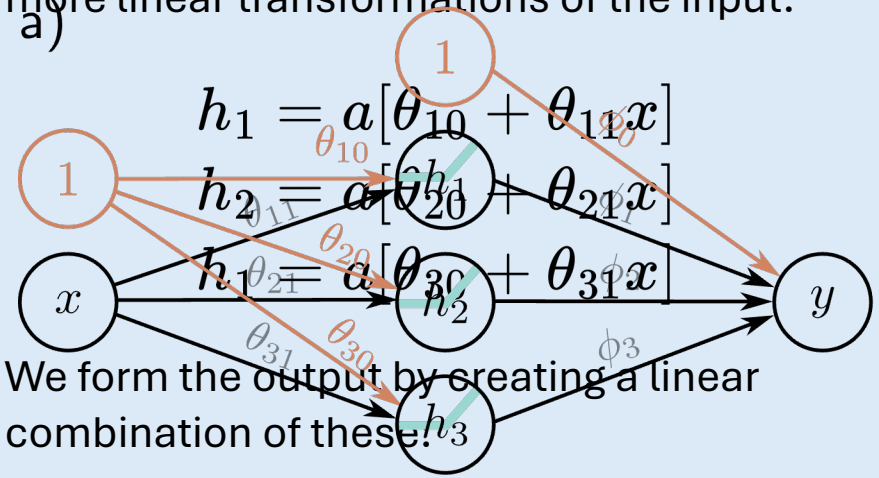
Other common choices include *sigmoid, tanh,* and variations on ReLU.



Increasing model complexity – adding more hidden units

We can add more hidden units by adding more linear transformations of the input:

a)



$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_3 = a[\theta_{30} + \theta_{31}x]$$

We form the output by creating a linear combination of these:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

b)

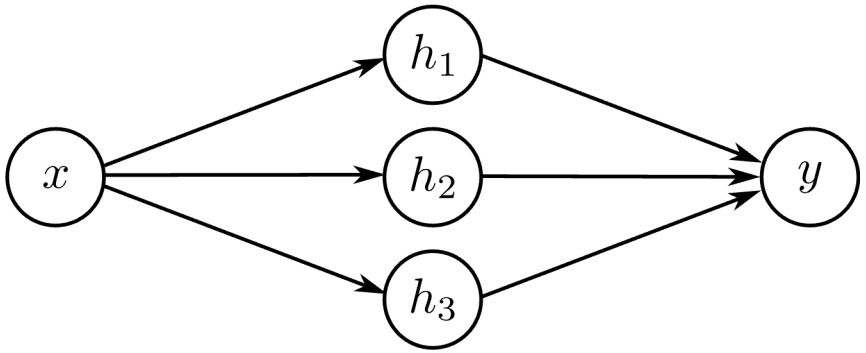
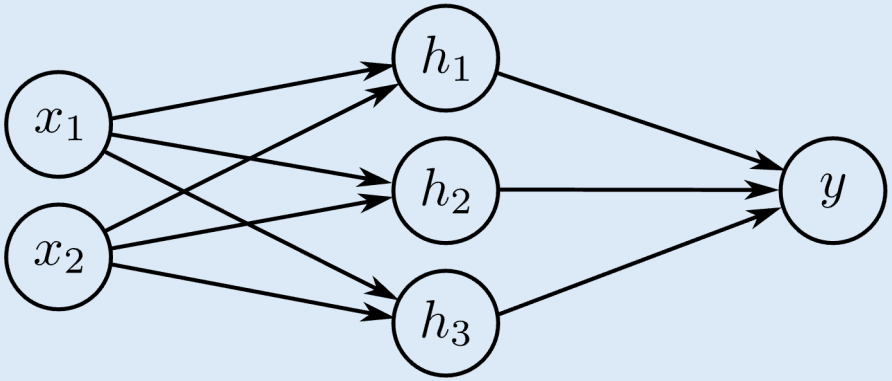


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Reflection: What happens when we have more than one input or output?



Extending shallow neural networks to take multiple inputs/outputs is quite straightforward.

For multiple inputs, the hidden units would then be activations applied to linear combinations of these inputs, e.g.:

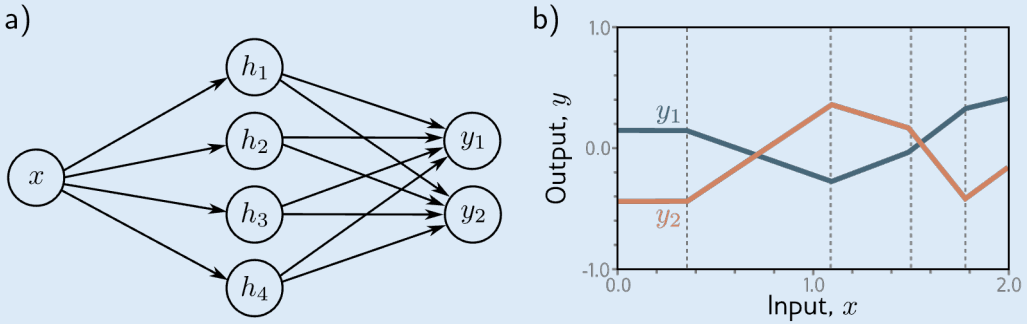
$$h_1 = a[\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2]$$
$$h_2 = a[\theta_{20} + \theta_{21}x_1 + \theta_{22}x_2]$$
$$h_3 = a[\theta_{30} + \theta_{31}x_1 + \theta_{32}x_2]$$

The output would still be formed as:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Reflection: What happens when we have more than one input or output?



For network with two outputs, we could, e.g., have hidden units:

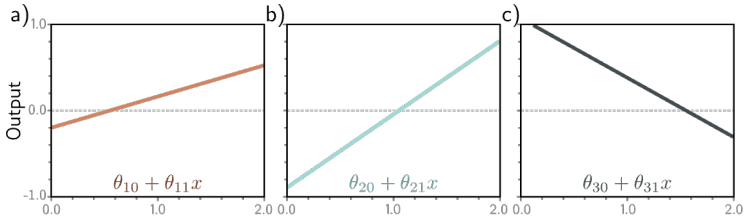
$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_3 = a[\theta_{30} + \theta_{31}x]$$
$$h_4 = a[\theta_{40} + \theta_{41}x]$$

These could then be used to create two different linear combinations, forming the two outputs:

$$y_1 = \phi_{10} + \phi_{11}h_1 + \phi_{12}h_2 + \phi_{13}h_3 + \phi_{14}h_4$$
$$y_2 = \phi_{20} + \phi_{21}h_1 + \phi_{22}h_2 + \phi_{23}h_3 + \phi_{24}h_4$$

Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

The effect of the ReLU activations – Forming piecewise linear functions



We can add more hidden units by adding more linear transformations of the input:

$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_1 = a[\theta_{30} + \theta_{31}x]$$

We form the output by creating a linear combination of these:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

The effect of the ReLU activations – Forming piecewise linear functions

We can add more hidden units by adding more linear transformations of the input:

$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_1 = a[\theta_{30} + \theta_{31}x]$$

We form the output by creating a linear combination of these:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

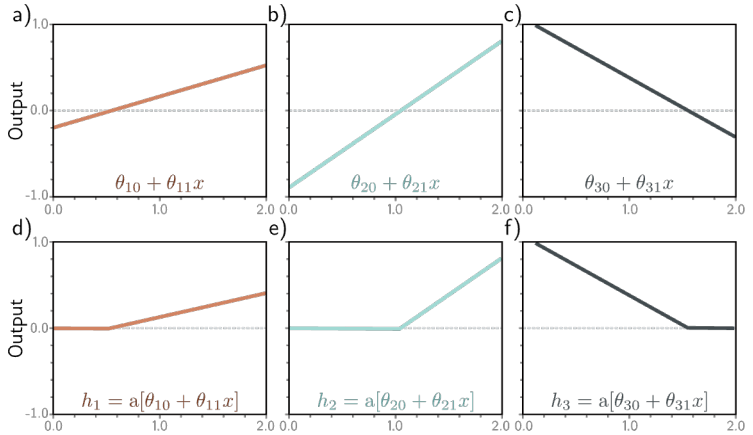


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

The effect of the ReLU activations – Forming piecewise linear functions

We can add more hidden units by adding more linear transformations of the input:

$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_1 = a[\theta_{30} + \theta_{31}x]$$

We form the output by creating a linear combination of these:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

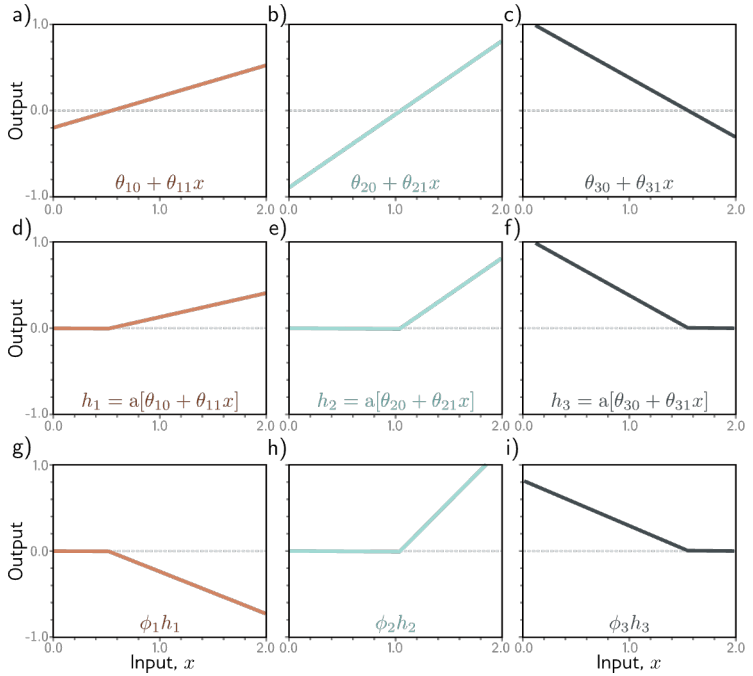


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

The effect of the ReLU activations – Forming piecewise linear functions

We can add more hidden units by adding more linear transformations of the input:

$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_1 = a[\theta_{30} + \theta_{31}x]$$

We form the output by creating a linear combination of these:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

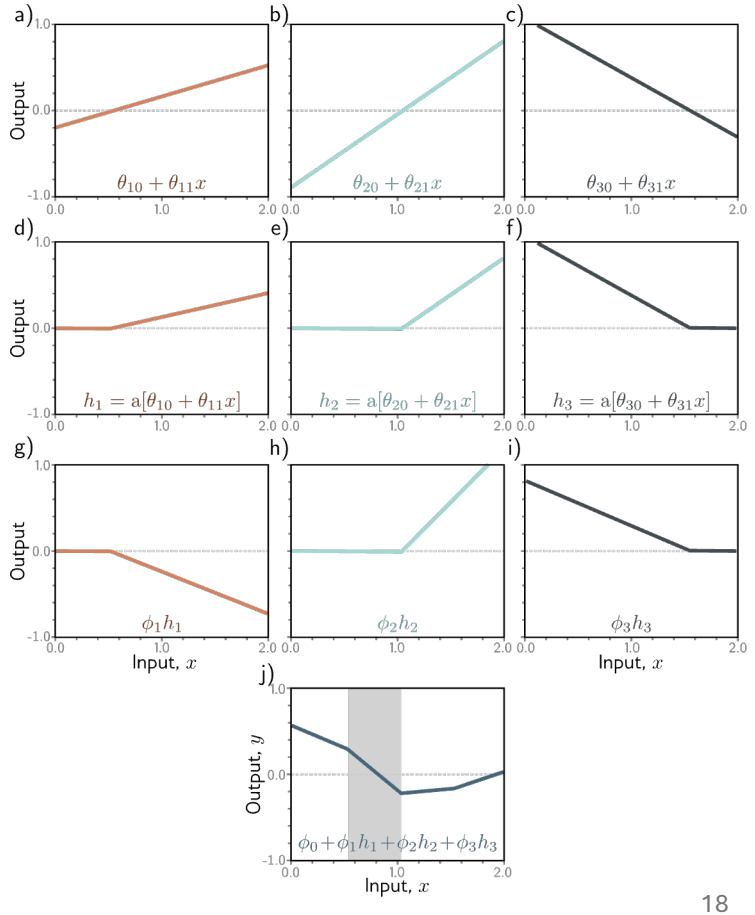
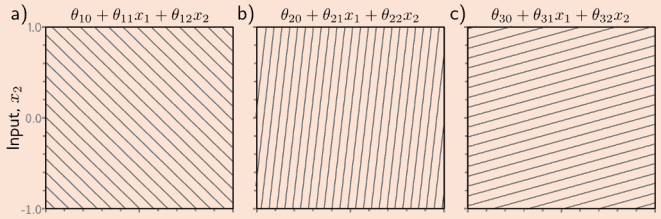


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Code example 1: Simple neural network

```python
# Define the neural network
class ShallowNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(1, 3)
        self.output = nn.Linear(3, 1)

    def forward(self, x):
        pre_act = self.hidden(x)
        hidden = F.relu(pre_act)
        output = self.output(hidden)
        return output, pre_act, hidden

# Create an instance of the network
shallow_model = ShallowNN()

# Example input
x = torch.tensor([[1.0]], dtype=torch.float32)

# Forward pass
output, *_ = shallow_model(x)
print(f"Input shape: {x.shape}")
print(f"Output shape: {output.shape}")
print(f"Output value: {output.item()}")
```

Digression: Activations for multiple inputs



Extending shallow neural networks to take multiple inputs/outputs is quite straightforward.

