Deep Learning for Social Scientists

University of Oslo

Reto Wüest

reto.wuest@coop.no

Data Scientist, Coop Norge SA

November 26, 2024

Course Schedule

- Wed, November 27, 2024
- Thu, November 28, 2024

Course Schedule

Session	Content	Readings
Morning	– ML and supervised learning	Prince, chs. 1–2
(9.00–12.00)	– Python and PyTorch	
	– *Demo 1: 1D Linear regression*	
	– Shallow neural networks	Prince, ch. 3
	– Deep neural networks	Prince, ch. 4
Afternoon	– Loss functions	Prince, ch. 5
(13.00–16.00)	– Fitting models	Prince, chs. 6–7
	– *Demo 2: Gradient descent*	
	– Measuring performance	Prince, ch. 8
	– *Demo 3: Fully connected deep NNs*	

Course Schedule

Session	Content	Readings
Morning	– Convolutional neural networks (CNNs)	Prince, ch. 10
(9.00–12.00)	– *Demo 4: CNNs*	
Afternoon	– Transformers	Prince, ch. 12
(13.00–16.00)	– *Demo 5: Transformers*	

Course Logistics

Course Logistics

- You will need for the course:
  - → a web browser,
  - → a Google account.
- The textbook for the course is:
  - → Prince, Simon J.D. 2024. ***Understanding Deep Learning***.
- Other useful deep learning books are:
  - → Bishop, Christopher M. and Hugh Bishop. 2024. ***Deep Learning: Foundations and Concepts***.
  - → Zhang, Aston, Zachary C. Lipton, Mu Li, and Alexander J. Smola. 2023. ***Dive into Deep Learning***.
- Course demos use Jupyter notebooks hosted on Google Colab and GitHub.

Overview of Jupyter Notebooks and Course Demos

Demo	Content	Notebook	Google Colab	GitHub
Demo 1	1D linear regression w/ `scikit-learn`	Notebook 1	Colab nb-1	GitHub nb-1
Demo 2	1D linear regression w/ gradient descent			
Demo 3	Fully connected deep neural networks	Notebook 2	Colab nb-2	GitHub nb-2
Demo 4	Convolutional neural networks (CNNs)	Notebook 3	Colab nb-3	GitHub nb-3
Demo 5	Transformers	Notebook 4	Colab nb-4	GitHub nb-4

ML Concepts and Definitions

AI, Machine Learning, and Deep Learning

Artificial Intelligence (AI)

Artificial intelligence (AI) is the field concerned with building systems that simulate intelligent behavior.

Machine Learning (ML)

Machine learning (ML) is the study of algorithms that can learn from experience. Learning means that as an algorithm accumulates experience, its performance improves.

ML is a subfield of AI.

Deep Learning (DL)

Deep learning (DL) uses deep neural networks, which are a type of ML model, to learn from experience.

DL is a subfield of ML.

AI, Machine Learning, and Deep Learning

- Machine learning methods can be divided into three areas:
  - → Supervised learning
  - → Unsupervised learning
  - → Reinforcement learning
- In all three areas, many of the cutting-edge methods rely on deep learning.
- Our focus in this course is on supervised deep learning.

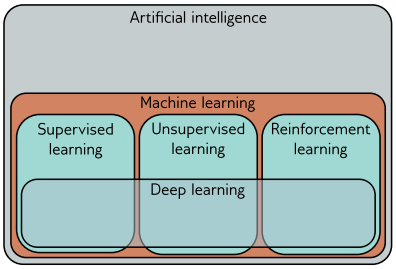

Figure 1: AI, ML, and DL (Prince 2024, 2)

Supervised Learning

Supervised Learning

Supervised learning models learn a mapping from one or more inputs to one or more outputs. They learn this mapping from labeled training data, which are examples of input/output pairs.

Supervised learning deals with regression and classification problems.

Regression Problems

In regression problems the goal is to map inputs to continuous outputs.

In a univariate regression problem the output is a real-valued scalar. In a multivariate regression problem the output is a real-valued vector.

Classification Problems

In classification problems the goal is to map inputs to categorical outputs.

In a binary classification problem the output has two categories. In a multiclass classification problem the output has more than two categories.

Unsupervised Learning

Unsupervised Learning

Unsupervised learning models aim to describe patterns or structure in data. They learn such patterns or structure from unlabeled training data, which are examples of inputs (w/o corresponding output labels).

Generative Unsupervised Learning

Generative unsupervised models learn to create new data examples that are statistically indistinguishable but distinct from the training examples.

Generative unsupervised models are a subset of unsupervised learning models. They have recently gained prominence (e.g., in generating synthetic survey data).[1]

Reinforcement Learning

Reinforcement Learning

Reinforcement learning is characterized by an agent (learner) that

- lives in a world described by a set of states,
- can choose among a set of actions that change the state of the world,
- can receive rewards as a result of the state of the world.

The goal is for the agent to learn actions that change the state in a way leading to high rewards (e.g., chess).

The learner does not receive any data, but has to explore the environment to gather data as it goes.

When to Use Deep Learning

- The performance of deep learning models is impressive for many learning tasks (e.g., image classification, speech recognition, language translation, etc.).
- *Should we therefore use deep learning on every learning problem?*
- We expect deep learning models to perform well when the training set is very large and can support the fitting of high-dimensional nonlinear models.
- On smaller data sets, simpler models (such as linear regression, lasso, etc.) may perform similarly well, yet they are easier to fit and interpret.[1]
- Therefore, it can make sense to fit deep neural networks as well as simpler models and then compare them based on their performance.

1. Bisbee et al. (2024) compared synthetic survey data produced by a large language model (ChatGPT 3.5) to real-world data from the American National Election Study (ANES). They found that the synthetic data come with reproducibility issues and are not reliable for inferences about public opinion.

1. See James et al. (2021) for an example where a (shallow) neural network performs similarly well as a linear model and a lasso model.

Supervised Learning

Supervised Learning

- In supervised learning, we build a model $\mathbf{f}[\cdot]$ that maps input $\mathbf{x}$ to output $\mathbf{y}$, $$\mathbf{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}].$$
- The model $\mathbf{f}[\cdot]$ is a function, which describes a family of possible relationships between input $\mathbf{x}$ and output $\mathbf{y}$.
- The model also contains parameters $\boldsymbol{\phi}$. The parameter values define the particular relationship between input and output.
- We typically learn the parameters by feeding an algorithm with a training data set $\mathcal{S}$ that contains N pairs of input/output examples, $$\mathcal{S} = \{(x_{i}, y_{i})\}_{i = 1}^{N}.$$

Supervised Learning

- The goal of the algorithm is to choose parameters that map inputs to outputs as closely as possible.
- To quantify the degree of mismatch in the mapping, we define a loss function $L[\boldsymbol{\phi}, \mathcal{S}]$ or, for short, $L[\boldsymbol{\phi}]$.
- When training the model, the algorithm chooses parameters $\boldsymbol{\hat{\phi}}$ that minimize the loss function, $$\boldsymbol{\hat{\phi}} = \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\big[L[\boldsymbol{\phi}]\big].$$
- After training the model, we assess its performance on a separate test data set to see how well it generalizes to new examples.
- If the performance is adequate, we can deploy our trained model $\mathbf{f}[\mathbf{x}, \boldsymbol{\hat{\phi}}]$.

Supervised Learning

Example: 1D Linear Regression

- The 1D linear regression model describes the relationship between input \(x\) and output \(y\) as a straight line, \[\begin{split} y &= f[x, \boldsymbol{\phi}] \\ &= \phi_{0} + \phi_{1}x. \end{split} \]
- Different choices for parameters \(\boldsymbol{\phi} = [\phi_{0}, \phi_{1}]^{T}\) result in different lines relating \(x\) and \(y\).

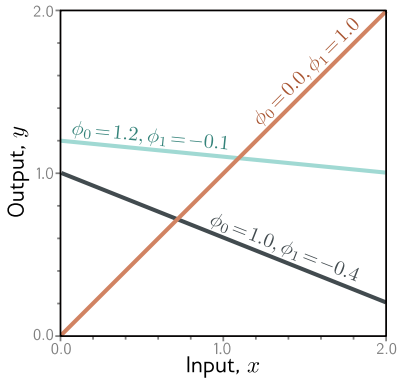

Figure 2: Linear regression model (Prince 2024, 19)

Supervised Learning

Example: 1D Linear Regression (Cont.)

- The training data set \(\mathcal{S}\) consists of \(N\) input/output pairs, \(\mathcal{S} = \{(x_{i}, y_{i})\}_{i = 1}^{N}\).
- To quantify the total mismatch between the model predictions \(f[x_{i}, \boldsymbol{\phi}]\) and the ground truth outputs \(y_{i}\), we use a least-squares loss function, \[\begin{split} L[\boldsymbol{\phi}] &= \sum_{i = 1}^{N}(f[x_{i}, \boldsymbol{\phi}] - y_{i})^{2} \\ &= \sum_{i = 1}^{N}(\phi_{0} + \phi_{1}x_{i} - y_{i})^{2}. \end{split} \]

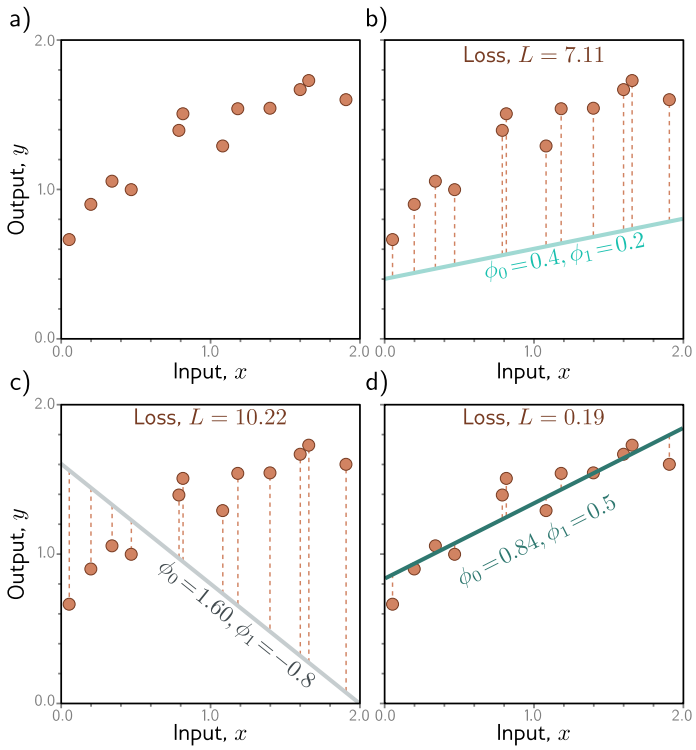

Figure 3: Training data (panel a). Linear regression model with different parameter values and loss (panels b-d) (Prince 2024, 20)

Supervised Learning

Example: 1D Linear Regression (Cont.)

- Each combination of parameters $\boldsymbol{\phi} = [\phi_{0}, \phi_{1}]^{T}$ has an associated loss.
- The goal is to find the parameters $\boldsymbol{\hat{\phi}}$ that minimize the loss function $L[\boldsymbol{\phi}]$,

$$
\begin{aligned}
\boldsymbol{\hat{\phi}} &= \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\big[L[\boldsymbol{\phi}]\big] \\
&= \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\Bigg[\sum_{i = 1}^{N}(f[x_{i}, \boldsymbol{\phi}] - y_{i})^{2}\Bigg] \\
&= \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\Bigg[\sum_{i = 1}^{N}(\phi_{0} + \phi_{1}x_{i} - y_{i})^{2}\Bigg].
\end{aligned}
$$

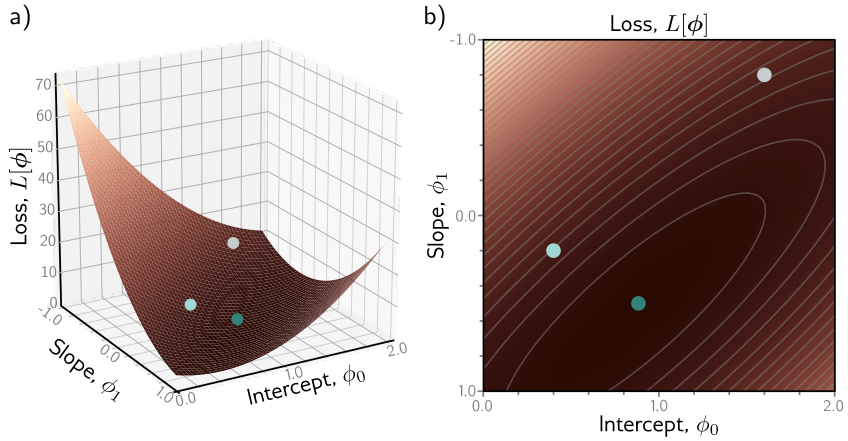

Figure 4: Loss function for above linear regression model and training data. Loss function can be visualized as a surface (panel a) and as a heatmap (panel b) (Prince 2024, 21)

Supervised Learning

Example: 1D Linear Regression (Cont.)

- The process of finding the parameters that minimize the loss function is called model fitting, training, or learning.
- The basic method for model fitting is the gradient descent algorithm:
  - → randomly choose initial parameters,
  - → improve them by “walking down” the loss function until you reach the bottom.
- After training the model, we want to assess how well it generalizes to new data. We do this by computing the loss of the trained model \(f[x, \boldsymbol{\hat{\phi}}]\) on a separate test data set.
- The trained model might not generalize well because:
  - → it is too simple and therefore not able to capture the true relationship between input and output (underfitting),
  - → it is too complex and fits random noise in the training data (overfitting).