For multiple inputs, the hidden units would then be activations applied to linear combinations of these inputs, e.g.:

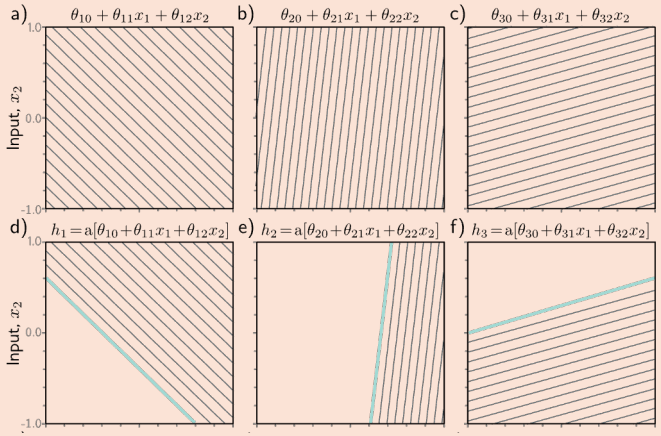
$$h_1 = a[\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2]$$
$$h_2 = a[\theta_{20} + \theta_{21}x_1 + \theta_{22}x_2]$$
$$h_3 = a[\theta_{30} + \theta_{31}x_1 + \theta_{32}x_2]$$

The output would still be formed as:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Digression: Activations for multiple inputs



Extending shallow neural networks to take multiple inputs/outputs is quite straightforward.

For multiple inputs, the hidden units would then be activations applied to linear combinations of these inputs, e.g.:

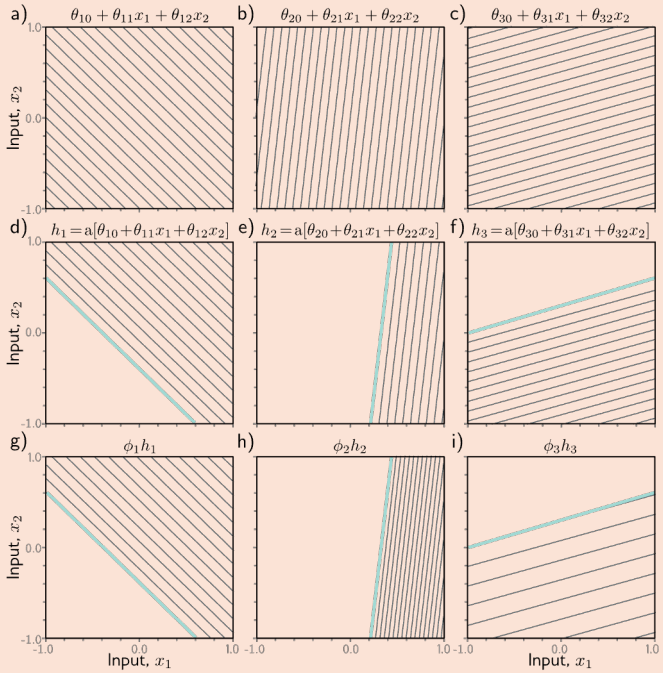
$$h_1 = a[\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2]$$
$$h_2 = a[\theta_{20} + \theta_{21}x_1 + \theta_{22}x_2]$$
$$h_3 = a[\theta_{30} + \theta_{31}x_1 + \theta_{32}x_2]$$

The output would still be formed as:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Digression: Activations for multiple inputs



Extending shallow neural networks to take multiple inputs/outputs is quite straightforward.

For multiple inputs, the hidden units would then be activations applied to linear combinations of these inputs, e.g.:

$$h_1 = a[\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2]$$
$$h_2 = a[\theta_{20} + \theta_{21}x_1 + \theta_{22}x_2]$$
$$h_3 = a[\theta_{30} + \theta_{31}x_1 + \theta_{32}x_2]$$

The output would still be formed as:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Digression: Activations for multiple inputs

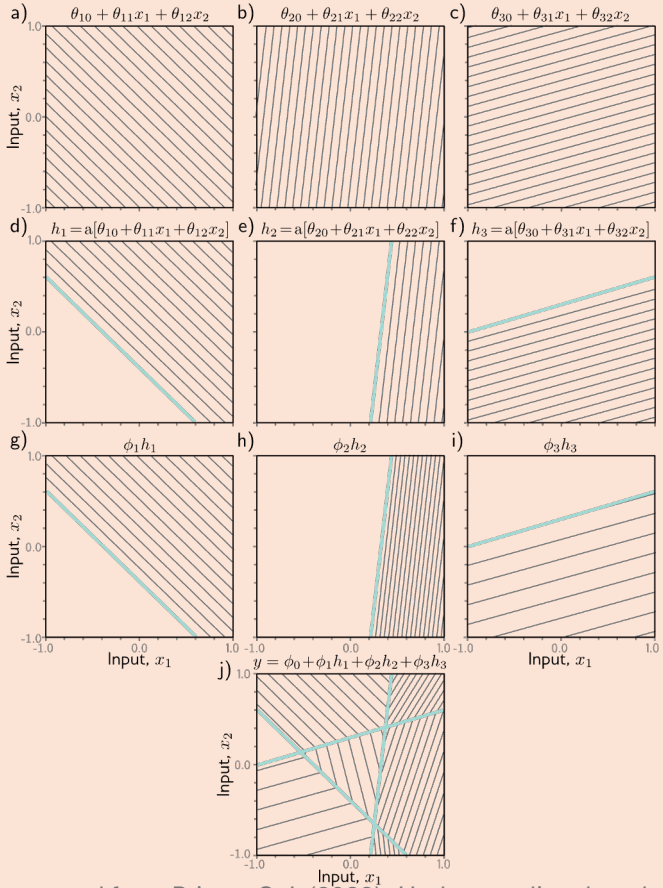


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Extending shallow neural networks to take multiple inputs/outputs is quite straightforward.

For multiple inputs, the hidden units would then be activations applied to linear combinations of these inputs, e.g.:

$$h_1 = a[\theta_{10} + \theta_{11}x_1 + \theta_{12}x_2]$$
$$h_2 = a[\theta_{20} + \theta_{21}x_1 + \theta_{22}x_2]$$
$$h_3 = a[\theta_{30} + \theta_{31}x_1 + \theta_{32}x_2]$$

The output would still be formed as:

$$y = \phi_0 + \phi_1 h_1 + \phi_2 h_2 + \phi_3 h_3$$

The Universal Approximation Theorem

We can extend shallow neural networks to have an arbitrary number of hidden units. In a network with *D* hidden units, we can describe the hidden units as:

$$h_d = a[\theta_{d0} + \theta_{d1}x]$$

And the network output as:

$$y = \phi_0 + \sum_{d=1}^{D} \phi_d h_d$$

The *universal approximation theorem* states that for any continuous function, there exists a shallow network that can approximate it.

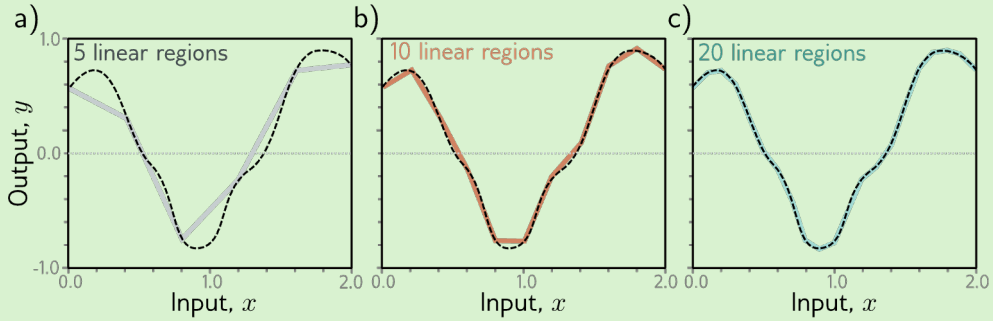


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Reflection: Parameter counts and linear regions

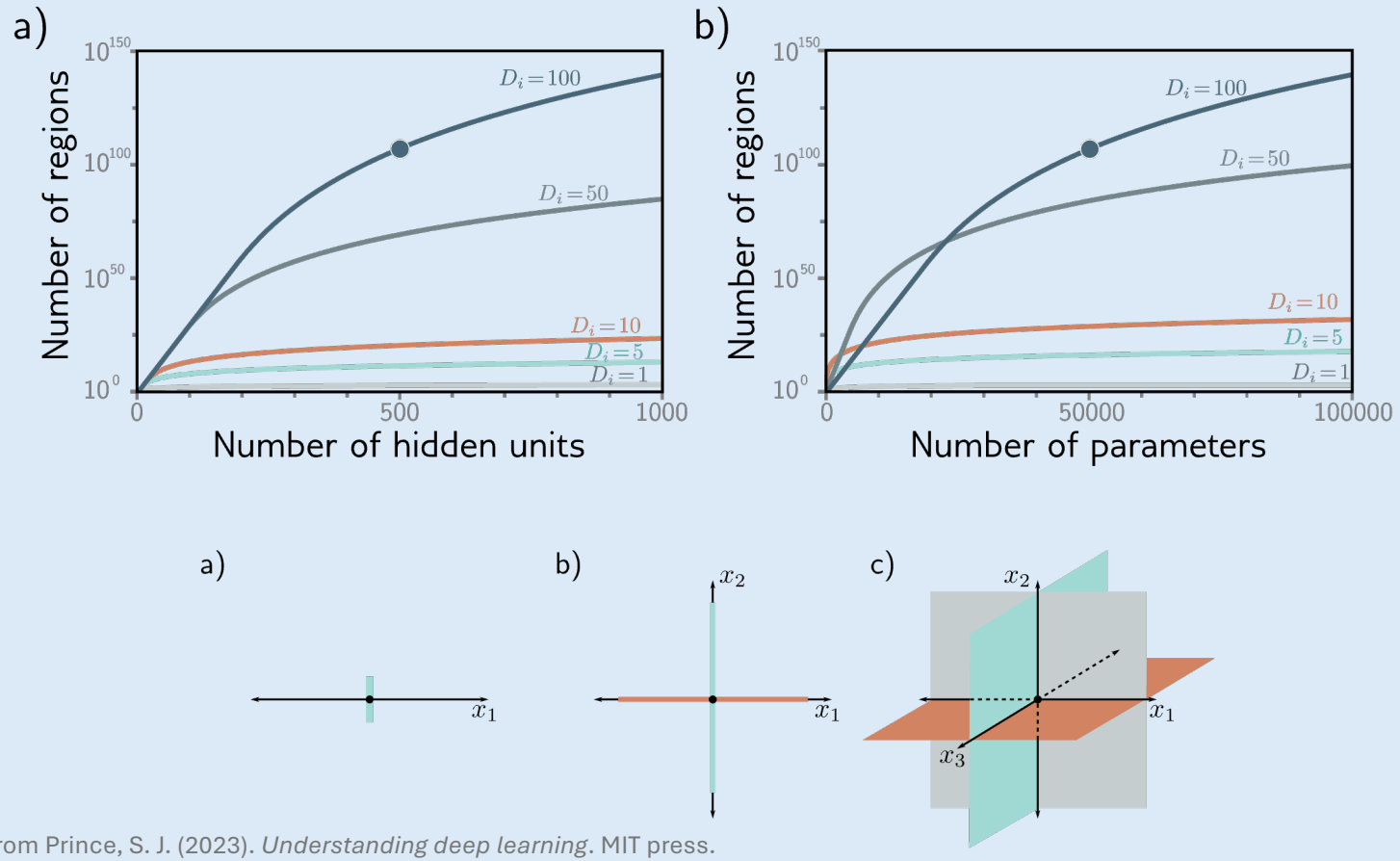


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Deep Networks

Adding layers to make networks deep

From shallow to deep – Adding another network

A simple way to extend our shallow network would be to pass the output through a new network.

We would then pass the output of the first network to new hidden units:

$$h'_1 = a[\theta'_{10} + \theta'_{11}y]$$
$$h'_2 = a[\theta'_{20} + \theta'_{21}y]$$
$$h'_3 = a[\theta'_{30} + \theta'_{31}y],$$

And create a final, new output:

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3.$$

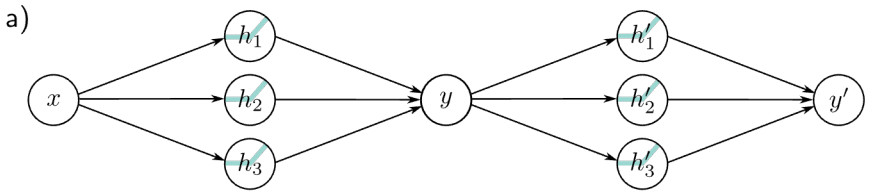


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

From shallow to deep – Changing perspective

If we insert the original expression for the output of the first network into the expression we had for the new hidden units, we get:

$$
\begin{aligned}
h'_1 &= a[\theta'_{10} + \theta'_{11}y] = a[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\
h'_2 &= a[\theta'_{20} + \theta'_{21}y] = a[\theta'_{20} + \theta'_{21}\phi_0 + \theta'_{21}\phi_1 h_1 + \theta'_{21}\phi_2 h_2 + \theta'_{21}\phi_3 h_3] \\
h'_3 &= a[\theta'_{30} + \theta'_{31}y] = a[\theta'_{30} + \theta'_{31}\phi_0 + \theta'_{31}\phi_1 h_1 + \theta'_{31}\phi_2 h_2 + \theta'_{31}\phi_3 h_3],
\end{aligned}
$$

From shallow to deep – Changing perspective

If we insert the original expression for the output of the first network into the expression we had for the new hidden units, we get:

$$
\begin{aligned}
h'_1 &= a[\theta'_{10} + \theta'_{11}y] = a[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\
h'_2 &= a[\theta'_{20} + \theta'_{21}y] = a[\theta'_{20} + \theta'_{21}\phi_0 + \theta'_{21}\phi_1 h_1 + \theta'_{21}\phi_2 h_2 + \theta'_{21}\phi_3 h_3] \\
h'_3 &= a[\theta'_{30} + \theta'_{31}y] = a[\theta'_{30} + \theta'_{31}\phi_0 + \theta'_{31}\phi_1 h_1 + \theta'_{31}\phi_2 h_2 + \theta'_{31}\phi_3 h_3],
\end{aligned}
$$

This can be rewritten as

$$
\begin{aligned}
h'_1 &= a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= a[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= a[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3]
\end{aligned}
$$

From shallow to deep – Changing perspective

If we insert the original expression for the output of the first network into the expression we had for the new hidden units, we get:

$$
\begin{aligned}
h'_1 &= a[\theta'_{10} + \theta'_{11}y] = a[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\
h'_2 &= a[\theta'_{20} + \theta'_{21}y] = a[\theta'_{20} + \theta'_{21}\phi_0 + \theta'_{21}\phi_1 h_1 + \theta'_{21}\phi_2 h_2 + \theta'_{21}\phi_3 h_3] \\
h'_3 &= a[\theta'_{30} + \theta'_{31}y] = a[\theta'_{30} + \theta'_{31}\phi_0 + \theta'_{31}\phi_1 h_1 + \theta'_{31}\phi_2 h_2 + \theta'_{31}\phi_3 h_3],
\end{aligned}
$$

This can be rewritten as

$$
\begin{aligned}
h'_1 &= a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= a[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= a[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3]
\end{aligned}
$$

From shallow to deep – Changing perspective

If we insert the original expression for the output of the first network into the expression we had for the new hidden units, we get:

$$
\begin{aligned}
h'_1 &= a[\theta'_{10} + \theta'_{11}y] = a[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\
h'_2 &= a[\theta'_{20} + \theta'_{21}y] = a[\theta'_{20} + \theta'_{21}\phi_0 + \theta'_{21}\phi_1 h_1 + \theta'_{21}\phi_2 h_2 + \theta'_{21}\phi_3 h_3] \\
h'_3 &= a[\theta'_{30} + \theta'_{31}y] = a[\theta'_{30} + \theta'_{31}\phi_0 + \theta'_{31}\phi_1 h_1 + \theta'_{31}\phi_2 h_2 + \theta'_{31}\phi_3 h_3],
\end{aligned}
$$

This can be rewritten as

$$
\begin{aligned}
h'_1 &= a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= a[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= a[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3]
\end{aligned}
$$

From shallow to deep – Changing perspective

If we insert the original expression for the output of the first network into the expression we had for the new hidden units, we get:

$$
\begin{aligned}
h'_1 &= a[\theta'_{10} + \theta'_{11}y] = a[\theta'_{10} + \theta'_{11}\phi_0 + \theta'_{11}\phi_1 h_1 + \theta'_{11}\phi_2 h_2 + \theta'_{11}\phi_3 h_3] \\
h'_2 &= a[\theta'_{20} + \theta'_{21}y] = a[\theta'_{20} + \theta'_{21}\phi_0 + \theta'_{21}\phi_1 h_1 + \theta'_{21}\phi_2 h_2 + \theta'_{21}\phi_3 h_3] \\
h'_3 &= a[\theta'_{30} + \theta'_{31}y] = a[\theta'_{30} + \theta'_{31}\phi_0 + \theta'_{31}\phi_1 h_1 + \theta'_{31}\phi_2 h_2 + \theta'_{31}\phi_3 h_3],
\end{aligned}
$$

This can be rewritten as

$$
\begin{aligned}
h'_1 &= a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\
h'_2 &= a[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\
h'_3 &= a[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3]
\end{aligned}
$$

From shallow to deep – Changing perspective

We can now define a deep network with two layers, where the first hidden layer is:

$$h_1 = a[\theta_{10} + \theta_{11}x]$$
$$h_2 = a[\theta_{20} + \theta_{21}x]$$
$$h_1 = a[\theta_{30} + \theta_{31}x]$$

The second hidden layer is:

$$h'_1 = a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3]$$
$$h'_2 = a[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3]$$
$$h'_3 = a[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3]$$

And the final output is:

$$y' = \phi'_0 + \phi'_1 h'_1 + \phi'_2 h'_2 + \phi'_3 h'_3$$

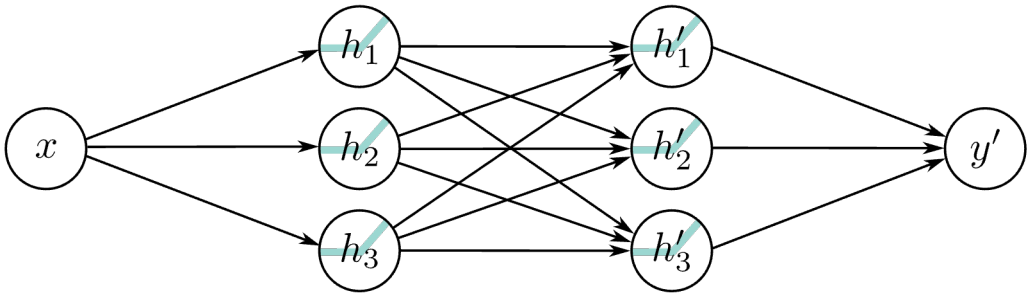


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Reflection: How does the second layer modify the output of the first layer?

Reflection: How does the second layer modify the output of the first layer?

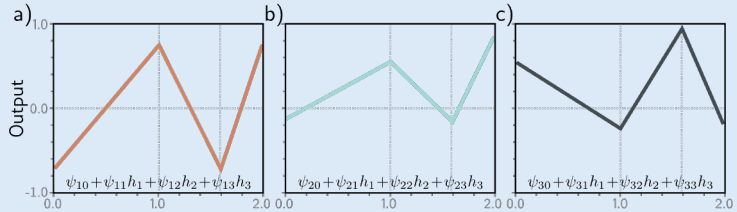


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Code example 2: Deep neural network

```python
# Define the neural network
class ShallowNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(1, 3)
        self.output = nn.Linear(3, 1)

    def forward(self, x):
        pre_act = self.hidden(x)
        hidden = F.relu(pre_act)
        output = self.output(hidden)
        return output, pre_act, hidden

# Create an instance of the network
shallow_model = ShallowNN()

# Example input
x = torch.tensor([[1.0]], dtype=torch.float32)

# Forward pass
output, *_ = shallow_model(x)
print(f"Input shape: {x.shape}")
print(f"Output shape: {output.shape}")
print(f"Output value: {output.item()}")
```

Digression: Matrix notation and GPUs

Using matrix notation, we can the hidden layers in our two-layer network as:

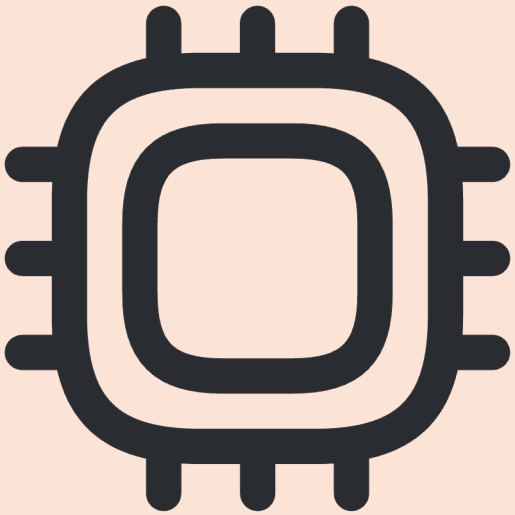
$$\begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix} = a\left[\begin{bmatrix} \theta_{10} \\ \theta_{20} \\ \theta_{30} \end{bmatrix} + \begin{bmatrix} \theta_{11} \\ \theta_{21} \\ \theta_{31} \end{bmatrix} x\right]$$

$$\begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix} = a\left[\begin{bmatrix} \psi_{10} \\ \psi_{20} \\ \psi_{30} \end{bmatrix} + \begin{bmatrix} \psi_{11} & \psi_{12} & \psi_{13} \\ \psi_{21} & \psi_{22} & \psi_{23} \\ \psi_{31} & \psi_{32} & \psi_{33} \end{bmatrix} \begin{bmatrix} h_1 \\ h_2 \\ h_3 \end{bmatrix}\right]$$

And the output as:

$$y' = \phi'_0 + \begin{bmatrix} \phi'_1 & \phi'_2 & \phi'_3 \end{bmatrix} \begin{bmatrix} h'_1 \\ h'_2 \\ h'_3 \end{bmatrix}$$

Digression: Matrix notation and GPUs



We write this even more compactly as:

$$\mathbf{h} = \mathbf{a}[\boldsymbol{\theta}_0 + \boldsymbol{\theta}x]$$
$$\mathbf{h}' = \mathbf{a}[\boldsymbol{\psi}_0 + \boldsymbol{\Psi}\mathbf{h}]$$
$$y' = \phi'_0 + \boldsymbol{\phi}'\mathbf{h}'$$

GPUs are highly optimized for doing matrix operations in parallel. This is why they are so effective at speeding up the computations needed to do deep learning.