Python/PyTorch

Demo 1: 1D Linear Regression

Notebook 1 in Google Colab

Python

- The Python programming language is widely used in data science due to its readability and vast ecosystem of libraries.
- Python is a dynamically typed language, so the data type of a variable is determined at runtime and does not need to be declared beforehand.
- Core Python libraries for data science include:
  - → NumPy (efficient computing with multi-dimensional arrays),
  - → Pandas (data manipulation with tables and time series),
  - → SciPy (scientific computing, e.g., optimization, integration, etc.),
  - → Matplotlib (plotting library),
  - → Scikit-learn (machine learning, e.g., linear models, tree-based models, SVM, etc.).
- Training deep learning models typically requires optimization of a large number of parameters.
- We need a framework that can handle large-scale optimization efficiently by utilizing graphics processing units (GPUs). PyTorch is one such framework.[1]

PyTorch

- PyTorch is one of the most popular frameworks for deep learning and has a tight integration with Python.
- In PyTorch, the fundamental data structure is the tensor, which is a multi-dimensional array:
  - → a rank-0 tensor is a scalar,
  - → a rank-1 tensor is a vector,
  - → a rank-2 tensor is a matrix,
  - → etc.
- Tensors can run on central processing units (CPUs) or graphics processing units (GPUs).
- PyTorch uses dynamic computation graphs, which are constructed at runtime and are therefore more flexible and easier to debug than static graphs.[1]

1. A computation graph is a directed acyclic graph (DAG), where the nodes are operations and the edges are data (tensors). Each node applies an operation to its input tensor(s) and produces zero or more tensors as the output.

1. Scikit-learn supports deep learning models, but its implementation is not optimized for large-scale applications. In particular, Scikit-learn offers no GPU support. (see here).

Shallow Neural Networks

Shallow Neural Networks

- We previously used a 1D linear regression model to describe the relationship between input \(x\) and output \(y\).
- However, this model can only describe the input/output relationship as a line.
- Shallow neural networks are a more flexible model.
- They describe piecewise linear functions and can approximate arbitrariliy complex relationships between inputs and outputs.

Shallow Neural Networks

Example: 1D Shallow Neural Network

- Shallow neural networks are functions $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with parameters $\boldsymbol{\phi}$ that map inputs $\mathbf{x}$ to outputs $\mathbf{y}$.
- Consider a 1D shallow neural network $f[x, \boldsymbol{\phi}]$ that maps a scalar input x to a scalar output y and has 10 parameters $\boldsymbol{\phi}$), $$\begin{aligned} y &= f[x, \boldsymbol{\phi}] \\ &= \phi_{0} + \phi_{1}a[\theta_{10} + \theta_{11}x] + \phi_{2}a[\theta_{20} + \theta_{21}x] + \phi_{3}a[\theta_{30} + \theta_{31}x]. \end{aligned} \qquad (1)$$
- This model has three parts:
  - → three linear functions of the input, $([\theta_{10} + \theta_{11}x], [\theta_{20} + \theta_{21}x], [\theta_{30} + \theta_{31}x]$),
  - → an activation function $a[\cdot]$,
  - → a sum of the weighted activations (weights are $\phi_{1}, \phi_{2}, \phi_{3}$) and offset ϕ_{0}.
- How do we define the activation function $a[\cdot]$?

Shallow Neural Networks

Example: 1D Shallow Neural Network (Cont.)

- The most common choice for $a[\cdot]$ is the rectified linear unit (ReLU), $$a[z] = \mathrm{ReLU}[z] = \begin{cases} 0 & \text{if } z < 0, \\ z & \text{if } z \geq 0. \end{cases}$$
- This returns the input z when z is positive and 0 when it is negative (it clips negative values of z to 0).
- Having defined $a[\cdot]$, we can proceed by specifying a least squares loss function $L[\boldsymbol{\phi}]$ and train the model on a training data set $\mathcal{S} = \{(x_{i}, y_{i})\}_{i = 1}^{N}$ by searching for the values $\boldsymbol{\hat{\phi}}$ that minimize $L[\boldsymbol{\phi}]$.

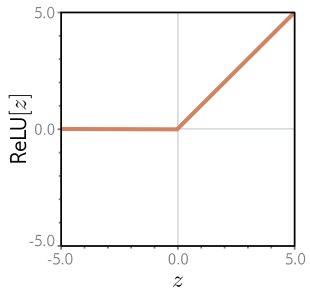

Figure 5: Rectified linear unit (ReLU) activation function (Prince 2024, 26)

Shallow Neural Networks

Example: 1D Shallow Neural Network (Cont.)

- Equation 1 represents a family of continuous piecewise linear functions with up to four linear regions.
- To see this, we split the function into two parts:
  - → the intermediate quantities (called hidden units), $$\begin{split} h_{1} &= a[\theta_{10} + \theta_{11}x] \\ h_{2} &= a[\theta_{20} + \theta_{21}x] \\ h_{3} &= a[\theta_{30} + \theta_{31}x], \end{split}$$
  - → the output combining these hidden units via a linear function, $$y = \phi_{0} + \phi_{1}h_{1} + \phi_{2}h_{2} + \phi_{3}h_{3}.$$
- Figure 6 (next slide) illustrates the flow of computation for the model described by Equation 1, using the ReLU activation function.

Shallow Neural Networks

Example: 1D Shallow Neural Network (Cont.)

- Each linear region in panel j) corresponds to a different activation pattern in the hidden units.
- When a unit is clipped, we say it is inactive; and when it is not clipped, we say it is active.
  - → The shaded region in panel j) receives contributions from \(h_{1}\) and \(h_{3}\) (which are active) but not from \(h_{2}\) (which is inactive).
  - → Each hidden unit can contribute one “joint” to the function, so there can be up to four linear regions.
  - → The positions where the lines in panels a)-c) cross 0 become the joints in the final output in panel j).

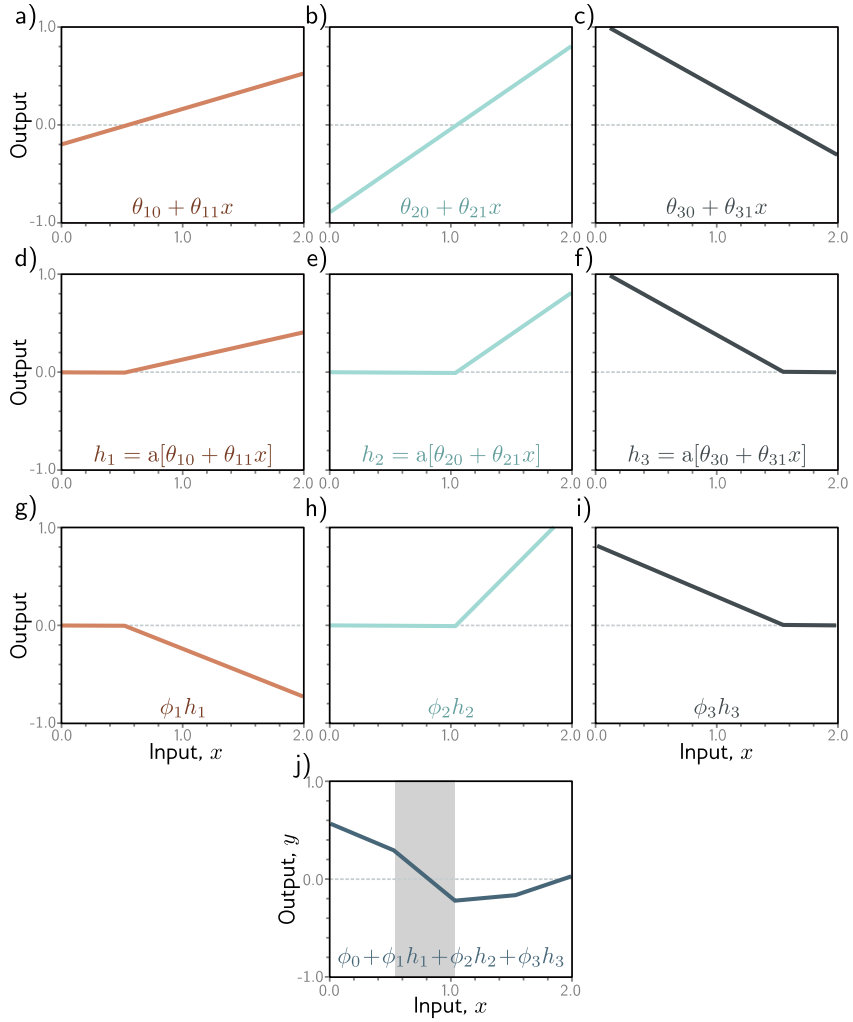

Figure 6: Flow of computation for Equation 1 w/ ReLU (Prince 2024, 28)

Shallow Neural Networks

Example: 1D Shallow Neural Network (Cont.)

- We visualize the above network as in **Figure 7**.
  - → The input is on the left.
  - → The hidden units are in the middle.
  - → The output is on the right.
  - → Each arrow represents a parameter.
- **Figure 8** shows some neural network terminology.
  - → The inputs form the input layer.
  - → The hidden units form the hidden layer.
  - → The outputs form the output layer.
  - → Shallow NN: one hidden layer.
  - → Deep NN: multiple hidden layers.
  - → Feed-forward NN: each layer is connected to the next by forward connections (arrows).
  - → Fully connected NN: every element of a layer connects to every element of the next.

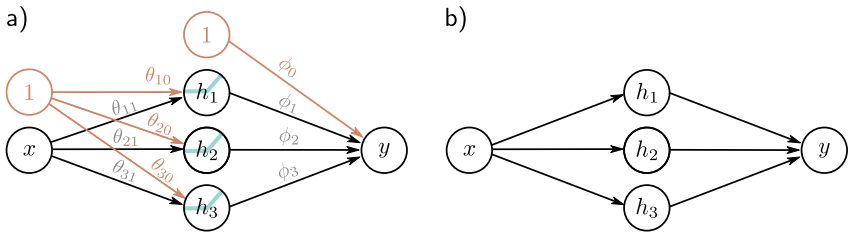

Figure 7: Visualizing neural networks. Typically, the visualization omits the intercepts, activation functions, and parameter names, as in panel b) (Prince 2024, 29)

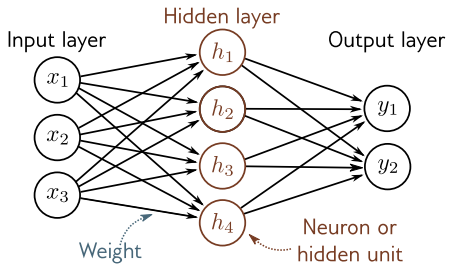

Figure 8: Terminology (Prince 2024, 36)

Shallow Neural Networks

Universal Approximation Theorem

- With ReLU activation functions, the output of a network with \(D\) hidden units is a piecewise linear function with at most \(D + 1\) linear regions.
- As we increase the number of hidden units \(D\), the model can approximate more complex functions.

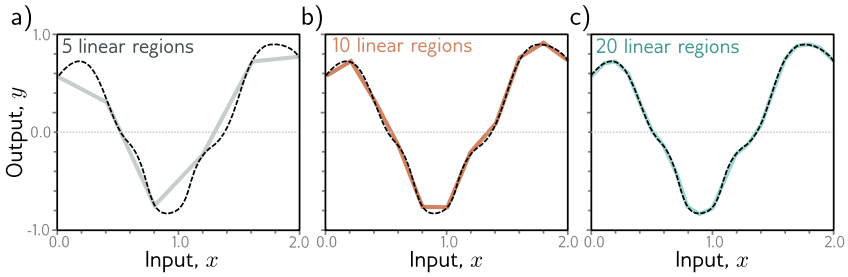

Figure 9: Approximation of a 1D function (dashed line) by a shallow neural network as network capacity (# of hidden units) increases (Prince 2024, 30)

Universal Approximation Theorem

With enough hidden units, a shallow neural network can describe any continuous function on a **compact** subset of \(\mathbb{R}^{D}\) to arbitrary precision.

Shallow Neural Networks

General Case

- The above discussion generalizes to the case of a shallow neural network with an arbitrary number of inputs, hidden units, and outputs.
- In this general case, a shallow neural network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ maps a multi-dimensional input $\mathbf{x} \in \mathbb{R}^{D_{i}}$ to a multi-dimensional output $\mathbf{y} \in \mathbb{R}^{D_{o}}$ using hidden units $\mathbf{h} \in \mathbb{R}^{D}$.
- Each hidden unit h_{d}, for $d = 1, \dotsc, D$, is computed as: $$h_{d} = a\Big[\theta_{d0} + \sum_{i = 1}^{D_{i}}\theta_{di}x_{i}\Big],$$ where $a[\cdot]$ is a non-linear activation function (e.g., ReLU).
- *Note:* The activation function $a[\cdot]$ allows the model to describe non-linear relations between $\mathbf{x}$ and $\mathbf{y}$ and, therefore, has to be non-linear (such as ReLU).

Shallow Neural Networks

General Case (Cont.)

- The hidden units are then combined linearly to create each output \(y_{j}\), for \(j = 1, \dotsc, D_{o}\): \[y_{j} = \phi_{j0} + \sum_{d = 1}^{D}\phi_{jd}h_{d}. \]

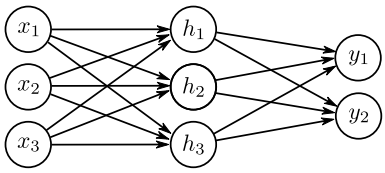

Figure 10: Example of a shallow neural network with three inputs, three hidden units, and two outputs. The network has 20 parameters (15 slopes, indicated by arrows; and 5 offsets, not shown) (Prince 2024, 35)

Deep Neural Networks

Deep Neural Networks

- Shallow neural networks have one hidden layer.
- Deep neural networks have more than one hidden layer.
- Both shallow and deep neural networks describe piecewise linear mappings from input to output (with ReLU activation functions).
- With enough hidden units, shallow neural networks can describe arbitrarily complex functions in high dimensions (universal approximation theorem).
  - → However, for some functions, doing so requires an impractically large number of hidden units.
- Deep neural networks are more practical.
  - → They can produce many more linear regions than shallow neural networks for a given number of parameters.