In general, deep networks can handle any number of inputs, outputs, layers and hidden units

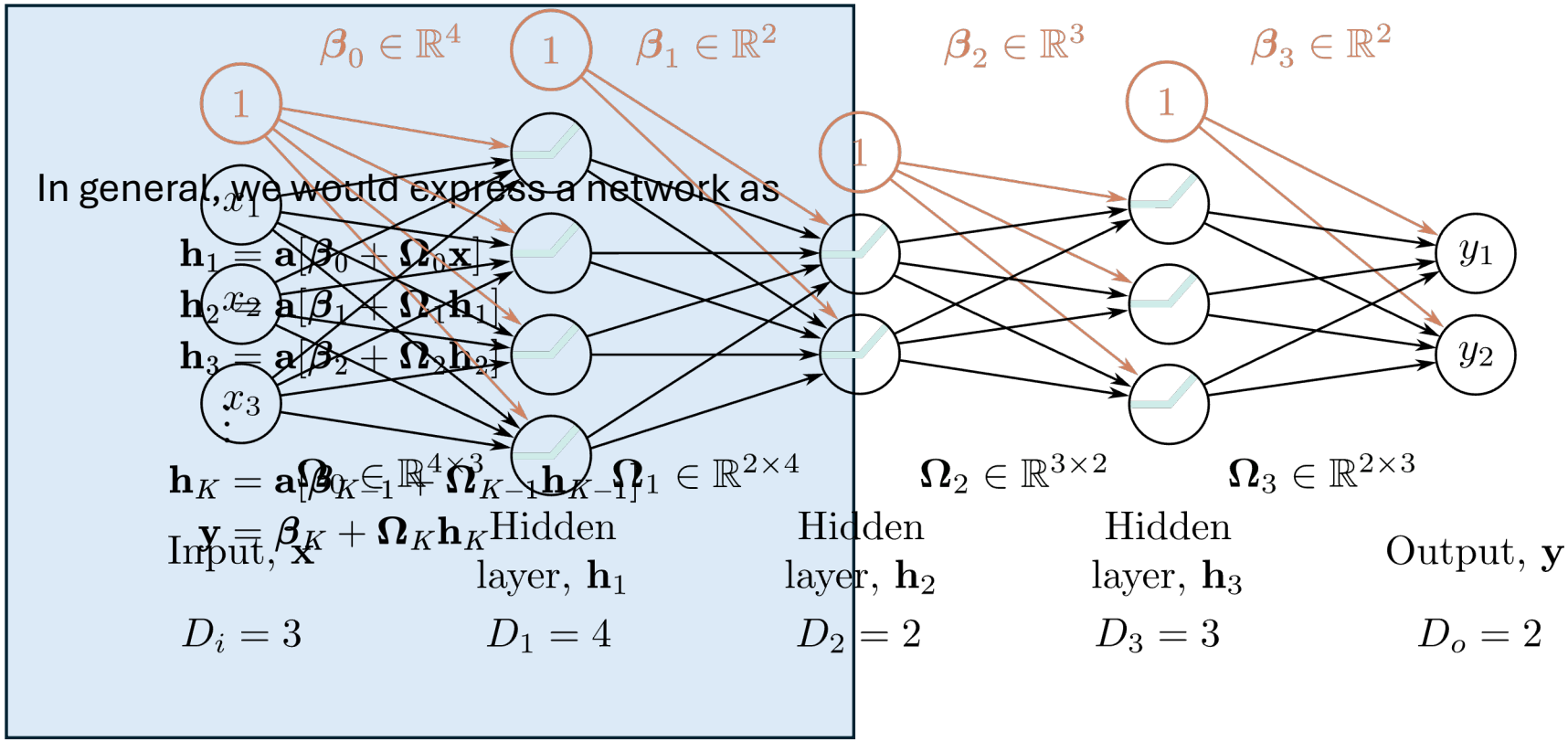


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Deep networks – Parameter counts and number of linear regions

A network with *one input,* K layers and D (>2) hidden units in every layer would have a parameter count of

$$3D + 1 + (K-1)D(D+1)$$

Such a network could create a function with

$$(D+1)^K$$

linear regions.

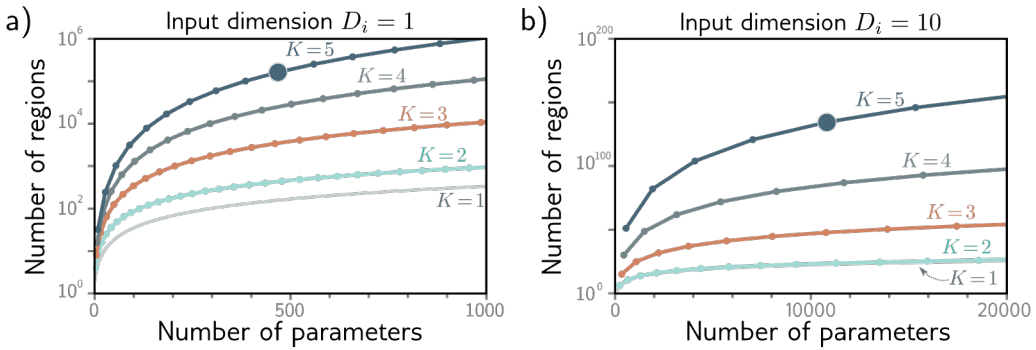


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Reflection: Why choose deep networks when the universal approximation theorem holds for shallow networks?

- There are functions for which shallow networks need exponentially more hidden units than deep networks to approximate them
- More advanced architectures that make more efficient use of the available parameters are easier to specify with multiple layers
- It is often easier to train (moderately) deep networks than shallow ones
- Deep networks typically generalize better to new data than shallow networks

Training neural networks

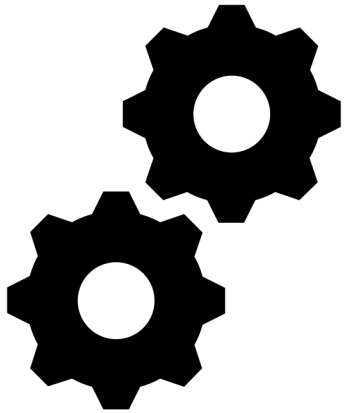
Loss functions, gradients and optimization

Training neural networks – Tuning parameters until the model is ‘good’

On a high level, training neural networks consists of two steps:

1. Defining a what a good model means for our problem through a *loss function*

2. Using an optimization algorithm to find parameter values that minimize the loss function



What is a loss function?

A loss function is a function of the model parameters that describes the mismatch between the model predictions and their associated ground truth values.

It is typically formulated such that a smaller loss reflects better model performance.

Thus, once we have defined our loss function, we train our model by seeking parameter values that *minimize* the loss.

$$\hat{\phi} = \underset{\phi}{\operatorname{argmin}}[L[\phi]]$$

The maximum likelihood approach for creating loss functions

A common framework for creating loss functions is the maximum likelihood approach.

Instead of thinking of the model as directly predicting an output **y** given an input **x**, we shift perspective, and treat the model as predicting a *conditional distribution* over possible outputs, given the input: $Pr(\mathbf{y}|\mathbf{x})$

Given this perspective, we want the loss function to encourage a high probability for the training outputs, i.e., we maximize $Pr(\mathbf{y_i}|\mathbf{x_i})$ across all samples.

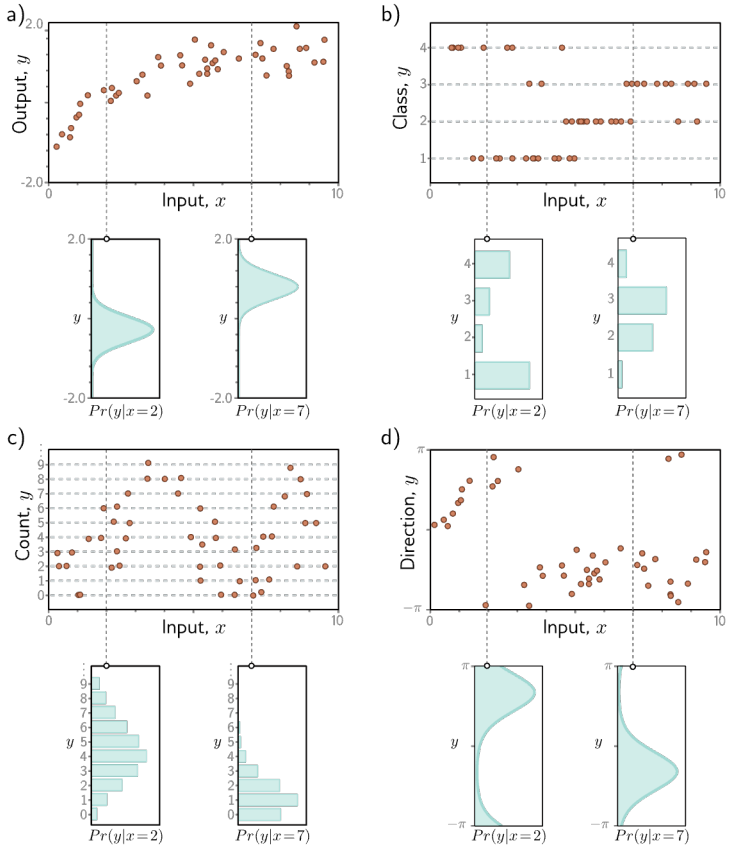


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

The maximum likelihood approach – Predicting distribution parameters

Under this framework, we set the network to predict the parameters of the conditional distribution, i.e., we set

$$\boldsymbol{\theta}_i = \mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$$

For the conditional distribution $Pr(\mathbf{y}_i|\boldsymbol{\theta}_i)$. We then find model parameters

$$\begin{aligned}\hat{\boldsymbol{\phi}} &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\prod_{i=1}^{I}\Pr(\mathbf{y}_i|\mathbf{x}_i)\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\prod_{i=1}^{I}\Pr(\mathbf{y}_i|\boldsymbol{\theta}_i)\right] \\ &= \underset{\boldsymbol{\phi}}{\operatorname{argmax}}\left[\prod_{i=1}^{I}\Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}])\right].\end{aligned}$$

The maximum likelihood approach – Negative log-likelihood

Since the probabilities in the maximum likelihood criterion often become small, and thus their product can become hard to represent with finite precision arithmetic in computers, we take the logarithm of the previous expression.

Additionally, since model fitting is framed as a minimization problem by convention, we multiply by negative one. Thus, we get

$$\begin{aligned}\hat{\phi} &= \underset{\phi}{\operatorname{argmin}}\left[-\sum_{i=1}^{I}\log\left[\Pr(\mathbf{y}_i|\mathbf{f}[\mathbf{x}_i,\phi])\right]\right] \\ &= \underset{\phi}{\operatorname{argmin}}[L[\phi]]\end{aligned}$$

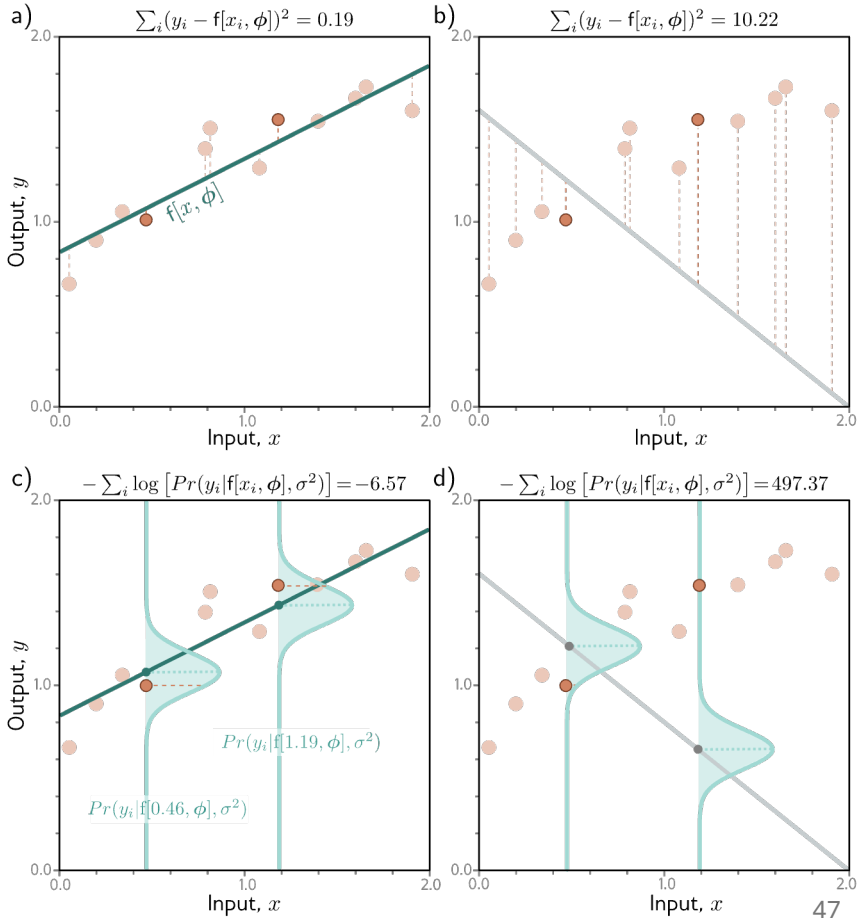
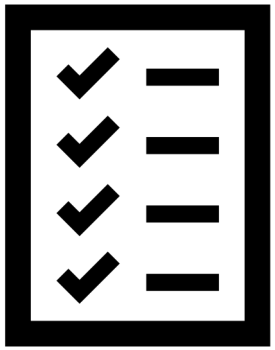


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

The maximum likelihood approach – A recipe for creating loss functions

1. Define a suitable probability distribution over the output space; $Pr(\mathbf{y}|\boldsymbol{\theta})$
2. Set the model to predict the parameters of this distribution; $\boldsymbol{\theta} = \mathbf{f}[\mathbf{x}, \phi]$
3. Find the model parameters that best fit the data by minimizing the negative log-likelihood loss function

For predicting on a new sample $\mathbf{x}$, we either return the full distribution $Pr(\mathbf{y}|\mathbf{f}[\mathbf{x}, \hat{\phi}])$, or its maximum.



Example: Binary classification – The output space

In binary classification, we try to predict whether an input belongs to one of two classes. Our output space can therefore be defined as $y \in \{0, 1\}$

To construct a loss function for this problem, we begin by defining a suitable probability distribution over the output space.

The most natural choice for binary classification is the Bernoulli distribution:

$$Pr(y|\lambda) = \begin{cases} 1-\lambda & y = 0 \\ \lambda & y = 1 \end{cases}$$

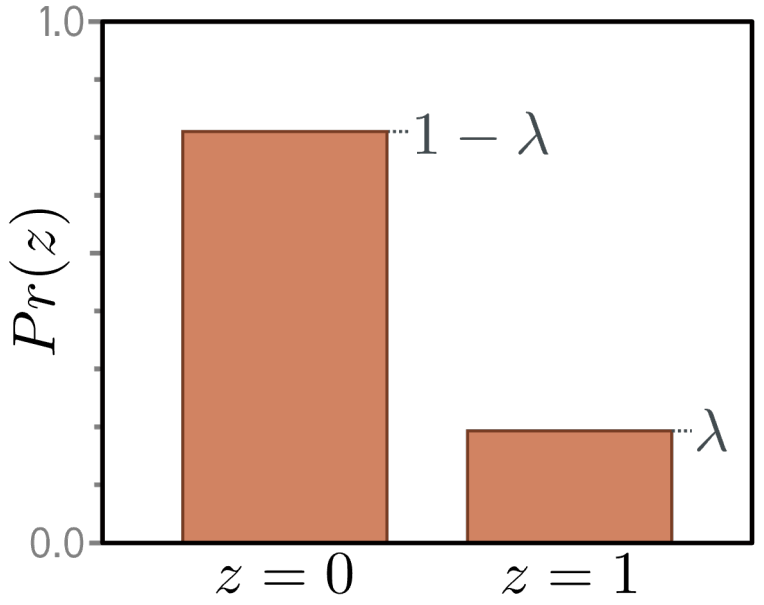


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Example: Binary classification – The sigmoid function

To ensure that our network output can describe such a probability, we need to force the it to be between 0 and 1.

To this end, we employ the sigmoid function:

$$\text{sig}[z] = \frac{1}{1+\exp[-z]}$$

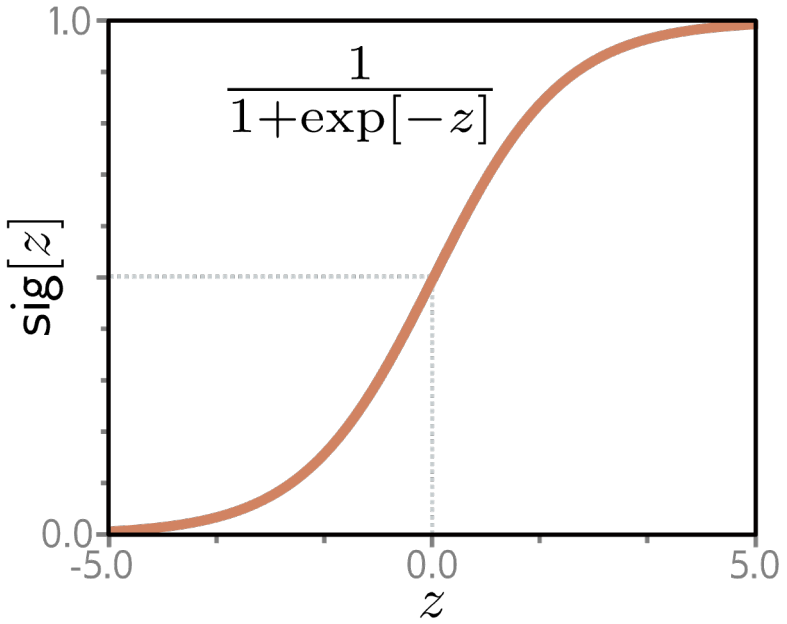


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Example: Binary classification – The loss function

We can rewrite the Bernoulli probability distribution as

$$Pr(y|\lambda) = (1-\lambda)^{1-y} \cdot \lambda^{y}$$

With this rewrite, we can define the likelihood as

$$Pr(y|\mathbf{x}) = (1-\text{sig}[\mathbf{f}[\mathbf{x},\phi]])^{1-y} \cdot \text{sig}[\mathbf{f}[\mathbf{x},\phi]]^{y}$$

Finally, we take the negative logarithm to obtain the negative log-likelihood:

$$L[\phi] = \sum_{i=1}^{I} -(1-y_i)\log\left[1-\text{sig}[\mathbf{f}[\mathbf{x}_i,\phi]]\right] - y_i \log\left[\text{sig}[\mathbf{f}[\mathbf{x}_i,\phi]]\right]$$

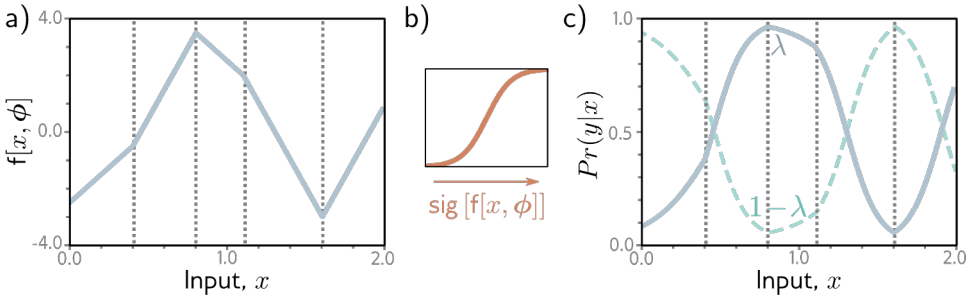


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

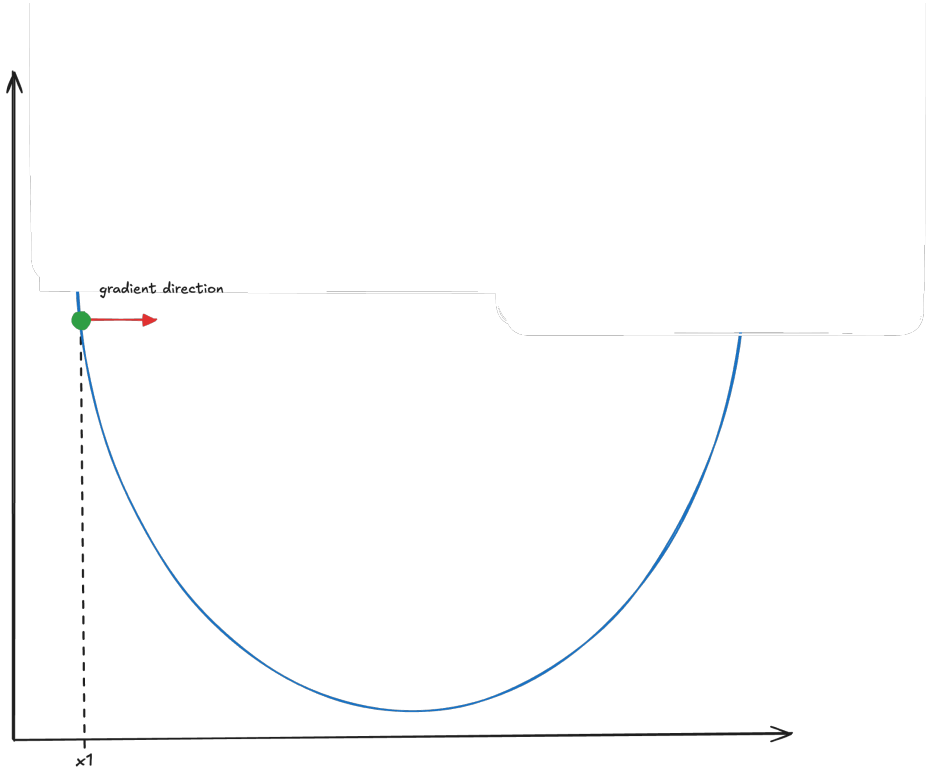
Code example 3: Binary cross-entropy

```python
# Define the neural network
class ShallowNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(1, 3)
        self.output = nn.Linear(3, 1)

    def forward(self, x):
        pre_act = self.hidden(x)
        hidden = F.relu(pre_act)
        output = self.output(hidden)
        return output, pre_act, hidden

# Create an instance of the network
shallow_model = ShallowNN()

# Example input
x = torch.tensor([[1.0]], dtype=torch.float32)

# Forward pass
output, *_ = shallow_model(x)
print(f"Input shape: {x.shape}")
print(f"Output shape: {output.shape}")
print(f"Output value: {output.item()}")
```

Fitting models – Gradient descent

We have defined the loss function, and now need to find the parameter values that minimize it.