Deep Neural Networks

Connecting Shallow Neural Networks

- To better understand the behavior of deep neural networks, we first consider connecting two shallow ones (with three hidden units each).
- The first shallow network takes input x and returns output y. The second shallow network takes y as input and returns output $y^{\prime}$, see **Figure 11**, panel a). $$\begin{split} h_{1} &= a[\theta_{10} + \theta_{11}x] \\ \text{Network 1:} \qquad h_{2} &= a[\theta_{20} + \theta_{21}x] \qquad y = \phi_{0} + \phi_{1}h_{1} + \phi_{2}h_{2} + \phi_{3}h_{3} \\ h_{3} &= a[\theta_{30} + \theta_{31}x]. \end{split}$$ $$\begin{split} h_{1}^{\prime} &= a[\theta_{10}^{\prime} + \theta_{11}^{\prime}y] \\ \text{Network 2:} \qquad h_{2}^{\prime} &= a[\theta_{20}^{\prime} + \theta_{21}^{\prime}y] \qquad y^{\prime} = \phi_{0}^{\prime} + \phi_{1}^{\prime}h_{1}^{\prime} + \phi_{2}^{\prime}h_{2}^{\prime} + \phi_{3}^{\prime}h_{3}^{\prime} \\ h_{3}^{\prime} &= a[\theta_{30}^{\prime} + \theta_{31}^{\prime}y]. \end{split}$$

Deep Neural Networks

Connecting Shallow Neural Networks (Cont.)

- Figure 11, panel b).
  - → Network 1 maps multiple inputs \(x \in [-1, 1]\) (gray dots) to the same output \(y \in [-1, 1]\) (cyan dot).
- Figure 11, panel c).
  - → Network 2 takes \(y\) (cyan dot) and returns \(y^{\prime}\) (brown dot).
- Figure 11, panel d).
  - → Combined effect of Network 1 and 2: three different inputs \(x\) are mapped to any given value of \(y\) by Network 1 and then processed in the same way by Network 2.
  - → The function defined by Network 2 in panel c) is duplicated three times, flipped and rescaled according to the slope of the regions in panel b), creating nine linear regions.

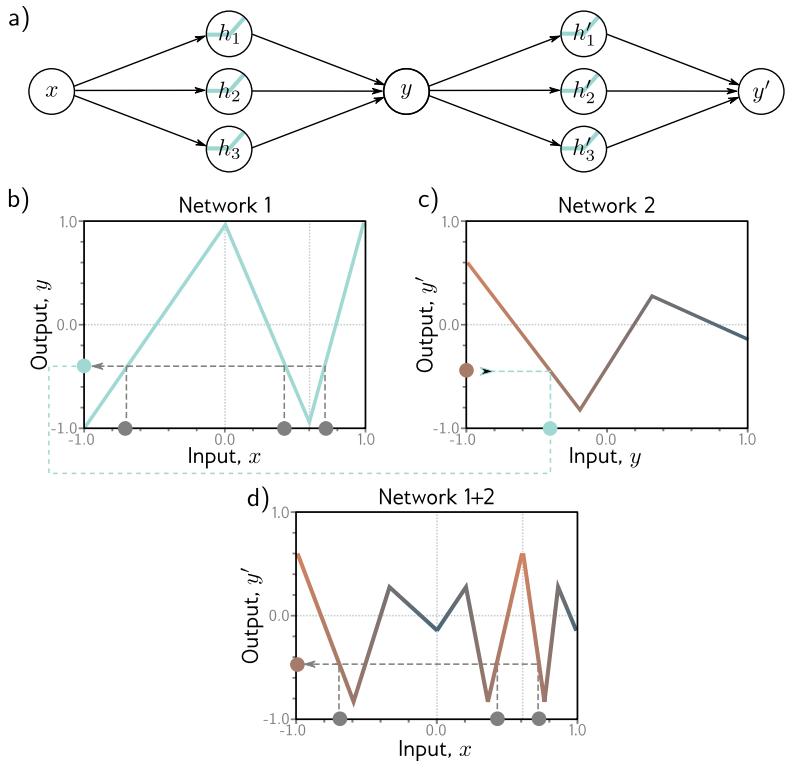

Figure 11: Connecting two shallow neural networks (Prince 2024, 42)

Deep Neural Networks

Connecting Shallow Neural Networks (Cont.)

- Two connected shallow networks with three hidden units each (Figure 12):
  - → 20 parameters,
  - → (at least) 9 linear regions (Figure 11, panel d).
- One shallow network with six hidden units (Figure 13):
  - → 19 parameters,
  - → (max.) 7 linear regions.

Figure 12: Two connected shallow networks with three hidden units each

---

Figure 13: One shallow neural network with six hidden units

Deep Neural Networks

Connected Shallow NN – Special Case of Deep NN

- Two connected shallow neural networks (each with one input, one output, and three hidden units):

$$\begin{split} h_{1} &= a[\theta_{10} + \theta_{11}x] \\ \text{Network 1:} \qquad h_{2} &= a[\theta_{20} + \theta_{21}x] \qquad \color{red}{y = \phi_{0} + \phi_{1}h_{1} + \phi_{2}h_{2} + \phi_{3}h_{3}} \\ h_{3} &= a[\theta_{30} + \theta_{31}x]. \end{split}$$

$$\begin{split} h_{1}^{\prime} &= a[\theta_{10}^{\prime} + \theta_{11}^{\prime}\color{red}{y}\color{black}{]} \\ \text{Network 2:} \qquad h_{2}^{\prime} &= a[\theta_{20}^{\prime} + \theta_{21}^{\prime}\color{red}{y}\color{black}{]} \qquad y^{\prime} = \phi_{0}^{\prime} + \phi_{1}^{\prime}h_{1}^{\prime} + \phi_{2}^{\prime}h_{2}^{\prime} + \phi_{3}^{\prime}h_{3}^{\prime} \\ h_{3}^{\prime} &= a[\theta_{30}^{\prime} + \theta_{31}^{\prime}\color{red}{y}\color{black}{]}. \end{split}$$

- Substituting $y = \phi_{0} + \phi_{1}h_{1} + \phi_{2}h_{2} + \phi_{3}h_{3}$ into $h_{1}^{\prime}, h_{2}^{\prime}, h_{3}^{\prime}$ and expanding yields

$$\begin{split} h_{1}^{\prime} &= a[\theta_{10}^{\prime} + \theta_{11}^{\prime}\color{red}{y}\color{black}{]} = a[\theta_{10}^{\prime} + \theta_{11}^{\prime}\color{red}{\phi_{0}} \color{black}{+} \theta_{11}^{\prime}\color{red}{\phi_{1}h_{1}} \color{black}{+} \theta_{11}^{\prime}\color{red}{\phi_{2}h_{2}} \color{black}{+} \theta_{11}^{\prime}\color{red}{\phi_{3}h_{3}}\color{black}{]} \\ h_{2}^{\prime} &= a[\theta_{20}^{\prime} + \theta_{21}^{\prime}\color{red}{y}\color{black}{]} = a[\theta_{20}^{\prime} + \theta_{21}^{\prime}\color{red}{\phi_{0}} \color{black}{+} \theta_{21}^{\prime}\color{red}{\phi_{1}h_{1}} \color{black}{+} \theta_{21}^{\prime}\color{red}{\phi_{2}h_{2}} \color{black}{+} \theta_{21}^{\prime}\color{red}{\phi_{3}h_{3}}\color{black}{]} \\ h_{3}^{\prime} &= a[\theta_{30}^{\prime} + \theta_{31}^{\prime}\color{red}{y}\color{black}{]} = a[\theta_{30}^{\prime} + \theta_{31}^{\prime}\color{red}{\phi_{0}} \color{black}{+} \theta_{31}^{\prime}\color{red}{\phi_{1}h_{1}} \color{black}{+} \theta_{31}^{\prime}\color{red}{\phi_{2}h_{2}}$$

\color{black}{+} \theta_{31}^{\prime}\color{red}
{\phi_{3}h_{3}}\color{black}{]}. \end{split} \]

Deep Neural Networks

Connected Shallow NN – Special Case of Deep NN (Cont.)

- Deep neural network (with one input, one output, and two hidden layers with three hidden units each):

$$\begin{aligned} &\text{Hidden layer 1:} &&\text{Hidden layer 2:} \\ &h_1 = a[\theta_{10} + \theta_{11}x] \qquad &&h_1' = a[\psi_{10} + \psi_{11}h_1 + \psi_{12}h_2 + \psi_{13}h_3] \\ &h_2 = a[\theta_{20} + \theta_{21}x] &&h_2' = a[\psi_{20} + \psi_{21}h_1 + \psi_{22}h_2 + \psi_{23}h_3] \\ &h_3 = a[\theta_{30} + \theta_{31}x] &&h_3' = a[\psi_{30} + \psi_{31}h_1 + \psi_{32}h_2 + \psi_{33}h_3], \end{aligned} \qquad (2)$$

$$\text{Output:} \quad y' = \phi_0' + \phi_1' h_1' + \phi_2' h_2' + \phi_3' h_3'.$$

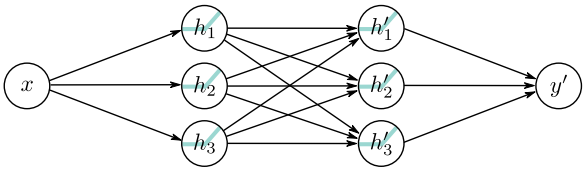

Figure 14: Deep neural network with two hidden layers (Prince 2024, 45)

- *Note:* The deep neural network still describes one equation relating input x to output y.

Deep Neural Networks

Connected Shallow NN – Special Case of Deep NN (Cont.)

- The above deep neural network represents the family of functions created by the two connected shallow neural networks if the slope parameters $\boldsymbol{\psi}$ in Equation 2 are:

$$\begin{aligned} \psi_{10} &= \theta_{10}^{\prime} + \theta_{11}^{\prime}\phi_{0} \\ \psi_{11} &= \theta_{11}^{\prime}\phi_{1} \\ \psi_{12} &= \theta_{11}^{\prime}\phi_{2} \\ \psi_{13} &= \theta_{11}^{\prime}\phi_{3} \\ &\vdots \end{aligned} \qquad (3)$$

- The deep neural network can therefore represent a broader family of functions because the slope parameters $\boldsymbol{\psi}$ in Equation 2 can take arbitrary values (i.e., they do not have to be constrained as in Equation 3).

Deep Neural Networks

Hyperparameters

- No. of hidden layers, K, is called the depth of the network.
- No. of hidden units in each layer, $D_{1}, D_{2}, \dotsc, D_{K}$, is called the width of the network. Total no. of hidden units is a measure of the capacity of the network.
- These parameters are hyperparameters $\rightarrow$ they are chosen before training the network.
- We can retrain the network with different hyperparameters $\rightarrow$ hyperparameter optimization / hyperparameter search.

Deep Neural Networks

General Formulation in Matrix Notation

- We can describe a general deep neural network, $\boldsymbol{y} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$, with K layers and $\boldsymbol{\phi} = \{\boldsymbol{\beta}_k, \boldsymbol{\Omega}_k\}_{k = 0}^{K}$ as:

$$\begin{aligned}
\mathbf{h}_1 &= \mathbf{a}[\boldsymbol{\beta}_0 + \boldsymbol{\Omega}_0\mathbf{x}] \\
\mathbf{h}_2 &= \mathbf{a}[\boldsymbol{\beta}_1 + \boldsymbol{\Omega}_1\mathbf{h}_1] \\
\mathbf{h}_3 &= \mathbf{a}[\boldsymbol{\beta}_2 + \boldsymbol{\Omega}_2\mathbf{h}_2] \\
&\vdots \\
\mathbf{h}_K &= \mathbf{a}[\boldsymbol{\beta}_{K-1} + \boldsymbol{\Omega}_{K-1}\mathbf{h}_{K-1}] \\
\mathbf{y} &= \boldsymbol{\beta}_K + \boldsymbol{\Omega}_K\mathbf{h}_K,
\end{aligned}$$

Figure 15: Example of a deep neural network with three hidden layers

Deep Neural Networks

Deep NNs vs. Shallow NNs

- Both deep and shallow neural networks can approximate any continuous function, but the former can often do so (much) more efficiently (i.e., with fewer parameters). This is called the depth efficiency of neural networks.
- For some structured inputs (e.g., images), we want to process local input regions in parallel and then gradually integrate information from increasingly larger regions. Such local-to-global processing is difficult to achieve without multiple layers.
- It is easier to train (moderately) deep neural networks than it is to train shallow ones.
- Deep neural networks seem to generalize better to new data than shallow neural networks.

Where Are We Now? 😲

We so far discussed:

- **Models:**
  1. Linear regression
  2. Shallow neural networks
  3. Deep neural networks

  → Each model is a family of functions mapping inputs to outputs.

  → The model parameters $\boldsymbol{\phi}$ determine the particular function of a family.

Up next:

- **Loss functions:**

  → Measuring how well a model describes the training data for a given set of parameters $\boldsymbol{\phi}$.

Loss Functions

Loss Functions

- When we train a model, we seek the model parameters $\boldsymbol{\phi}$ that produce the best possible mapping from inputs to outputs.
- To formalize "best possible" mapping, we need:
  - → a traininig data set of N input/output pairs, $\mathcal{S} = \{(\mathbf{x}_i, \mathbf{y}_i)\}_{i = 1}^{N}$,
  - → a loss function $L[\boldsymbol{\phi}]$ that measures the mismatch between the model predictions $\mathbf{f}[\mathbf{x}_i, \boldsymbol{\phi}]$ and the training outputs $\mathbf{y}_i$.
- During training, we seek parameter values $\boldsymbol{\phi}$ that minimize the loss function (and hence map the training inputs to the outputs as closely as possible).
- We need a framework that allows us to build loss functions for different prediction tasks, such as
  - → univariate regression (where $y_i \in \mathbb{R}$),
  - → multivariate regression (where $\mathbf{y}_i \in \mathbb{R}^{D_o}$),
  - → binary classification (where $y_i \in \{0, 1\}$),
  - → multiclass classification (where $y_i \in \{1, 2, \dotsc, K\}$).
- A commonly used framework is maximum likelihood, which gives us a principled approach to building loss functions.

Loss Functions

Maximum Likelihood

- So far, we assumed that a model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ directly predicts output $\boldsymbol{y}$ from input $\boldsymbol{x}$.
- The maximum likelihood framework takes a different perspective:
  - → a model computes a conditional probability distribution $Pr(\boldsymbol{y} | \boldsymbol{x})$ over possible outputs $\boldsymbol{y}$ given input $\boldsymbol{x}$;
  - → the loss function $L[\boldsymbol{\phi}]$ serves to maximize the likelihood of the observed data under the probability distribution $Pr(\boldsymbol{y} | \boldsymbol{x})$.

Steps for Constructing a Loss Function

- Choose a probability distribution $Pr(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta})$ defined on the output domain \

($\mathbf{y}$) with parameters $\boldsymbol{\theta}$).

- Set model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ to predict distribution parameters $\boldsymbol{\theta}$, so that $\boldsymbol{\theta} = \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ and $Pr(\mathbf{y} | \mathbf{x}; \boldsymbol{\theta}) = Pr(\mathbf{y} | \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}])$.
- Train the model on $\mathcal{S} = \{(\mathbf{x}_{i}, \mathbf{y}_{i})\}_{i = 1}^{N}$ by finding the model parameters $\boldsymbol{\hat{\phi}}$ that maximize the combined probability $\prod_{i = 1}^{N}Pr(\mathbf{y}_{i} | \mathbf{f}[\mathbf{x}_{i}, \boldsymbol{\phi}])$, or minimize the negative log-likelihood (w/ *i.i.d.* data),

$$
\begin{split}
\boldsymbol{\hat{\phi}} &= \underset{\boldsymbol{\phi}}{\color{blue}{\mathrm{argmin}}}\Bigg[\color{purple}{\underbrace{-\sum_{i = 1}^{N}\log\big[Pr(\mathbf{y}_{i} | \mathbf{f}[\mathbf{x}_{i}, \boldsymbol{\phi}])\big]}_{\text{negative log-likelihood (loss)}}}\Bigg] \\
&= \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\big[L[\boldsymbol{\phi}]\big].
\end{split}
$$

Loss Functions

Distributions for Different Prediction Tasks

Common Probability Distributions for Different Prediction Tasks

Prediction task	Output domain	Distribution
Univariate regression	$y \in \mathbb{R}$	Univariate normal
Multivariate regression	$\mathbf{y} \in \mathbb{R}^{D_{o}}$	Multivariate normal
Binary classification	$y \in \{0, 1\}$	Bernoulli
Multiclass classification	$y \in \{1, 2, \dotsc, K\}$	Categorical

Where Are We Now? 😲

We so far discussed:

- **Models:**
  1. Linear regression
  2. Shallow neural networks
  3. Deep neural networks

  → Each model is a family of functions mapping input to output.

  → The model parameters $\boldsymbol{\phi}$ determine the particular function of a family.

- **Loss functions:**

  → Measuring how well a model describes the training data for a given set of parameters $\boldsymbol{\phi}$.

Up next:

- **Model training:**

  → Aiming to find the model parameters $\boldsymbol{\phi}$ that minimize the loss function.

Fitting Models

Fitting Models

- To fit a model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$, we need a training data set of input/output pairs, $\mathcal{S} = \{(\mathbf{x}_{i}, \mathbf{y}_{i})\}_{i = 1}^{N}$.
- We seek parameters $\boldsymbol{\phi}$ for the model $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ that map the inputs $\mathbf{x}_{i}$ to the outputs $\mathbf{y}_{i}$ as well as possible.
- We therefore define a loss function $L[\boldsymbol{\phi}]$ that quantifies the mismatch in this mapping.
- We also need an optimization algorithm to find the parameters $\boldsymbol{\hat{\phi}}$ that minimize the loss,

$$\boldsymbol{\hat{\phi}} = \underset{\boldsymbol{\phi}}{\mathrm{argmin}}\big[L[\boldsymbol{\phi}]\big].$$

- The simplest (iterative) optimization algorithm is gradient descent.

Fitting Models

Gradient Descent (GD)

- The gradient descent (GD) algorithm starts by choosing initial parameters $\boldsymbol{\phi} = [\phi_{0}, \phi_{1}, \dotsc, \phi_{M}]^{T}$.
- It then iterates two steps (until it converges):
  - → **Step 1.** Compute the derivatives of the loss w.r.t. the parameters at their position in iteration t, $$ \frac{\partial L[\boldsymbol{\phi}_{t}]}{\partial \boldsymbol{\phi}} = \left[\begin{matrix} \frac{\partial L[\boldsymbol{\phi}_{t}]}{\partial \phi_{0}} \\ \frac{\partial L[\boldsymbol{\phi}_{t}]}{\partial \phi_{1}} \\ \vdots \\ \frac{\partial L[\boldsymbol{\phi}_{t}]}{\partial \phi_{M}} \end{matrix} \right]. $$
  - → **Step 2.** Update the parameters in iteration $t + 1$ according to the rule $$ \boldsymbol{\phi}_{t + 1} \leftarrow \boldsymbol{\phi}_{t} - \alpha \frac{\partial L[\boldsymbol{\phi}_{t}]}{\partial \boldsymbol{\phi}}, $$ where $\alpha > 0$ determines the magnitude of the change (if fixed, it is called the learning rate).
- GD has converged when the gradient reaches zero and the parameters no longer change. In practice, the algorithm is typically stopped when the gradient magnitude falls below a predefined threshold.

Python/PyTorch

Demo 2: Gradient Descent

Notebook 1 in Google Colab

Fitting Models

GD Challenges

- For convex loss functions, there is a single (global) minimum (Figure 16 panel b).
  - → Wherever we initialize the parameters, GD will reach the minimum after sufficient iterations.
- Loss functions for non-linear models (such as shallow and deep NNs) are typically non-convex, i.e., they have local minima (Figure 16 panel a) and saddle points (Figure 16 panel c) in addition to the global minimum.
  - → The "minimum" reached by GD is determined by the initial values.
  - → Depending on the initial values, GD might end up at a local minimum or saddle point.

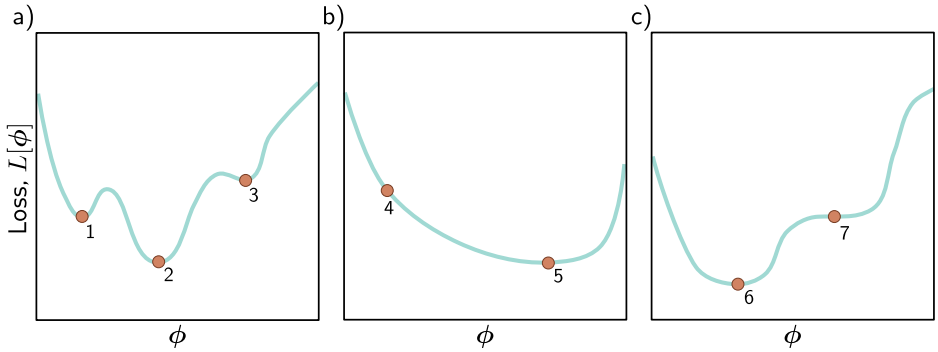

Figure 16: Problems of non-convex loss functions. The loss function in panel b) is convex, whereas the loss functions in panels a) and c) are non-convex (Prince 2024, 95)

Fitting Models

Stochastic Gradient Descent (SGD)

- Stochastic gradient descent (SGD) tries to overcome the problems of GD by adding noise to the gradient at each step.
  - → SGD still moves downhill on average, but at any iteration, its direction is not necessarily the steepest downhill direction, or not downhill at all.
  - → The ability to temporarily move uphill allows SGD to jump from a valley of the loss function.

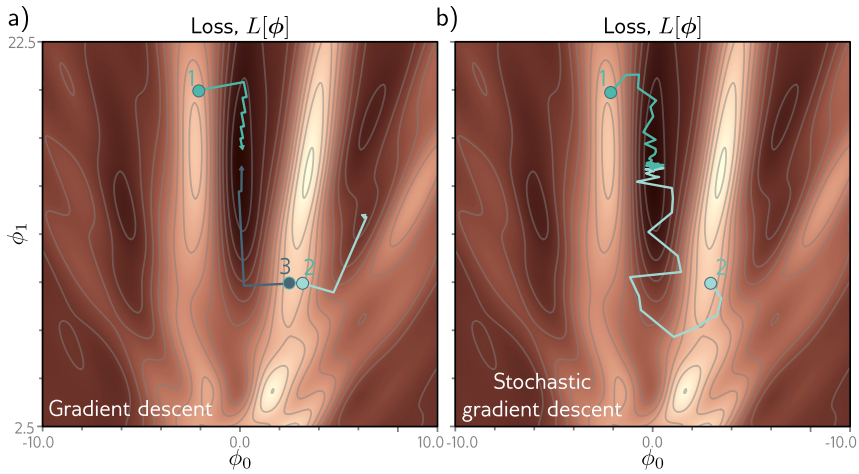

Figure 17: Gradient descent (panel a) vs. stochastic gradient descent (panel b) (Prince 2024, 83)

- *Panel a).* If GD is initialized at a “right” point on the loss function (e.g., points 1 and 3), it will move to the global minimum. If it is initialized at a “wrong” point (e.g., point 2), it will move to a local minimum.
- *Panel b).* By adding noise to the optimization process, SGD is able to escape from “wrong” valleys (e.g., point 2) and reach the global minimum

Fitting Models

Stochastic Gradient Descent (SGD) (Cont.)

- SGD introduces randomness by computing the gradient based on only a random subset (batch) of the training data at each iteration, $$\boldsymbol{\phi}_{t + 1} \leftarrow \boldsymbol{\phi}_{t} - \alpha \sum_{i \in \mathcal{B}_{t}}\frac{\partial \ell_{i} [\boldsymbol{\phi}_{t}]}{\partial \boldsymbol{\phi}},$$ where $\mathcal{B}_{t}$ is the set of indices of input/output pairs in the batch at the tth iteration and $1 \leq |\mathcal{B}_{t}| \leq N$.
- SGD draws batches without replacement from the training data set until it has used all the data.

  → Therefore, all training examples contribute equally to the solution.
- After SGD has iterated through the training data set, it repeats the process. A single pass through the training data set is called an epoch.
- SGD often uses a learning rate schedule: The learning rate α starts at a high value and is decreased over epochs.

Fitting Models

SGD with Momentum

- A modification to SGD is to add a momentum term, $$\begin{split} \mathbf{m}_{t + 1} &\leftarrow \beta \cdot \mathbf{m}_{t} + (1 - \beta) \sum_{i \in \mathcal{B}_{t}}\frac{\partial \ell_{i}[\boldsymbol{\phi}_{t}]}{\partial \boldsymbol{\phi}} \\ \boldsymbol{\phi}_{t + 1} &\leftarrow \boldsymbol{\phi}_{t} - \alpha \cdot \mathbf{m}_{t + 1}, \end{split}$$ where:
  - → $\mathbf{m}_{t}$ is the momentum, which captures change made to the parameters in the previous step t,
  - → $\beta \in [0, 1)$ is the momentum coefficient and controls the degree to which the gradient is smoothed over time,
  - → α is the learning rate.
- We therefore update the parameters with a weighted combination of the gradient computed from the current batch and the previous change.

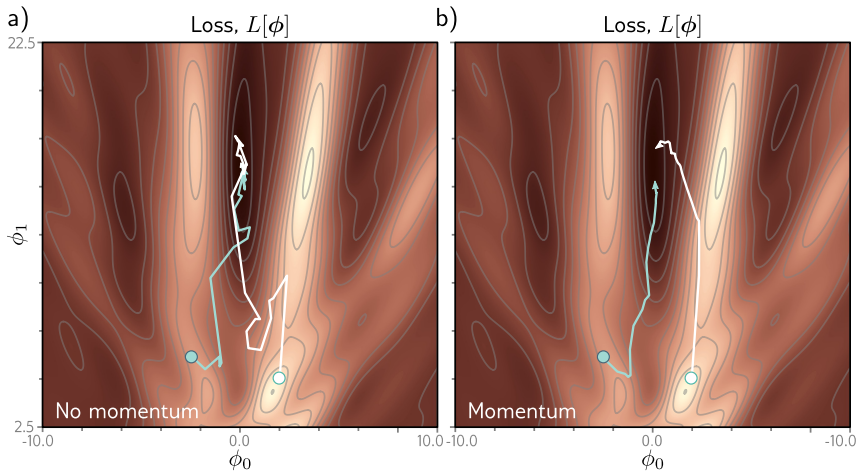

Figure 18: Stochastic gradient descent without momentum (panel a) and with momentum (panel b) (Prince 2024, 87). Note that the momentum leads to a smoother trajectory (this is because the terms in the sum cancel out if the gradient rapidly changes direction over iterations.)

Fitting Models

GD with Normalized Gradients

- GD with a fixed learning rate \(\alpha\) makes
  - → large changes to parameters associated with large gradients,
  - → small changes to parameters associated with small gradients.
- This problem can be overcome by normalizing the gradients when updating the parameters, $$\begin{split} \mathbf{m}_{t + 1} &\leftarrow \frac{\partial L[\boldsymbol{\phi}_{t}]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t + 1} &\leftarrow \bigg(\frac{\partial L[\boldsymbol{\phi}_{t}]}{\partial \boldsymbol{\phi}}\bigg)^{2} \\ \boldsymbol{\phi}_{t + 1} &\leftarrow \boldsymbol{\phi}_{t} - \alpha \frac{\mathbf{m}_{t + 1}}{\sqrt{\mathbf{v}_{t + 1}} + \delta}, \end{split}$$ where \(\delta\) is a small constant preventing division by zero when the gradient magnitude is 0.

Figure 19: GD with normalized gradients

- Normalizing the gradient only retains the sign. The algorithm moves a fixed distance \(\alpha\) along each coordinate in the direction determined by the sign.
- The algorithm will typically not converge but bounce back and forth around the minimum.