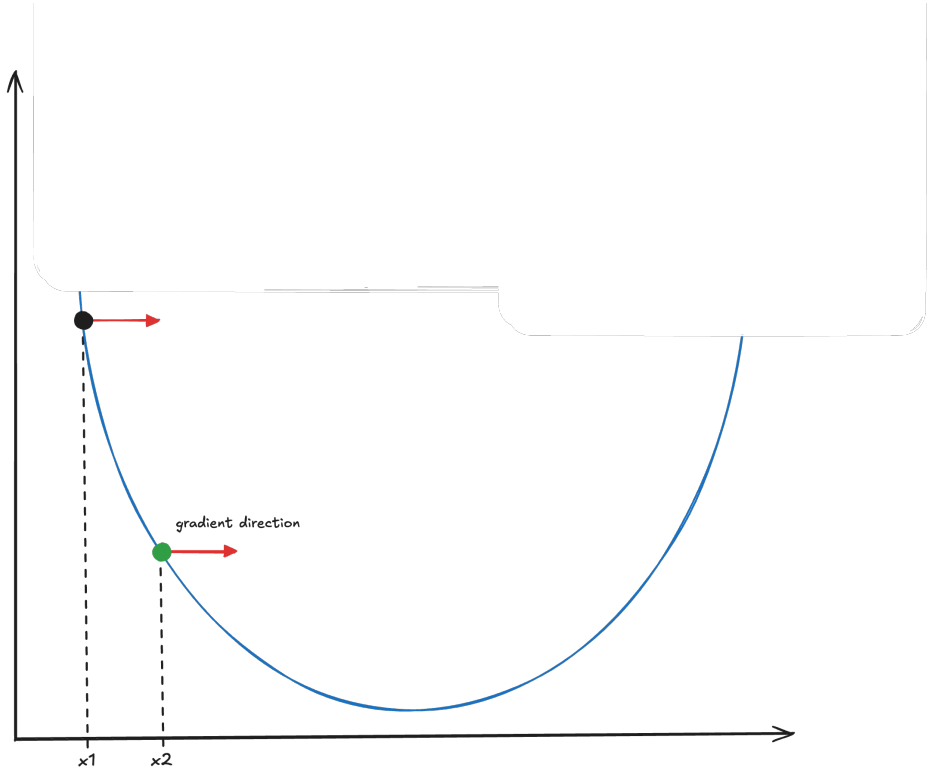
Gradient descent is an *optimization algorithm* that uses *gradient* of the loss function to minimize it.



Fitting models – Gradient descent

We have defined the loss function, and now need to find the parameter values that minimize it.

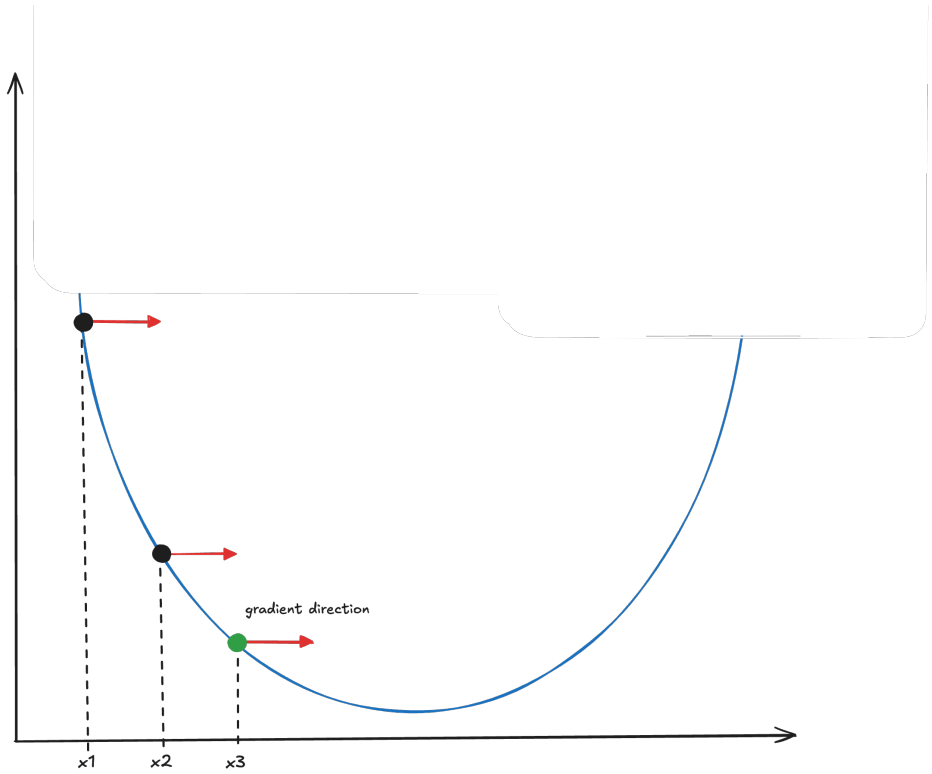
Gradient descent is an *optimization algorithm* that uses *gradient* of the loss function to minimize it.



Fitting models – Gradient descent

We have defined the loss function, and now need to find the parameter values that minimize it.

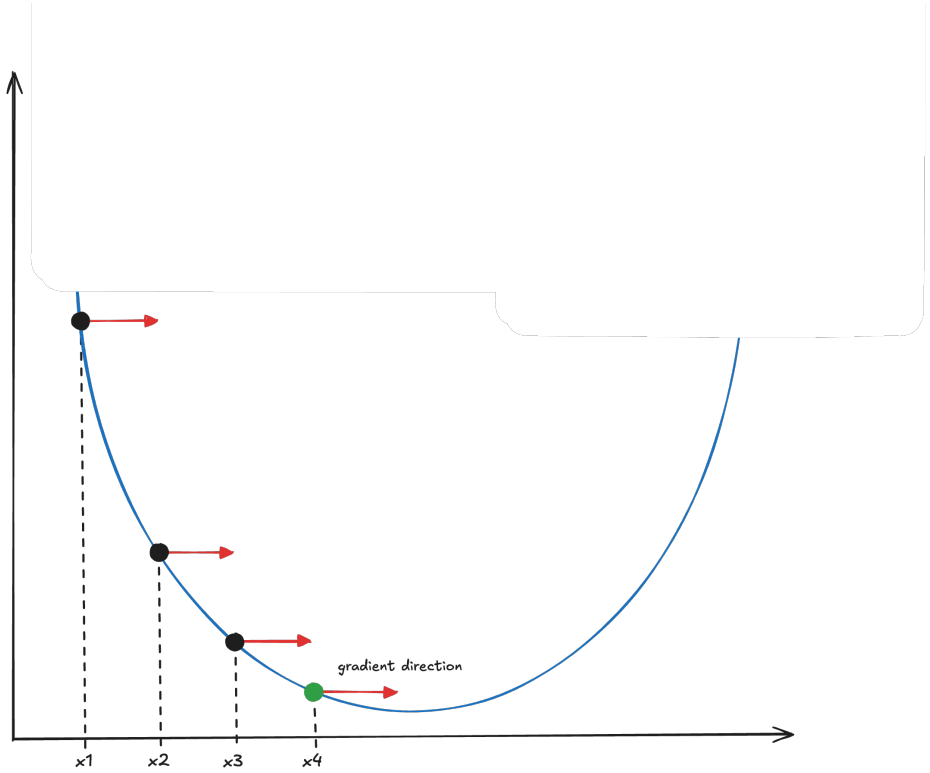
Gradient descent is an *optimization algorithm* that uses *gradient* of the loss function to minimize it.



Fitting models – Gradient descent

We have defined the loss function, and now need to find the parameter values that minimize it.

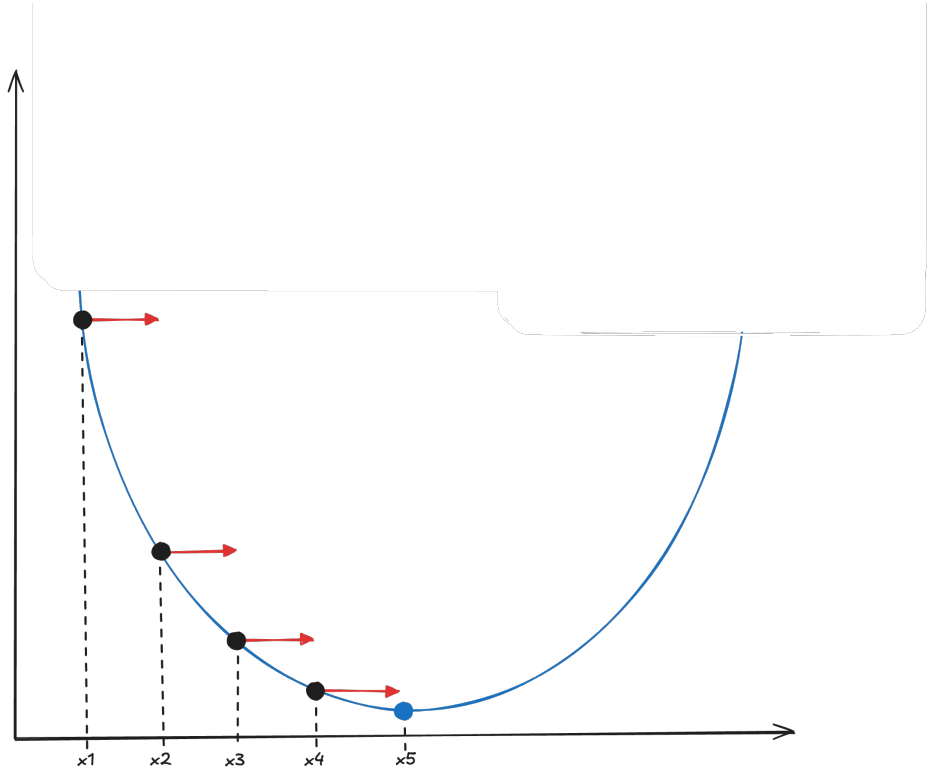
Gradient descent is an *optimization algorithm* that uses *gradient* of the loss function to minimize it.



Fitting models – Gradient descent

We have defined the loss function, and now need to find the parameter values that minimize it.

Gradient descent is an *optimization algorithm* that uses *gradient* of the loss function to minimize it.



Gradient descent – The algorithm

The gradient descent algorithm has two simple steps:

1. Compute the gradient of the loss w.r.t. its parameters:

$$\frac{\partial L}{\partial \phi} = \begin{bmatrix} \frac{\partial L}{\partial \phi_0} & \frac{\partial L}{\partial \phi_1} & \cdots & \frac{\partial L}{\partial \phi_N} \end{bmatrix}^T$$

2. Update the parameter values:

$$\phi \leftarrow \phi - \alpha \cdot \frac{\partial L}{\partial \phi}$$

Here, alpha determines the rate of change, and is typically called the *learning rate*.