Fitting Models

Adaptive Moment Estimation (Adam)

- Adaptive moment estimation (Adam) uses normalized gradients but adds momentum to the gradient and the squared gradient to overcome the non-convergence problem: $$\begin{split} \mathbf{m}_{t + 1} &\leftarrow \beta \cdot \mathbf{m}_{t} + (1 - \beta)\sum_{i \in \mathcal{B}_{t}}\frac{\partial \ell_{i}[\boldsymbol{\phi}_{t}]}{\partial \boldsymbol{\phi}} \\ \mathbf{v}_{t + 1} &\leftarrow \gamma \cdot \mathbf{v}_{t} + (1 - \gamma)\bigg(\sum_{i \in \mathcal{B}_{t}}\frac{\partial \ell_{i}[\boldsymbol{\phi}_{t}]}{\partial \boldsymbol{\phi}}\bigg)^{2}, \end{split}$$ where $\beta, \gamma \in [0, 1)$ are the momentum coefficients. Typical values for the momentum coefficients are $\beta = 0.9$ and $\gamma = 0.99$ (Bishop and Bishop 2024, 224).
- The statistics $\mathbf{m}_{t}$ and $\mathbf{v}_{t}$ are initialized to 0. This results in the problem that $\mathbf{m}_{t + 1}$ and $\mathbf{v}_{t + 1}$ are biased towards zero during the early steps in the iterative procedure.

Fitting Models

Adaptive Moment Estimation (Adam) (Cont.)

- To counteract this bias, Adam modifies $\mathbf{m}_{t + 1}$ and $\mathbf{v}_{t + 1}$ before updating the parameters: $$\begin{split} \mathbf{\tilde{m}}_{t + 1} \leftarrow \frac{\mathbf{m}_{t + 1}}{1 - \beta^{t + 1}}&, \quad \mathbf{\tilde{v}}_{t + 1} \leftarrow \frac{\mathbf{v}_{t + 1}}{1 - \gamma^{t + 1}} \\ \boldsymbol{\phi}_{t + 1} \leftarrow \boldsymbol{\phi}_{t} - \alpha & \frac{\mathbf{\tilde{m}}_{t + 1}}{\sqrt{\mathbf{\tilde{v}}_{t + 1}} + \delta}. \end{split}$$
- Since $\beta, \gamma \in [0, 1)$, the denominators $(1 - \beta^{t + 1})$ and $(1 - \gamma^{t + 1})$ approache 1 as t increases, and the bias corrections have a diminishing effect.
- SGD and Adam are the most widely used optimization algorithms in deep learning.[1]

Figure 20: Adam uses momentum in the gradient and squared gradient.

Fitting Models

Hyperparameters

- We previously discussed hyperparameters governing the network architecture:
  - → no. of hidden layers, K,
  - → no. of hidden units in each layer, $D_{1}, D_{2}, \dotsc, D_{K}$.
- We now discussed further hyperparameters, which govern the behavior of the learning algorithm:
  - → batch size $|\mathcal{B}_{t}|$,
  - → learning rate, α,
  - → momentum coefficients, β and γ.
- Training models with different sets of hyperparameters and choosing the best ones is called hyperparameter search.

Where Are We Now? 😲

We so far discussed:

- **Models:**
  1. Linear regression
  2. Shallow neural networks
  3. Deep neural networks

  → Each model is a family of functions mapping input to output.

  → The model parameters $\boldsymbol{\phi}$ determine the particular function of a family.
- **Loss functions:**

  → Measuring how well a model describes the training data for a given set of parameters $\boldsymbol{\phi}$.
- **Model training:**

  → Aiming to find the model parameters $\boldsymbol{\phi}$ that minimize the loss function.

Up next:

- **Model training for neural networks:**

  → Computing gradients w.r.t. the parameters of a neural network.

  → Initializing the network parameters before optimization begins.

1. There are many other widely used algorithms that improve upon GD. Most notable is the Newton method, which takes into account the curvature of the surface by using the inverse of the Hessian matrix. However, deep neural networks have a large number of parameters, which makes computing the inverse Hessian intractable.

Gradients and Initialization

Gradients and Initialization

- Iterative optimization algorithms such as GD are general-purpose methods for finding the minimum of a function.
- In the context of neural networks, we use these methods to find parameters that minimize the loss function.
- Two issues require consideration when we use these methods to train neural networks.
  - → Efficient calculation of the gradients:
    - ⇾ *Problem:* Deep neural networks can have a large number of parameters and the gradient needs to be computed for every parameter at every iteration of the training algorithm.
    - ⇾ *Solution:* The backpropagation algorithm computes gradients efficiently.
  - → Initialization of the model parameters before optimization:
    - ⇾ *Problem:* If the parameters are not initialized sensibly, the pre-activations can become very small or large, and the gradient magnitudes can decrease or increase uncontrollably during the training process (vanishing and exploding gradient problem).
    - ⇾ *Solution:* Different solutions addressing this problem have been proposed (e.g., He initialization for ReLU activation functions). Deep learning frameworks have reasonable defaults for initializing parameters of regular neural networks.

Gradients and Initialization

Backpropagation Algorithm

- Consider a network $\mathbf{f}[\mathbf{x}, \boldsymbol{\phi}]$ with input $\mathbf{x}$, parameters $\boldsymbol{\phi}$, and three hidden layers h_{1}, h_{2}, and h_{3}: $$ \begin{split} \mathrm{h}_{1} &= \mathrm{a}[\boldsymbol{\beta}_{0} + \boldsymbol{\Omega}_{0}\mathrm{x}] \\ \mathrm{h}_{2} &= \mathrm{a}[\boldsymbol{\beta}_{1} + \boldsymbol{\Omega}_{1}\mathrm{h}_{1}] \\ \mathrm{h}_{3} &= \mathrm{a}[\boldsymbol{\beta}_{2} + \boldsymbol{\Omega}_{2}\mathrm{h}_{2}] \\ \mathbf{f}[\mathbf{x}, \boldsymbol{\phi}] &= \boldsymbol{\beta}_{3} + \boldsymbol{\Omega}_{3}\mathrm{h}_{3}, \end{split} $$ where $\boldsymbol{\phi} = \{\boldsymbol{\beta}_{0}, \boldsymbol{\Omega}_{0}, \boldsymbol{\beta}_{1}, \boldsymbol{\Omega}_{1}, \boldsymbol{\beta}_{2}, \boldsymbol{\Omega}_{2}, \boldsymbol{\beta}_{3}, \boldsymbol{\Omega}_{3}\}$ includes the bias vectors $\boldsymbol{\beta}_{k}$ and weight matrices $\boldsymbol{\Omega}_{k}$.

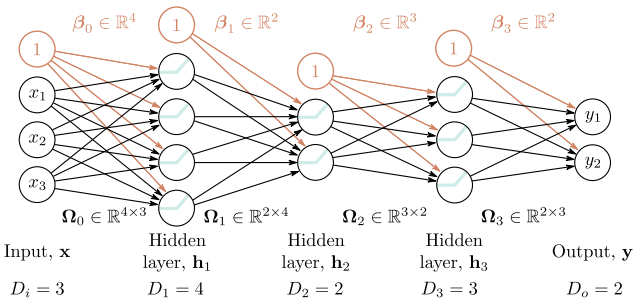

Figure 21: Deep neural network with three inputs, two outputs, and three hidden layers

Gradients and Initialization

Backpropagation Algorithm (Cont.)

- We have a loss function $L[\boldsymbol{\phi}]$, $$ L[\boldsymbol{\phi}] = \sum_{i = 1}^{N}\ell_{i} = L[f[\mathbf{x}_{i}, \boldsymbol{\phi}], y_{i}]. $$
- We use SGD as the optimization algorithm for training the network, $$ \boldsymbol{\phi}_{t + 1} \leftarrow \boldsymbol{\phi}_{t} - \alpha \sum_{i \in \mathcal{B}_{t}}\frac{\partial \ell_{i} [\boldsymbol{\phi}_{t}]}{\partial \boldsymbol{\phi}}, $$ where α is the learning rate and $\mathcal{B}_{t}$ contains the batch indices at iteration t.
- SGD requires us to calculate the derivatives $$ \frac{\partial \ell_{i}}{\partial \boldsymbol{\beta}_{k}} \quad \text{and} \quad \frac{\partial \ell_{i}}{\partial \boldsymbol{\Omega}_{k}}, $$ for the parameters $\{\boldsymbol{\beta}_{k}, \boldsymbol{\Omega}_{k}\}$ at every layer $k = 0, 1, \dotsc, K$ and for each index i in batch $\mathcal{B}_{t}$.

Gradients and Initialization

Backpropagation Algorithm (Cont.)

- The derivatives $\frac{\partial \ell_{i}}{\partial \boldsymbol{\beta}_{k}} \quad \text{and} \quad \frac{\partial \ell_{i}}{\partial \boldsymbol{\Omega}_{k}}$ tell us how the loss changes when we make a small change to the parameters $\boldsymbol{\beta}_{k}$ and $\boldsymbol{\Omega}_{k}$.
- We need to compute the derivatives of the loss ℓ_{i} w.r.t. each of the biases $\boldsymbol{\beta}_{k}$ and weights $\boldsymbol{\Omega}_{k}$.
- The backpropagation algorithm computes these derivatives via two steps:
  - → a forward pass, in which we compute and store intermediate values and the network output;
  - → a backward pass, in which we calculate the derivatives of each parameter (starting at the end of the network and reusing previous calculations as we move toward the beginning of the network).[1]

Gradients and Initialization

Backpropagation Forward Pass

- *Observation:* Each weight (element of $\boldsymbol{\Omega}_{k}$) multiplies the activation of a source hidden unit and adds the result to a destination hidden unit in the next layer.
- Therefore, any small change to the weight is amplified by the activation of the source hidden unit.
  → E.g., a weight in $\boldsymbol{\Omega}_{1}$ is applied to the second hidden unit in hidden layer $\mathbf{h}_{1}$ (the activation) and the result is added to the second hidden unit in hidden layer $\mathbf{h}_{2}$ (the pre-activation). If the value of the activation doubles, then the effect of a small change to the weight will double too (Figure 22).
- To compute the derivatives of the weights, we therefore need to calculate and store the activations of all hidden units. This is done in the forward pass.

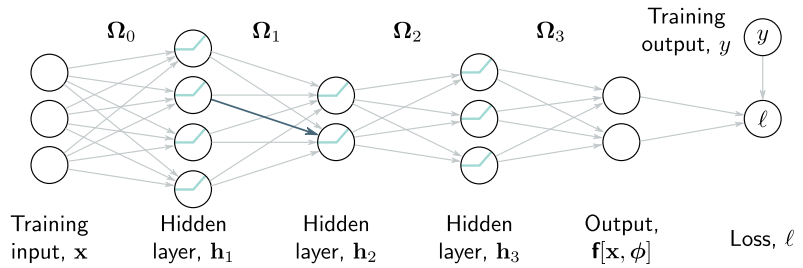

Figure 22: Backpropagation forward pass (Prince 2024, 98)

Gradients and Initialization

Backpropagation Backward Pass

- *Observation:* A small change in a bias or weight causes a ripple effect of changes through the subsequent network. E.g.,
  - → To know how a small change to a weight feeding into layer $\mathbf{h}_{3}$ changes the loss ℓ, 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 ℓ (Figure 23, panel a).
  - → To know how a small change to a weight feeding into layer $\mathbf{h}_{2}$ changes the loss ℓ, we need to know how the hidden unit in $\mathbf{h}_{2}$ changes $\mathbf{h}_{3}$, how $\mathbf{h}_{3}$ changes $\mathbf{f}$, and how $\mathbf{f}$ changes the loss ℓ (Figure 23, panel b).
  - → To know how a small change to a weight feeding into layer $\mathbf{h}_{1}$ changes the loss ℓ, we need to know how the hidden unit in $\mathbf{h}_{1}$ changes $\mathbf{h}_{2}$ and how these changes propagate through to the loss (Figure 23, panel c).
- Calculation of the same quantities is required when we consider other parameters in the same or earlier layers. We can therefore calculate them once and resuse them.
- The backward pass does this by first computing derivatives at the end of the network and reusing previous calculations when moving backward.[1]

Gradients and Initialization

Backpropagation Backward Pass (Cont.)

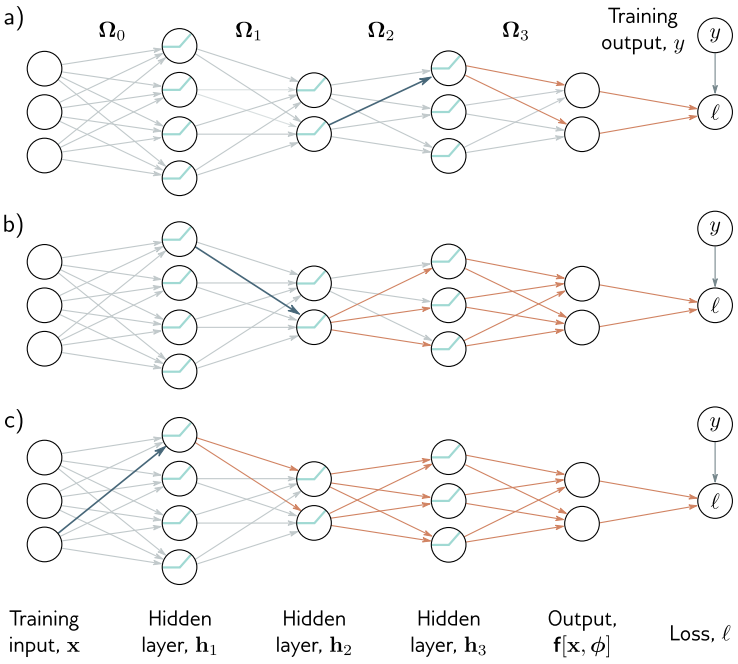

Figure 23: Backpropagation backward pass (Prince 2024, 99)

Where Are We Now? 😲

We so far discussed:

- **Models:**
  1. Linear regression
  2. Shallow neural networks
  3. Deep neural networks

  → The model parameters $\boldsymbol{\phi}$ determine the particular function of a family.

- **Loss functions:**

  → Measuring how well a model describes the training data for a given set of parameters $\boldsymbol{\phi}$.

- **Model training:**

  → Aiming to find the model parameters $\boldsymbol{\phi}$ that minimize the loss function.

- **Model training for neural networks:**

  → Computing gradients w.r.t. the parameters of a neural network.

Up next:

- **Model performance:**

  → Measuring how well a trained model performs in terms of generalizing to new data.

1. The backpropagation algorithm is extremely efficient: the most demanding computational step in both the forward and backward pass is matrix multiplication (which can be parallelized). However, it is not memory efficient: the intermediate values in the forward pass must all be stored, which can limit the size of the model we can train.

1. See pp. 100-103 in Prince (2024) for a walkthrough of backpropagation for a simple toy model.

Model Performance

Model Performance

- A neural network with sufficient capacity (no. of hidden units) will often perform perfectly on the training data.
- However, this does not mean that it will generalize well to new test data, i.e., have a low test error.
- There are three possible sources of test error:
  - → noise,
  - → bias,
  - → variance.

Model Performance

Sources of Test Error

Noise

Noise is inherent randomness/uncertainty in the data generation process.

Bias

Bias is the systematic deviation of our model from the mean of the true function we are approximating.

Variance

Variance is the uncertainty in our fitted model due to the particular training data set we sample.

Model Performance

Sources of Test Error (Cont.)

- The three sources noise, bias, and variance characterize the (expected) test error of any prediction model.
- For regression models (where the noise is additive with variance σ^{2}) and a least squares loss, they combine additively: $$E_{\mathcal{S}}\Big[E_{y}\big[L[x]\big]\Big] = \underbrace{E_{\mathcal{S}}\Big[\big(f[x, \boldsymbol{\phi}[\mathcal{S}]] - f_{\mu}[x]\big)^{2}\Big]}_{\text{Variance}} + \underbrace{\big(f_{\mu}[x] - \mu[x]\big)^{2}}_{\text{Bias}} + \underbrace{\sigma^{2}}_{\text{Noise}},$$ where
  - → $E_{\mathcal{S}}\Big[E_{y}\big[L[x]\big]\Big]$ is the expected loss after considering the uncertainty in the training data set $\mathcal{S} = \{(x_{i}, y_{i})\}_{i = 1}^{N}$ and the test data y,
  - → $f_{\mu}[x] = E_{\mathcal{S}}\big[f[x, \boldsymbol{\phi}[\mathcal{S}]]\big]$ is the expected model output with respect to all possible training data sets $\mathcal{S}$,
  - → $\mu[x] = E_{y}[y[x]] = \int y[x]\Pr(y|x)dy$ is the conditional expectation of y given x (since the data generation process is noisy, we can observe different outputs y for the same input x; so for each x, there is a distribution $\Pr(y|x)$).

Model Performance

Sources of Test Error (Cont.)

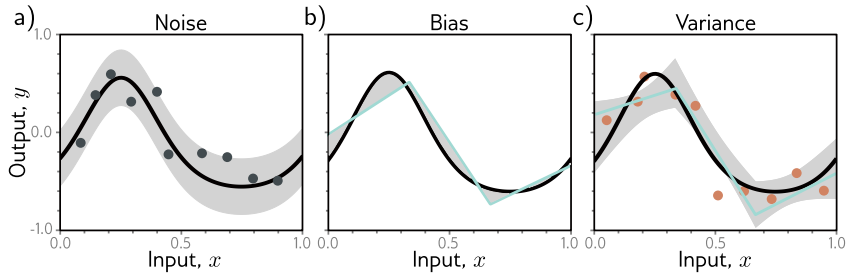

Figure 24: Sources of test error: noise (panel a), bias (panel b), and variance (panel c) (Prince 2024, 122)

- *Panel a) Noise.* Data generation is noisy. Even if the model exactly fits the true underlying function (black line), the noise in the test data (black points) means that some error will remain.
- *Panel b) Bias.* Even with the best possible parameters, the three-region model (cyan line) cannot exactly fit the true function (black line).
- *Panel c) Variance.* In practice, we have limited noisy training data (orange points). When we fit the model, we don't recover the best possible function from panel b) but a slightly different function (cyan line) that reflects idiosyncrasies of the training data.

Model Performance

Reducing Sources of Test Error

Noise

This source of error is irreducible and represents the upper limit on expected model performance.

Bias

Can be reduced by making the model more flexible (e.g., increasing network capacity by adding more hidden units).

Variance

Can be reduced by increasing the quantity of training data or, for a fixed-size training data set, making the model less flexible.

Bias-Variance Trade-Off

For a fixed-size training data set, increasing model flexibility may decrease the bias, but at the cost of increasing the variance. \(\rightarrow\) Hence, there is an optimal model flexibility.

Model Performance

Bias-Variance Trade-Off

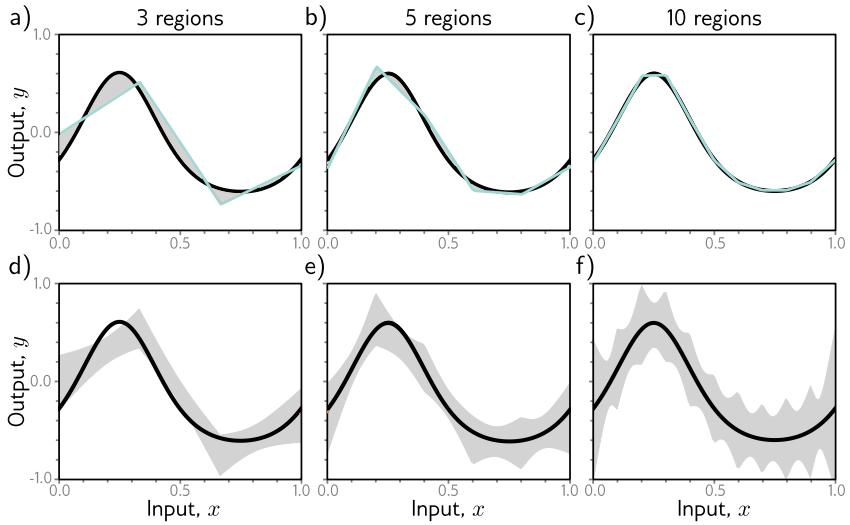

Figure 25: Bias and variance as a function of model capacity. *Panels a)-c).* As model capacity increases, bias decreases. *Panels d)-f).* As model capacity increases, variance increases (due to overfitting!) (Prince 2024, 127)

Model Performance

Overfitting (Fitting Noise in Training Data)

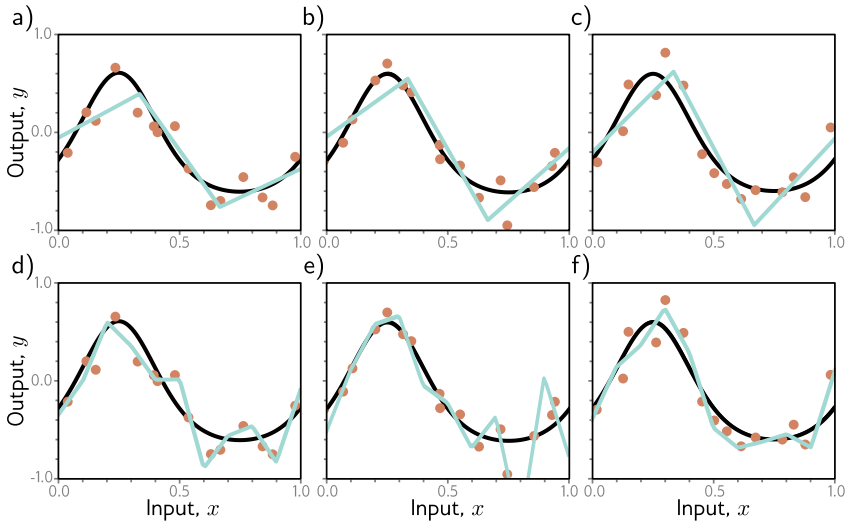

Figure 26: Overfitting. *Panels a)-c).* Model with three regions is fit to three different data sets of 15 points each. The result is similar in all cases (low variance). *Panels d)-f).* Model with ten regions is fit to the same data sets. They fit the training data better, but not the true function (overfitting, large variance) (Prince 2024, 128)

Model Performance

Optimal Bias-Variance Trade-Off

- As the model capacity increases, the bias decreases, but the variance increases.
- There is a sweet spot where the combination of bias and variance reaches its minimum.

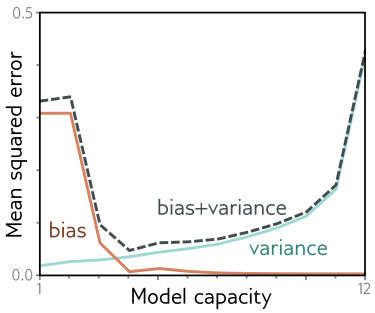

Figure 27: Optimal bias-variance trade-off (Prince 2024, 128)

Model Performance

Double Descent

- For some data sets, loss functions, and models, the test error shows a pattern different from the one predicted by the bias-variance trade-off:
  - → the test error first decreases and then increases as we approach the point where the training data are memorized (classical/under-parameterized regime);
  - → subsequently, the test error decreases again and may reach a better performance level, thus exhibiting a double descent (modern/over-parameterized regime).[1]

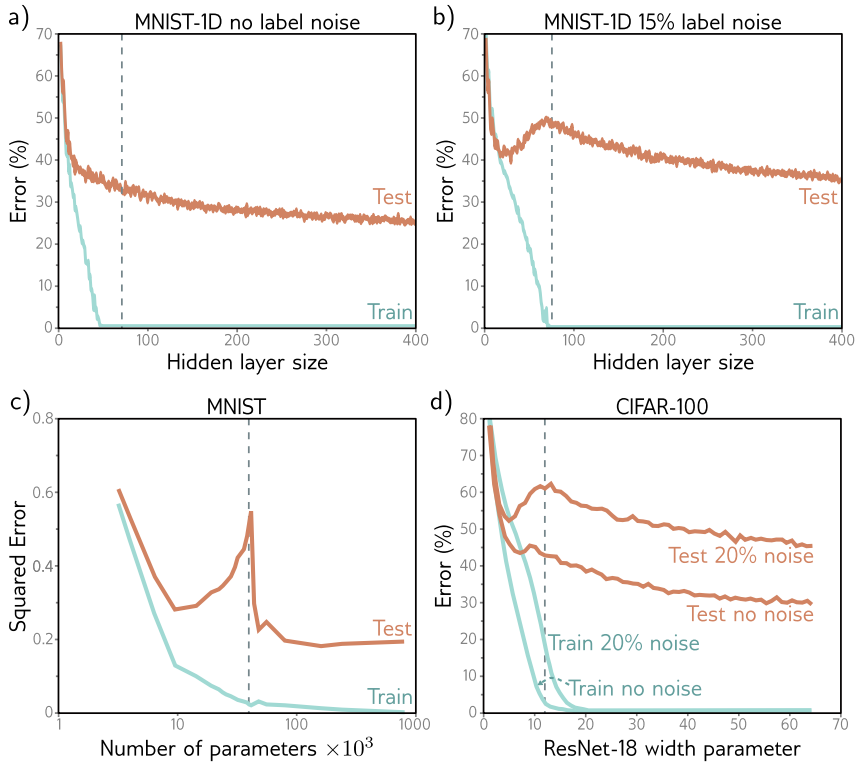

Figure 28: Double descent. The dashed line indicates that the model parameters equal the number of training examples (Prince 2024, 130)

Model Performance

Choosing Hyperparameters

- We saw previously that the test performance of a model changes with its capacity.
- In practice, how do we find the model capacity that leads to optimal test performance?
  - → In the classical regime, we know neither the bias (requires knowledge of the true function) nor the variance (requires multiple independently sampled data sets).
  - → In the moderrn regime, we do not know how much capacity to add before the test error stops improving.
- The capacity of a deep neural network is governed by its hyperparameters (no. of hidden layers, no. of hidden units per hidden layer, etc.).
- We take an empirical approach to finding the best set of hyperparameters (hyperparameter search).
  1. We train models with different hyperparameters on the same training set.
  2. We measure the performance of the fitted models on an independent validation set.
  3. We select the model that performed best on the validation set and measure its performance on an independent test set (to obtain an estimate of the true performance).
  - → This approch is called cross-validation. When training data are limited, we often use an approach called k-fold cross-validation.

1. Overparameterized means that there are not enough training data points to constrain the model parameters uniquely.

Python/PyTorch

Demo 3: Fully Connected Deep Neural Networks

Notebook 2 in Google Colab

Where Are We Now? 😲

We so far discussed:

- **Models:**
  1. Linear regression
  2. Shallow neural networks
  3. Deep neural networks
- **Loss functions:**
  - → Measuring how well a model describes the training data for a given set of parameters $\boldsymbol{\phi}$.
- **Model training in general & Model training for neural networks:**
  - → Aiming to find the model parameters $\boldsymbol{\phi}$ that minimize the loss function.
  - → Computing gradients w.r.t. the parameters of a neural network.
- **Model performance:**
  - → Measuring how well a trained model performs in terms of generalizing to new data.

Up next:

- **Convolutional Networks & Transformers:**
  - → Network architectures specialized for processing images and text data.

Convolutional Networks

Convolutional Networks

- We so far discussed fully connected neural networks with a single path from input to output.
- Convolutional neural networks (CNNs) are a more specialized type of network with sparser connections and shared weights.
- CNNs are mainly used for processing image data.
- Images have three properties that call for a more specialized network architecture:
  - → they are high-dimensional (e.g., an 224 \(\times\) 224 RGB image has 150,528 inputs);
  - → nearby image pixels are related, but fully connected networks have no notion of "nearby;"
  - → the interpretation of an image is stable under geometric transformations, yet fully connected networks do not exploit this (no invariance and equivariance).
- CNNs use fewer parameters than fully connected networks.
  - → They process local image regions and share parameters across the whole image.

Convolutional Networks

Invariance and Equivarance

- A function $\mathbf{f}[\mathbf{x}]$ is invariant to a transformation $\mathbf{t}[\cdot]$ if: $$\mathbf{f}\big[\mathbf{t}[\mathbf{x}]\big] = \mathbf{f}[\mathbf{x}].$$
  → The output of the function is the same regardless of whether transformation $\mathbf{t}[\cdot]$ is applied to $\mathbf{x}$.
- A function $\mathbf{f}[\mathbf{x}]$ is equivariant to a transformation $\mathbf{t}[\cdot]$ if: $$\mathbf{f}\big[\mathbf{t}[\mathbf{x}]\big] = \mathbf{t}\big[\mathbf{f}[\mathbf{x}]\big].$$
  → The output of the function changes in the same way under the transformation as the input.
- Networks for image classification should be invariant to geometric transformations of an image.
  → Network $\mathbf{f}[\mathbf{x}]$ should identify an image $\mathbf{x}$ as containing the same object, even if it is translated.
- Networks for image segmentation should be equivariant to geometric transformations of an image.
  → If an image $\mathbf{x}$ is translated, network $\mathbf{f}[\mathbf{x}]$ should return a segmentation that is transformed similarly.[1]

Convolutional Networks

Invariance and Equivarance (Cont.)