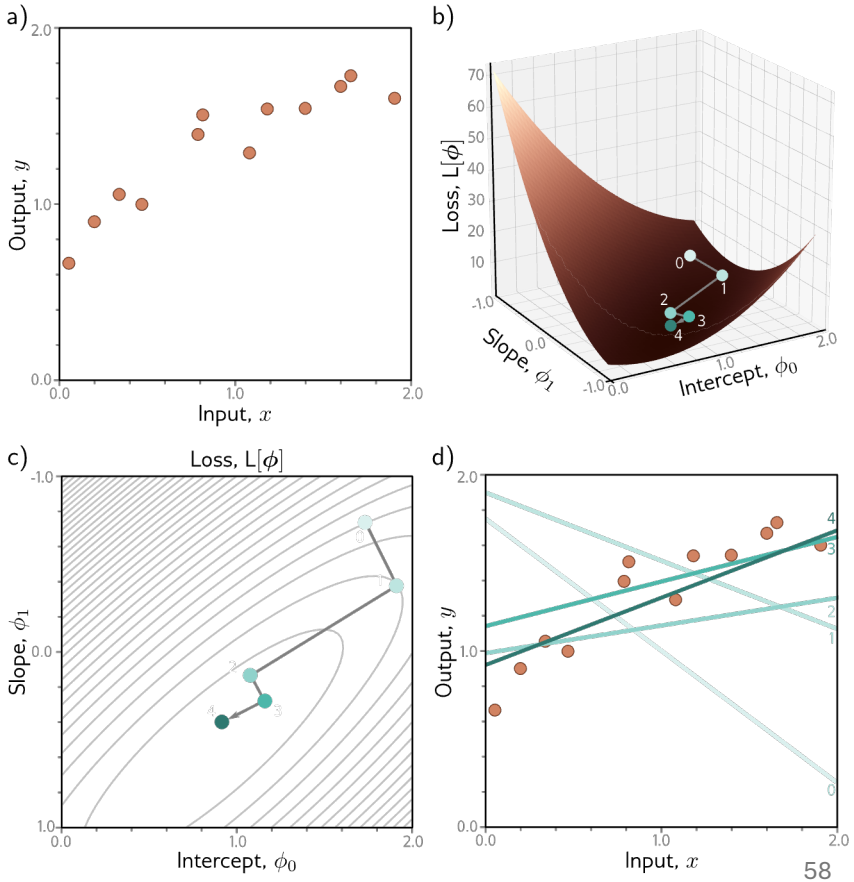
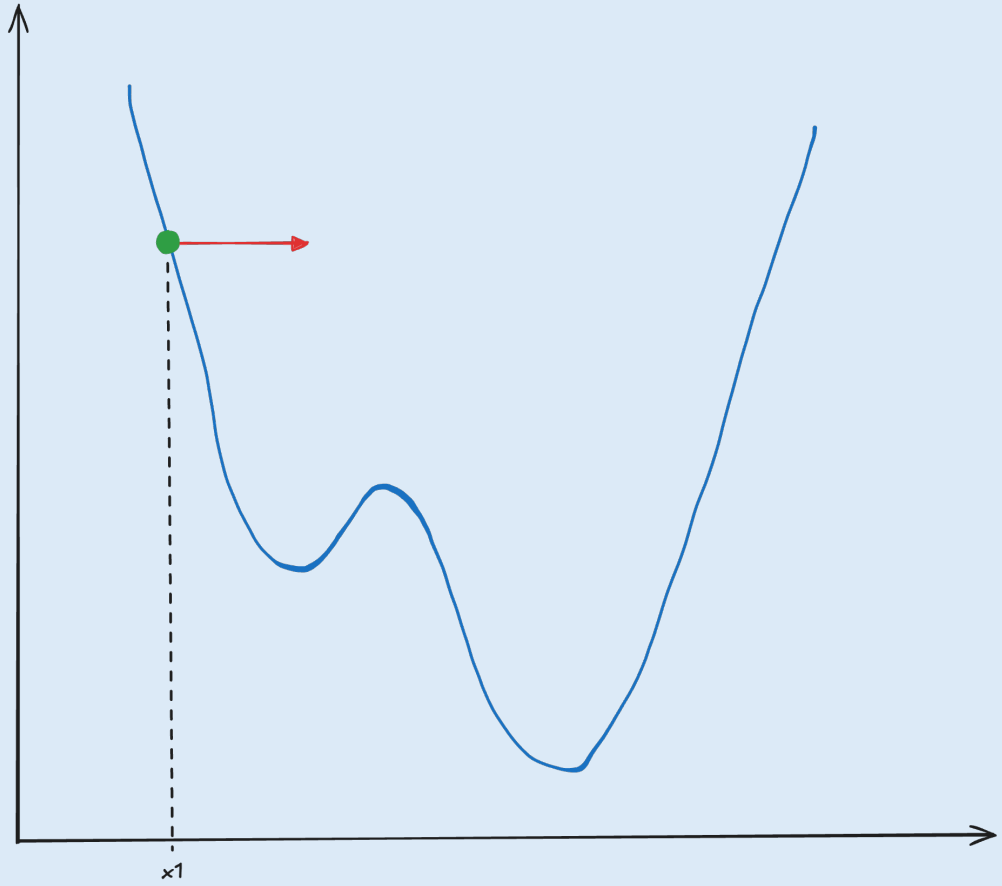


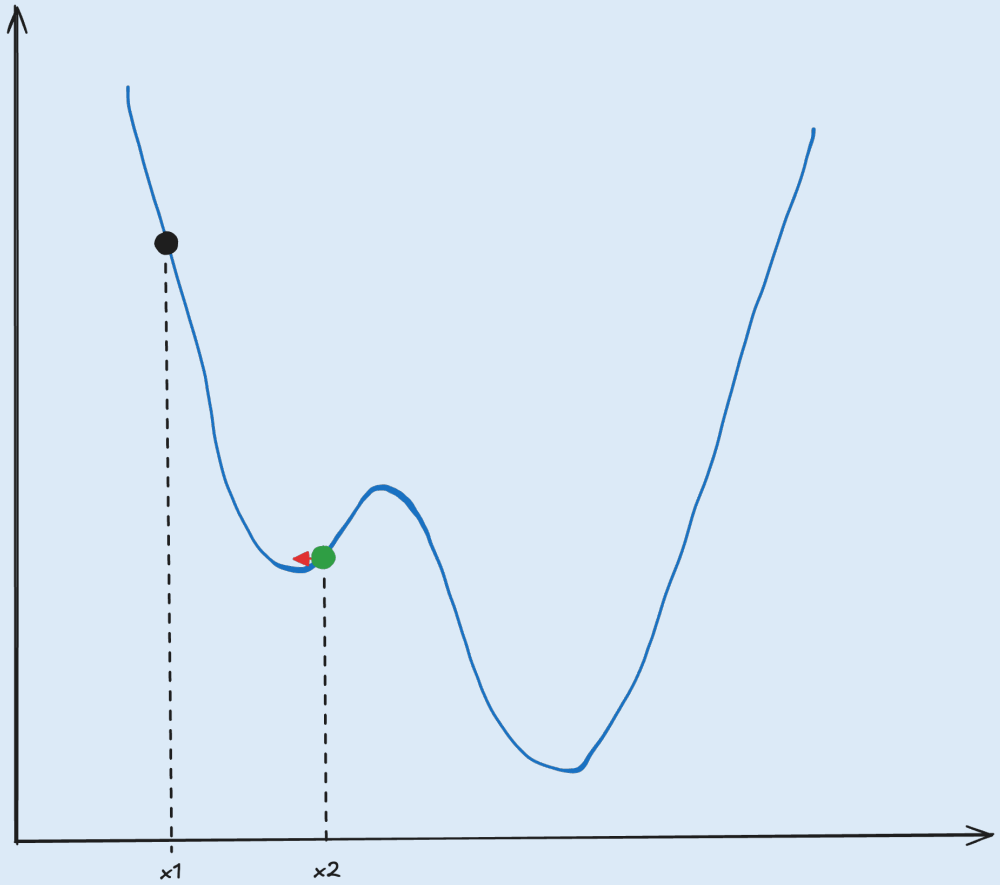
Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Reflection: What problems could we encounter with gradient descent?

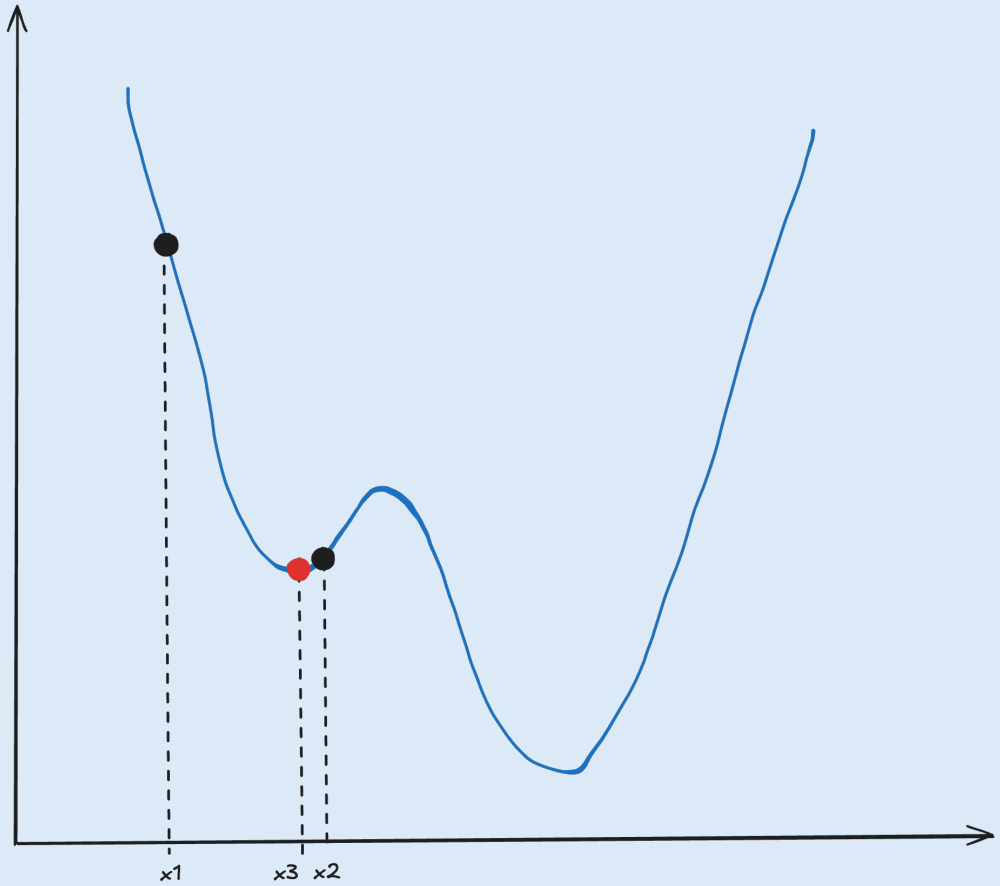
Reflection: What problems could we encounter with gradient descent?



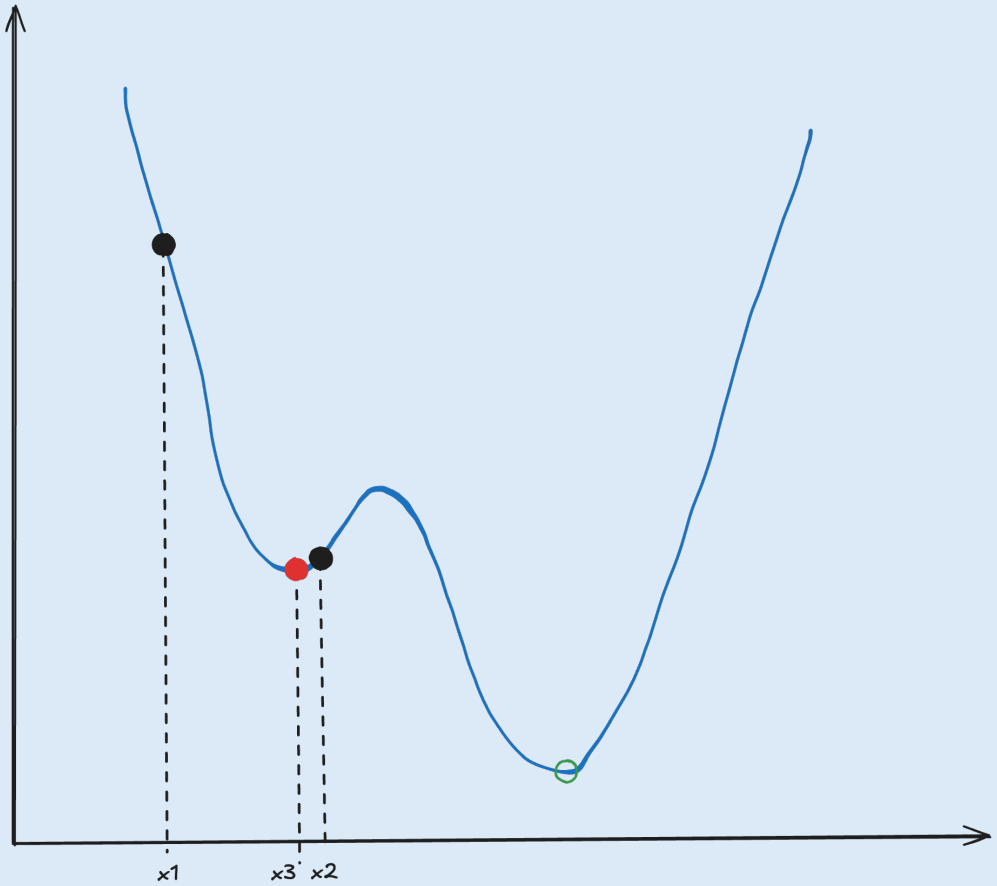
Reflection: What problems could we encounter with gradient descent?



Reflection: What problems could we encounter with gradient descent?



Reflection: What problems could we encounter with gradient descent?