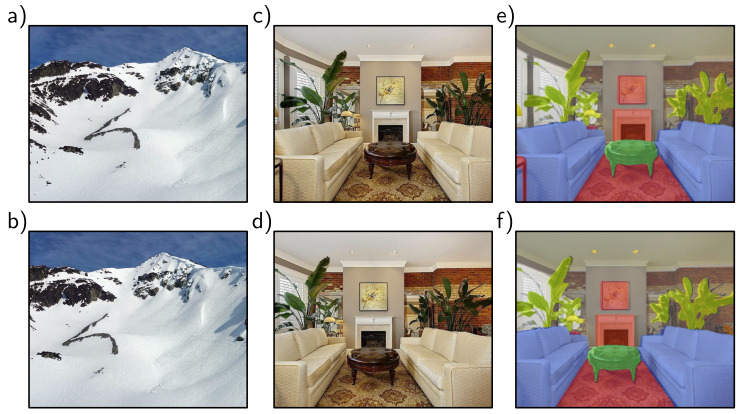

Figure 29: Network invariance and equivariance (Prince 2024, 162)

- Figure 29, Image classification, *panels a)-b)*. The goal is to categorize both images as “mountain” regardless of the horizontal shift. The network prediction should be invariant to the transformation.
- Figure 29, Image segmentation, *panels c)-f)*. The goal is to associate a label with each pixel. When the input is transformed, the output (color overlay) should be equivariant (be transformed in the same way).

Convolutional Networks

Convolutional Networks for 1D Inputs

- Convolutional networks consist of a series of convolutional layers:
  - → each of them is equivariant to translation;
  - → they include pooling mechanisms that induce partial invariance to translation.[1]
- Convolutional layers perform a convolution operation, which transforms an input vector $\mathbf{x}$ into an output vector $\mathbf{z}$ so that each z_{i} is a weighted sum of the nearby inputs. E.g., $$ z_{i} = \omega_{1}x_{i - 1} + \omega_{2}x_{i} + \omega_{3}x_{i + 1}, \qquad(4)$$ where $\boldsymbol{\omega} = [\omega_{1}, \omega_{2}, \omega_{3}]^{T}$ is called the convolution kernel and the region over which inputs are combined is called the kernel size.
- Each convolution layer employs parameter sharing (all hidden units in a layer share the same weights), so local image patches are processed similarly at every position in the image.
  - → This induces translation equivariance: if a local patch of an image produces a particular response in the unit connected to the patch, then the same set of pixel values at a different location will produce the same response.

Convolutional Networks

Padding and Valid Convolutions

- Equation 4 shows that each output \(z_{i}\) is a weighted sum of the three nearest inputs \(x_{i - 1}\), \(x_{i}\), and \(x_{i + 1}\).
- The first output does not have an input at position \(i - 1\), while the last output does not have an input at position \(i + 1\).
- There are two approaches to address this problem:
  - → Zero padding: assumes the input is zero outside its valid range (Figure 30, panel c).
  - → Valid convolutions: only computes outputs with a valid input range; this introduces no extra information at the input edges but reduces the output size relative to the input (Figure 30, panel d).

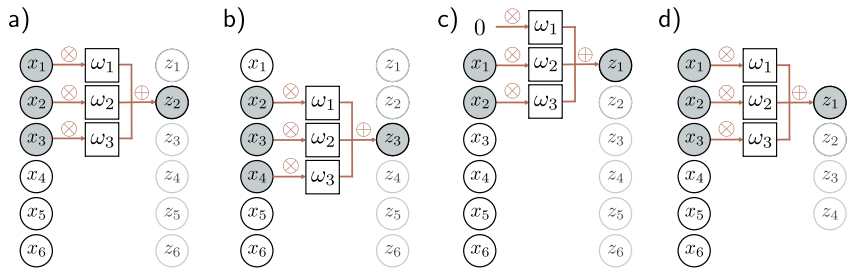

Figure 30: Zero padding and valid convolutions (Prince 2024, 163)

Convolutional Networks

Stride, Kernel Size, and Dilation

- **Stride.** For a stride of \(k\), the kernel is evaluated at every \(k\)th position in the input vector (Figure 31, panels a)-b): first output \(z_{1}\) has kernel centered at \(x_{1}\), second output \(z_{2}\) has kernel centered at \(x_{3}\), etc.).

  → Decreases size of output relative to input.

- **Kernel Size.** Kernel size can be increased (Figure 31, panel c).

  → Combines information from a larger area, but requires more parameters.

- **Dilation.** Intersperse kernel values with zeros (no. of zeros is called dilation rate) (Figure 31, panel d).

  → Still combines information from a larger area, but needs fewer parameters.

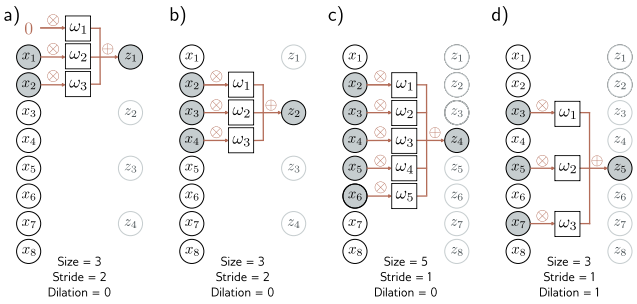

Figure 31: Stride, kernel size, and dilation (Prince 2024, 164)

Convolutional Networks

Convolutional vs. Fully-Connected Layers

- A convolutional layer computes its output by convolving the input, adding a bias β, and passing each result through an activation function $a[\cdot]$.
- Hidden unit h_{i} in a convolutional layer: $$\begin{split} h_{i} &= a[\beta + \omega_{1}x_{i - 1} + \omega_{2}x_{i} + \omega_{3}x_{i + 1}] \\ &= a \Bigg[\beta + \sum_{j = 1}^{3}\omega_{j}x_{i + j - 2}\Bigg]. \end{split}$$
  → 3 weights, 1 bias (note that parameters are shared across hidden units in the same hidden layer).
- Hidden unit h_{i} in a fully-connected layer: $$h_{i} = a \Bigg[\beta_{i} + \sum_{j = 1}^{D}\omega_{ij}x_{j}\Bigg].$$
  → D^{2} weights, D biases, where D is the number of inputs and the number of hidden units.
- A convolutional layer is a special case of a fully-connected layer (with most parameters set to zero and others constrained to be identical).

Convolutional Networks

Channels

- If we only apply a single convolution, information will be lost (since we average nearby inputs and the ReLU activation function clips results that are below zero).
- Therefore, several convolutions are typically used to produce sets of hidden units (channels).

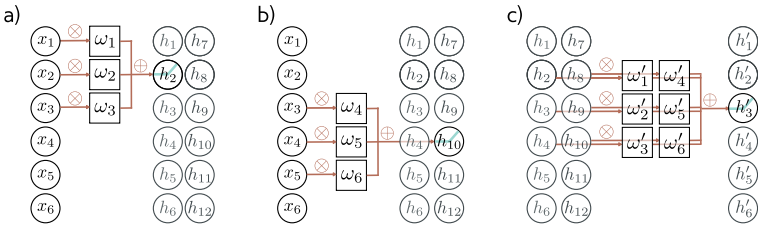

Figure 32: Channels (Prince 2024, 166)

- Figure 32, *panel a)*: a first kernel computes a weighted sum of the three nearest inputs, adds a bias, and passes the results through the activation function to produce hidden units \(h_{1}\) to \(h_{6}\) (first channel).
- Figure 32, *panel b)*: a second kernel computes a different weighted sum, adds a different bias, and passes the results through the activation function to produce hidden units \(h_{7}\) to \(h_{12}\) (second channel).
- Figure 32, *panel c)*: if a layer has \(C_{i}\) channels, the hidden units in the next layer are computed as a weighted sum over these channels.

Convolutional Networks

Receptive Fields

- The receptive field of a hidden unit in the network is the region of the original input that feeds into it.
- Convolutional networks comprise a sequence of convolutional layers.

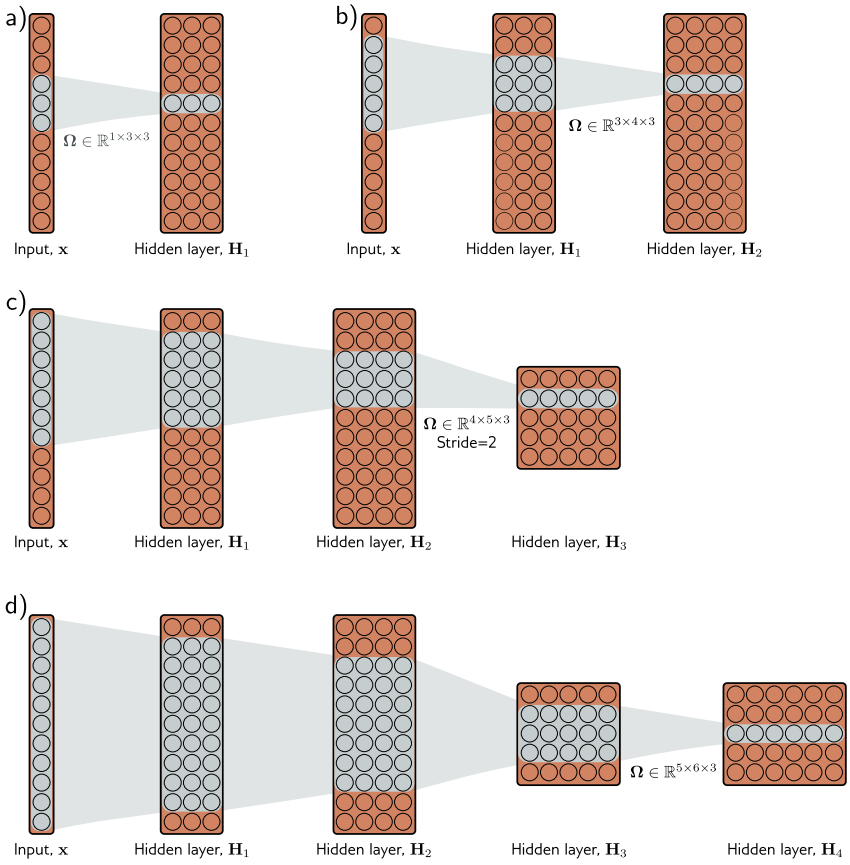

Figure 33: Receptive fields (Prince 2024, 168)

- Figure 33, *panel a)*: The hidden units in $\mathbf{H}_1$ are weighted sums of the three nearest inputs. The receptive field in $\mathbf{H}_1$ is of size 3.
- Figure 33, *panel b)*: The hidden units in $\mathbf{H}_2$ are weighted sums of the three nearest positions in $\mathbf{H}_1$, which are themselves weighted sums of the three nearest inputs. The receptive field in $\mathbf{H}_2$ is of size 5.
- Figure 33, *panel c-d)*: By the time we add $\mathbf{H}_4$, the receptive field covers the entire input.

  → Therefore, the receptive field of units in successive layers increases, integrating information from more inputs.

Convolutional Networks

Convolutional Networks for 2D Inputs

- We so far discussed convolutional networks for processing 1D data.
- However, convolutional networks are typically applied to 2D image data.
- With 2D inputs, a convolutional layer with a 3 \(\times\) 3 kernel \(\boldsymbol{\Omega} \in \mathbb{R}^{3 \times 3}\) computes a hidden unit \(h_{ij}\) in a hidden layer as: \[h_{ij} = a \Bigg[\beta + \sum_{m = 1}^{3}\sum_{n = 1}^{3}\omega_{mn}x_{i + m - 2, j + n - 2}\Bigg], \] where \(\omega_{mn}\) are the entries of the convolutional kernel and \(x_{ij}\) are elements of the 2D input matrix.
  - → The convolution operation therefore computes a weighted sum over a square 3 \(\times\) 3 input region (Figure 34).

Convolutional Networks

Convolutional Networks for 2D Inputs (Cont.)

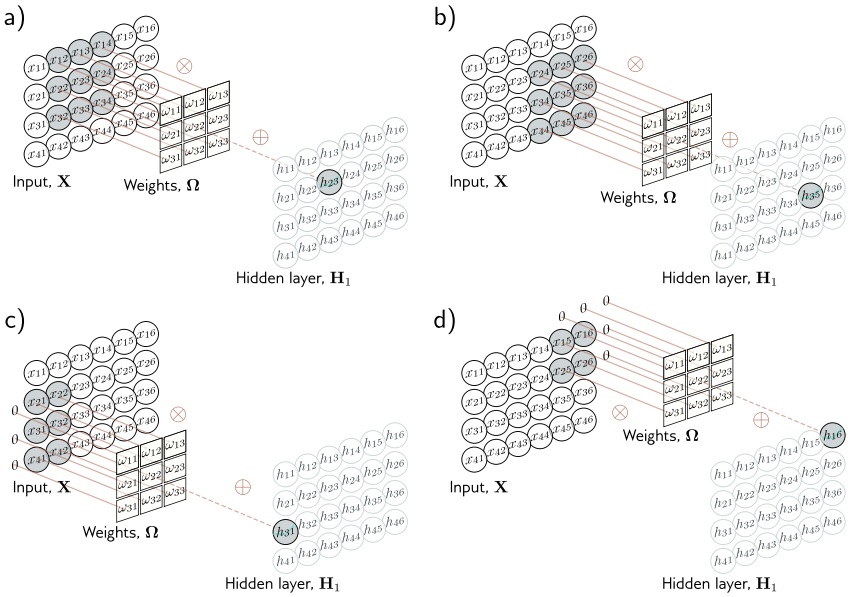

Figure 34: 2D convolutional layer. *Panels a)-b):* each output (shaded output) is a weighted sum of the \(3 \times 3\) nearest inputs (shaded inputs). *Panels c)-d):* with zero padding, positions beyond the image's edge are considered to be 0 (Prince 2024, 171)

Convolutional Networks

Downsampling and Upsampling

- In the case of convolutional networks for 1D inputs, we can use stride and dilation to scale down (downsample) the representation and increase the receptive field at each layer.
- In the case of convolutional networks for 2D inputs, there are three main methods for downsampling (e.g., scaling down both dimensions of a 4 \(\times\) 4 representation by a factor of 2):
  - → subsampling (retain every other position),
  - → max pooling (retain the maximum value of the corresponding 2 \(\times\) 2 block),
  - → mean pooling (retain the mean of the values in the 2 \(\times\) 2 block).