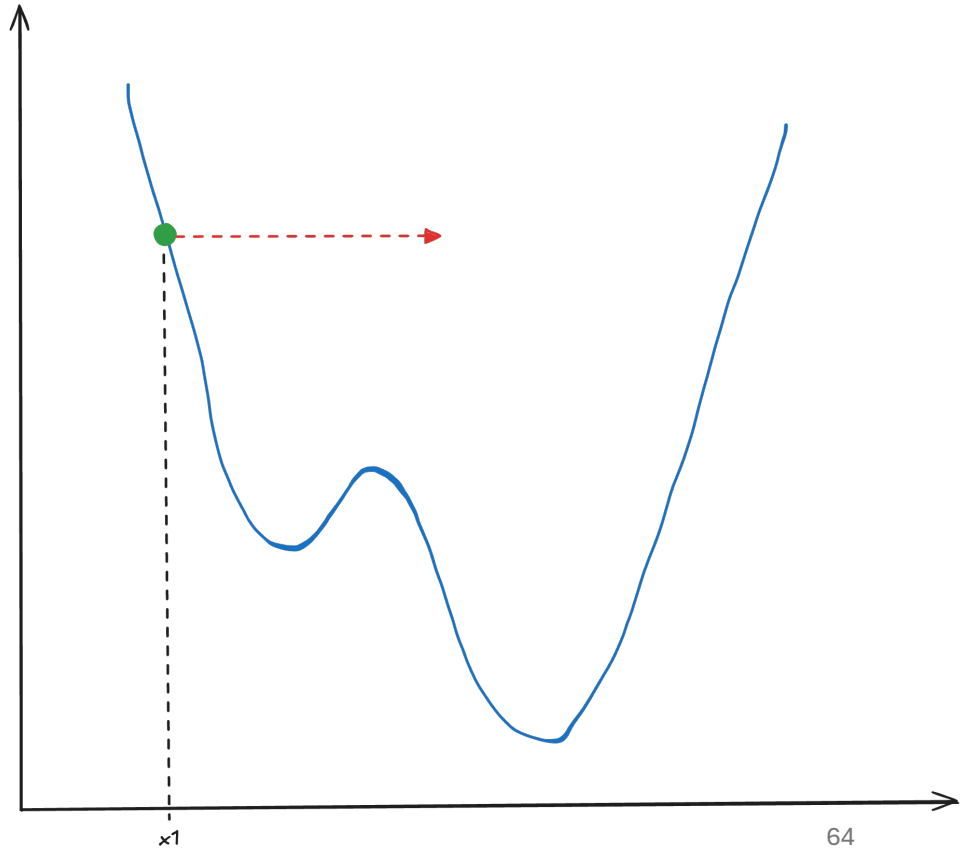


Stochastic gradient descent – Introducing noise to the optimization algorithm

Often, the functions we are trying to minimize are *non-convex*, and can have several local minima.

In such cases, regular gradient descent optimization might get stuck.

Therefore, we introduce noise to the optimization, which reduces the chance of this happening.



SGD – Training on random mini-batches at each step

In general, our loss functions can be thought of as a sum of loss terms for each training sample:

$$L[\phi] = \sum_{i=1}^{I} \ell_i[\phi]$$

In regular gradient descent, we include all of these terms in the computation of the gradient.

In SGD, we introduce randomness by only computing the gradient for a random subset of the samples at each step:

$$\phi_{t+1} \leftarrow \phi_t - \alpha \cdot \sum_{i \in \mathcal{B}_t} \frac{\partial \ell_i[\phi_t]}{\partial \phi}$$

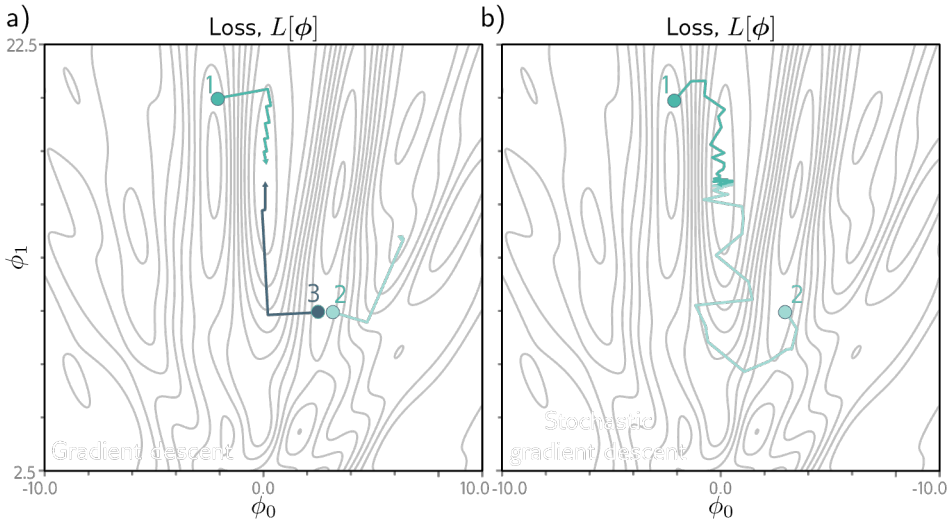


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Batches, epochs, monitoring and checkpointing

The subset of training samples that we use for each update is called a *batch*. The number of elements in the batches can vary from one to the entire training set, in which case we are back at regular gradient descent.

When we have small batch sizes, we need several iterations to cover the entire training set. A full pass through the training set is called an *epoch*.

In practice, we often make checkpoints, and monitor progress at each step.

Digression: The backpropagation algorithm

The backpropagation algorithm is an efficient way of computing derivatives in a neural network, that applies the *chain rule*.

It involves doing a *forward pass,* where we compute (and store) all the intermediate values needed to calculate the loss.

Once we have calculated the loss, we can compute derivatives of all the intermediate variables, applying the chain rule.

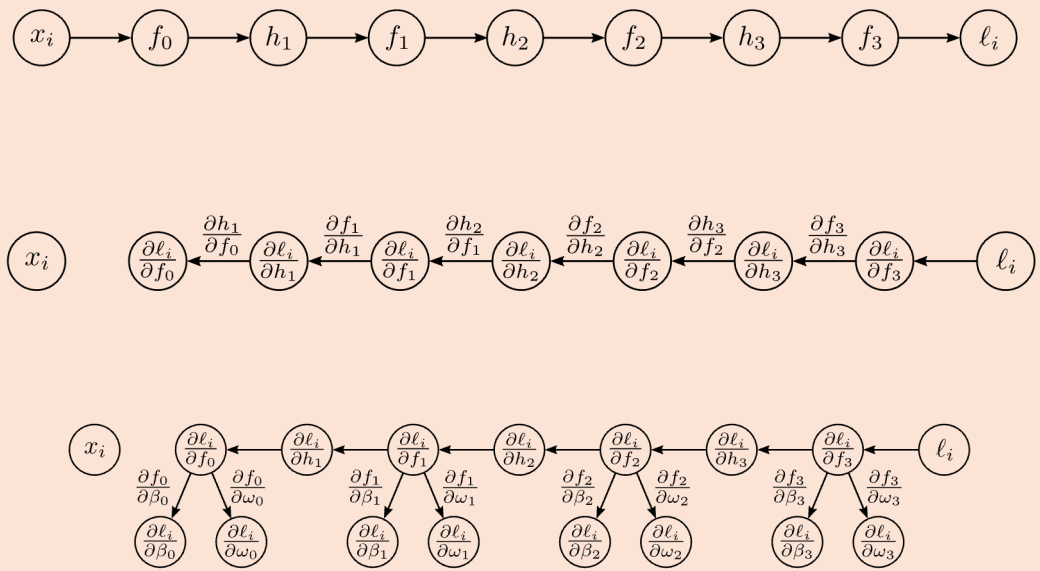


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Code example 4: Training a model using SGD

```python
# Define the neural network
class ShallowNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(1, 3)
        self.output = nn.Linear(3, 1)

    def forward(self, x):
        pre_act = self.hidden(x)
        hidden = F.relu(pre_act)
        output = self.output(hidden)
        return output, pre_act, hidden

# Create an instance of the network
shallow_model = ShallowNN()

# Example input
x = torch.tensor([[1.0]], dtype=torch.float32)

# Forward pass
output, *_ = shallow_model(x)
print(f"Input shape: {x.shape}")
print(f"Output shape: {output.shape}")
print(f"Output value: {output.item()}")
```

Specialized architectures

Models designed to tackle hard problems

Specialized architectures – Modifications to deal with complex inputs, etc.

We have now looked at the basic components of deep learning models, what functions they represent and how we can train them.

Specialized *architectures* have been developed to deal with task specific problems such as large inputs, inputs of varying length and spatial or textual context.

Examples of such architectures include *convolutional neural networks* for image analysis and *transformers* for text processing.

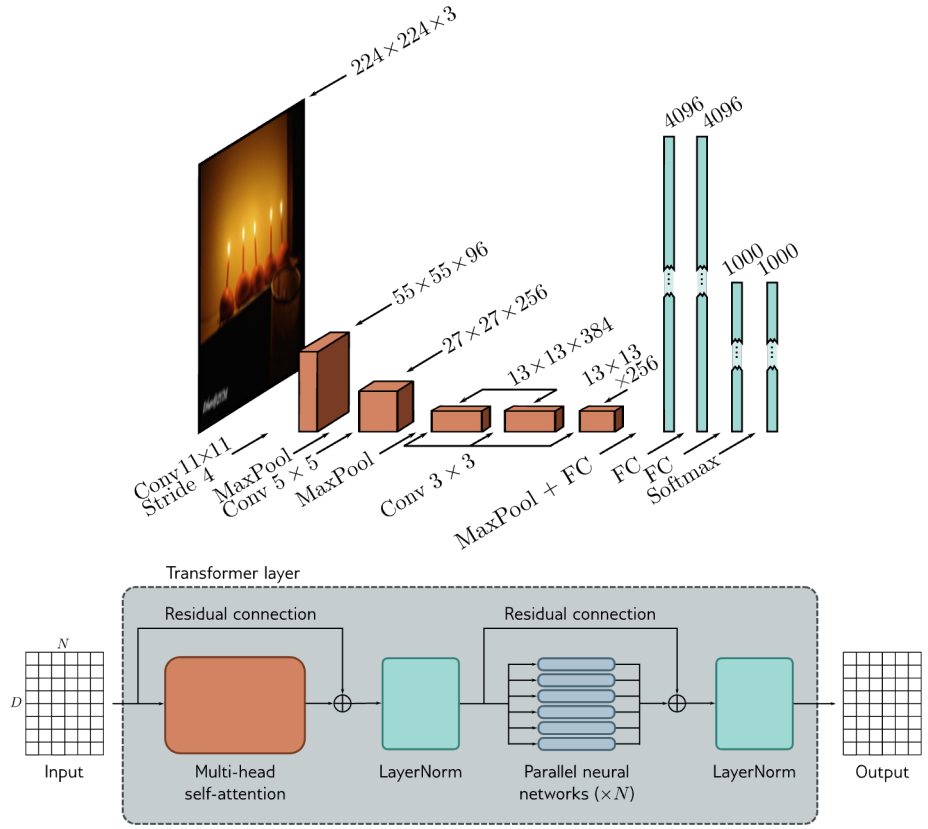


Figure borrowed from Prince, S. J. (2023). *Understanding deep learning*. MIT press.

Code example 5: MNIST

```python
# Define the neural network
class ShallowNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden = nn.Linear(1, 3)
        self.output = nn.Linear(3, 1)

    def forward(self, x):
        pre_act = self.hidden(x)
        hidden = F.relu(pre_act)
        output = self.output(hidden)
        return output, pre_act, hidden

# Create an instance of the network
shallow_model = ShallowNN()

# Example input
x = torch.tensor([[1.0]], dtype=torch.float32)

# Forward pass
output, *_ = shallow_model(x)
print(f"Input shape: {x.shape}")
print(f"Output shape: {output.shape}")
print(f"Output value: {output.item()}")
```

Some important topics that haven't been covered

- Regularization
  - Explicit and implicit
- Learning rates and momentum
- Weight initialization
- Residual connections and batch normalization

Useful resources

- UDL Book: https://udlbook.github.io/udlbook/
- A Recipe For Training Neural Networks: https://karpathy.github.io/2019/04/25/recipe/
- PyTorch examples: https://github.com/pytorch/examples/tree/main

Thank you for listening!

You can reach me at elias@simula.no if you have follow-up questions.