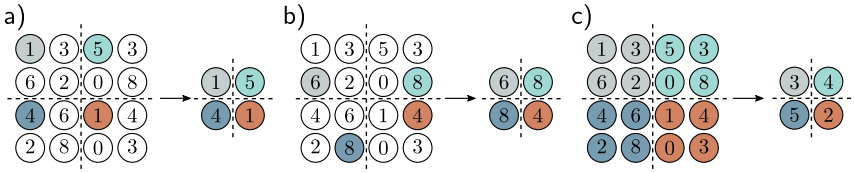

Figure 35: Methods for scaling down representation size (downsampling). Subsampling (panel a), max pooling (panel b), and mean pooling (panel c) (Prince 2024, 173)

Convolutional Networks

Downsampling and Upsampling (Cont.)

- We sometimes want to scale representations back up (upsampling), such as in cases where the output is also an image.
- Three common methods for upsampling are (e.g., for scaling up both dimensions of a 2 \(\times\) 2 representation by a factor of 2):
  - → duplication (duplicate each input 4 times),
  - → max unpooling (if we previously used a max pooling operation, we distribute the values back to the positions they originally came from, i.e., where the maxima were),
  - → bilinear interpolation (fill in the missing values between the points where we have samples).

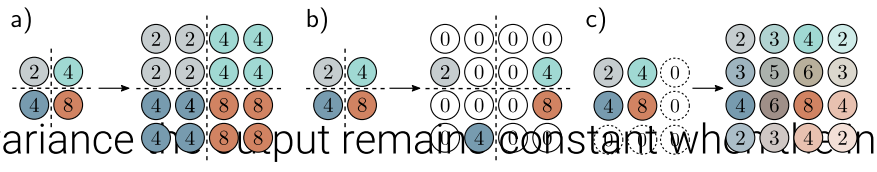

Figure 26: Methods for scaling up representation size (upsampling): Duplication (panel a), max unpooling (panel b) and bilinear interpolation (panel c) (Prince 2024, 173)

1. Under invariance the output remains constant when the input is transformed, while under equivariance it is transformed in a specific way. Hence, invariance is a special case of equivariance in which the output transformation is the identity function (Bishop and Bishop 2024, 259).

1. In computer vision, "translation" is a type of transformation and means displacement in space. Other types of transformations are, e.g., reflection, rotation, and scaling.

Python/PyTorch

Demo 4: Convolutional Neural Networks (CNNs)

Notebook 3 in Google Colab

Transformers

Transformers

- We previously discussed convolutional networks, which are well suited to processing images.
  - → Images have a large number of inputs and nearby inputs are related.
- Transformers were initially developed for natural language processing (NLP) problems.
- Text data have some similarities with image data.
  - → The number of inputs can be very large and inputs are related.
- However, text sequences can vary in length (and, unlike images, there is no easy way to resize them).
- For NLP problems, we therefore need a network architecture other than fully-connected networks and convolutional networks.

Transformers

Motivation

Design a network to process this text into a representation for some downstream task (Prince 2024, 207):

The restaurant refused to serve me a ham sandwich because it only cooks vegetarian food. In the end, they just gave me two slices of bread. Their ambiance was just as good as the food and service.

Three problems:

- The encoded input can be very large: representing each of the above 37 words by an **embedding vector** of length 1,024 leads to an input of length 37 \(\times\) 1,024 = 37,888.
- Each input can be of a different length: unclear how we would apply a fully-connected network.
- Language is ambiguous: to understand the text, the words *it*, *they*, and *their* should be connected to the word *restaurant*. This means that the former words should pay attention to the latter word.

Solution: A network for processing text that will

- use parameter sharing to cope with long inputs of differing lengths;
- contain connections between word representations.

  → A transformer acquires both properties by using dot-product self-attention.

Transformers

Dot-Product Self-Attention

- A standard neural network layer $\mathbf{f}[\mathbf{x}]$ takes a $D \times 1$ input $\mathbf{x}$, applies a linear transformation, and passes the result through an activation function, e.g., ReLU: $$\mathbf{f}[\mathbf{x}] = \mathbf{ReLU}[\boldsymbol{\beta} + \boldsymbol{\Omega}\mathbf{x}],$$ where $\boldsymbol{\beta}$ contains the biases and $\boldsymbol{\Omega}$ contains the weights.
- A self-attention block $\mathbf{sa}[\cdot]$ takes N input vectors $\mathbf{x}_{1}, \dotsc, \mathbf{x}_{N}$ of dimension $D \times 1$ each (e.g., N words) and returns N output vectors of the same size, $D \times 1$.
  - → First, it computes a set of values for each input $\mathbf{x}_{m}$: $$\mathbf{v}_{m} = \boldsymbol{\beta}_{v} + \boldsymbol{\Omega}_{v}\mathbf{x}_{m}. \qquad (5)$$
  - → Second, the nth output vector $\mathbf{sa}_{n}[\mathbf{x}_{1}, \dotsc, \mathbf{x}_{N}]$ is a weighted sum of all the value vectors $\mathbf{v}_{1}, \dotsc, \mathbf{v}_{N}$: $$\mathbf{sa}_{n}[\mathbf{x}_{1}, \dotsc, \mathbf{x}_{N}] = \sum_{m = 1}^{N}a[\mathbf{x}_{m}, \mathbf{x}_{n}]\mathbf{v}_{m}. \qquad (6)$$
  - → The scalar weight $a[\mathbf{x}_{m}, \mathbf{x}_{n}]$ is the attention that the nth output pays to input $\mathbf{x}_{m}$.

Transformers

Dot-Product Self-Attention (Cont.)

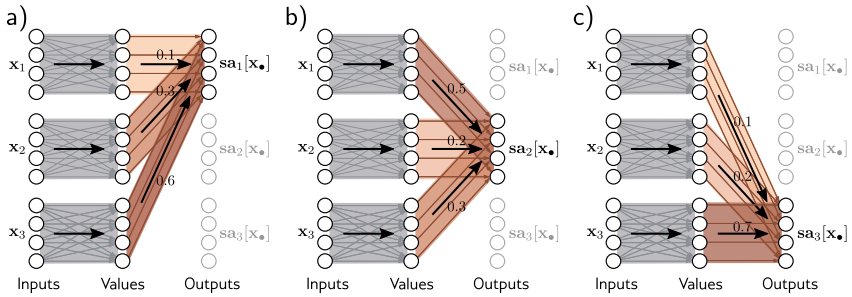

Figure 37: Self-attention as routing. *Step 1.* The self-attention mechanism takes \(N\) inputs \(\mathbf{x}_{1}, \dotsc, \mathbf{x}_{N} \in \mathbb{R}^{D}\) (here \(N = 3\) and \(D = 4\)) and processes each separately to compute \(N\) value vectors. *Step 2.* The \(n\)th output vector \(\mathbf{sa}_{n}[\mathbf{x}_{1}, \dotsc, \mathbf{x}_{N}]\) is then computed as a weighted sum of the \(N\) value vectors. E.g., *panel a):* Output vector \(\mathbf{sa}_{1}[\cdot]\) is computed as \(a[\mathbf{x}_{1}, \mathbf{x}_{1}] = 0.1\) times the first value vector, \(a[\mathbf{x}_{2}, \mathbf{x}_{1}] = 0.3\) times the second value vector, and \(a[\mathbf{x}_{3}, \mathbf{x}_{1}] = 0.6\) times the third value vector (Prince 2024, 209)

Transformers

Computing Attention Weights

- In **Equation 5**, we saw that the value vectors $\boldsymbol{\beta}_{v} + \boldsymbol{\Omega}_{v}\mathbf{x}_{m}$ are computed linearly for each input $\mathbf{x}_{m}$.
- As we saw in **Equation 6**, these value vectors are then combined linearly by the attention weights $a[\mathbf{x}_{m}, \mathbf{x}_{n}]$.
- However, the overall self-attention computation is nonlinear because the attention weights $a[\mathbf{x}_{m}, \mathbf{x}_{n}]$ are nonlinear functions of the inputs.
- To compute the attention weights, we calculate two more linear functions of the inputs: $$\begin{split} \mathbf{q}_{n} &= \boldsymbol{\beta}_{q} + \boldsymbol{\Omega}_{q}\mathbf{x}_{n} \\ \mathbf{k}_{m} &= \boldsymbol{\beta}_{k} + \boldsymbol{\Omega}_{k}\mathbf{x}_{m}, \end{split}$$ where the $\{\mathbf{q}_{n}\}$ are called queries and the $\{\mathbf{k}_{m}\}$ are called keys.
- Next, we compute dot products (i.e., similarities) between the queries and the keys and pass the results through a softmax function to obtain the attention weight: $$a[\mathbf{x}_{m}, \mathbf{x}_{n}] = \frac{\exp\big[\mathbf{k}_{m}^{T}\mathbf{q}_{n}\big]}{\sum_{m^{\prime} = 1}^{N}\exp\big[\mathbf{k}_{m^{\prime}}^{T}\mathbf{q}_{n}\big]}.$$ For each $\mathbf{x}_{n}$, these weights are nonnegative and sum to 1.

Transformers

Computing Attention Weights (Cont.)

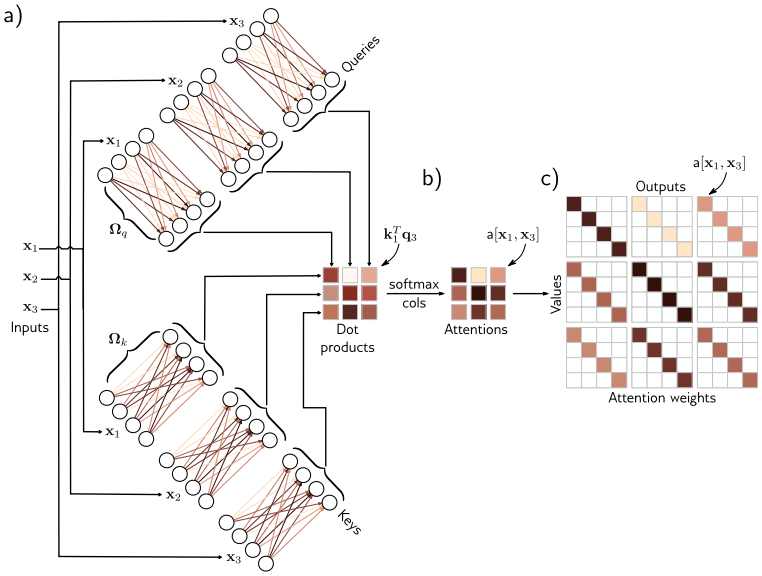

Figure 38: Computing attention weights. *a)* Query vectors $\mathbf{q}_{n} = \boldsymbol{\beta}_{q} + \boldsymbol{\Omega}_{q}\mathbf{x}_{n}$ and key vectors $\mathbf{k}_{m} = \boldsymbol{\beta}_{k} + \boldsymbol{\Omega}_{k}\mathbf{x}_{m}$ are computed for each input $\mathbf{x}_{n}$. *b)* The dot products between each query and key are passed through a softmax function (to form non-negative attentions that sum to one). *c)* These attentions then route the value vectors, see Figure 37 (Prince 2024, 211)

Transformers

Back to the Motivation

- We discussed three problems characterizing the task of designing a neural network to encode and process text:
  - → the encoded input can be very large;
  - → each input can be of a different length;
  - → to understand a text, inputs (words) should be connected.
- We also discussed that to overcome these problems, a network should have the following properties:
  - → it should use parameter sharing to cope with long inputs of differing lengths;
  - → it should contain connections between word representations.
- The attention mechanism has these properties:
  - → it uses a single shared set of parameters, $\boldsymbol{\phi} = \{\boldsymbol{\beta}_{v}, \boldsymbol{\Omega}_{v}, \boldsymbol{\beta}_{q}, \boldsymbol{\Omega}_{q}, \boldsymbol{\beta}_{k}, \boldsymbol{\Omega}_{k}\}$, which is independent of the number of inputs N (so the network can be applied to different sequence lengths).
  - → it contains connections between the inputs (words), and the strength of these connections depends on the inputs themselves (via the attention weights).

Transformers

Transformer Layer

- Self attention is just one part of a transformer layer.
- In a transformer layer, self attention is followed by other blocks, such as fully-connected layers.

Figure 39: Transformer. The input consists of a \(D \times N\) matrix containing the \(D\)-dimensional word embeddings for each of the \(N\) words. The output is a matrix of the same size. The transformer layer consists of a series of operations: first, there is a (multi-head) attention block; second, this block is followed by other blocks, such as a block of fully-connected layers

For a more visual explanation of how transformers work, see Grant Sanderson's `3Blue1Brown` videos on Transformers and Attention in transformers.

Python/PyTorch

Demo 5: Transformers

Notebook 4 in Google Colab

References

Bisbee, James, Joshua D. Clinton, Cassy Dorff, Brenton Kenkel, and Jennifer M. Larson. 2024. “Synthetic Replacements for Human Survey Data? The Perils of Large Language Models.” *Political Analysis* 32 (4): 401–16. https://doi.org/10.1017/pan.2024.5.

Bishop, Christopher M., and Hugh Bishop. 2024. *Deep Learning: Foundations and Concepts*. Springer. https://www.bishopbook.com/.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani. 2021. *An Introduction to Statistical Learning: With Applications in R*. 2nd ed. New York: Springer. https://www.statlearning.com.

Prince, Simon J. D. 2024. *Understanding Deep Learning*. MIT Press. http://udlbook.com.