Introduction to Machine Learning in Data Assimilation

Eviatar Bach

10 June 2025

Machine learning (ML) is increasingly being used in combination with data assimilation.

Machine learning (ML) is increasingly being used in combination with data assimilation.

In this lecture, we will discuss:

- Types of ML algorithms, including some important architectures.

Machine learning (ML) is increasingly being used in combination with data assimilation.

In this lecture, we will discuss:

- Types of ML algorithms, including some important architectures.
- How ML can be incorporated into DA (more on this on Friday).

Machine learning (ML) is increasingly being used in combination with data assimilation.

In this lecture, we will discuss:

- Types of ML algorithms, including some important architectures.
- How ML can be incorporated into DA (more on this on Friday).
- Some important topics in ML:

Machine learning (ML) is increasingly being used in combination with data assimilation.

In this lecture, we will discuss:

- Types of ML algorithms, including some important architectures.
- How ML can be incorporated into DA (more on this on Friday).
- Some important topics in ML:
    - Introduction to optimisation

Machine learning (ML) is increasingly being used in combination with data assimilation.

In this lecture, we will discuss:

- Types of ML algorithms, including some important architectures.
- How ML can be incorporated into DA (more on this on Friday).
- Some important topics in ML:
    - Introduction to optimisation
    - Automatic differentiation

Machine learning (ML) is increasingly being used in combination with data assimilation.

In this lecture, we will discuss:

- Types of ML algorithms, including some important architectures.
- How ML can be incorporated into DA (more on this on Friday).
- Some important topics in ML:
    - Introduction to optimisation
    - Automatic differentiation
    - Train and test error, cross-validation

ML in DA

How to incorporate machine learning (ML) into DA?

- Unknown or partially unknown system dynamics: Learning forecast model

ML in DA

How to incorporate machine learning (ML) into DA?

- Unknown or partially unknown system dynamics: Learning forecast model
- Using a learned forecast model in DA

ML in DA

How to incorporate machine learning (ML) into DA?

- Unknown or partially unknown system dynamics: Learning forecast model
- Using a learned forecast model in DA
- Learning the assimilation step, or the forecast + assimilation step jointly.

Inverse Problems and Data Assimilation: A Machine Learning Approach

Eviatar Bach [♯♮], Ricardo Baptista [♯], Daniel Sanz-Alonso [♭], Andrew Stuart [♯]

https://www.arxiv.org/abs/2410.10523

Types of machine learning

Machine learning algorithms can broadly be classified into

- **Supervised learning**: data is given "labelled", objective is to learn relationship to predict on new data.

Machine learning algorithms can broadly be classified into

- **Supervised learning**: data is given "labelled", objective is to learn relationship to predict on new data.
- **Unsupervised learning**: data is unlabelled, and objective is to learn some patterns in the data.

Supervised learning

Supervised learning

Assume we have a dataset of N pairs of points

$$\{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N},$$

that is, $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)$.

Supervised learning

Assume we have a dataset of N pairs of points

$$\{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N},$$

that is, $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)$.

Labels: the $\mathbf{x}_n$ are inputs, $\mathbf{y}_n$ are outputs.

Supervised learning

Assume we have a dataset of N pairs of points

$$\{(\mathbf{x}_n, \mathbf{y}_n)\}_{n=1}^{N},$$

that is, $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)$.

Labels: the $\mathbf{x}_n$ are inputs, $\mathbf{y}_n$ are outputs.

We assume that the $\mathbf{y}_n$ are related to the $\mathbf{x}_n$ through some possibly noisy function, i.e.,

$$\mathbf{y}_n = f(\mathbf{x}_n; \theta) + \text{noise}.$$

The objective is to learn the parameters θ for some specific form of $\mathbf{f}$.

Regression

In *regression*, the outputs $\mathbf{y}_n$ take continuous values.

Regression

In *regression*, the outputs $\mathbf{y}_n$ take continuous values.

A common cost function is the mean-squared loss:

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \|\mathbf{f}(\mathbf{x}_n; \boldsymbol{\theta}) - \mathbf{y}_n\|^2.$$

Regression

In *regression*, the outputs $\mathbf{y}_n$ take continuous values.

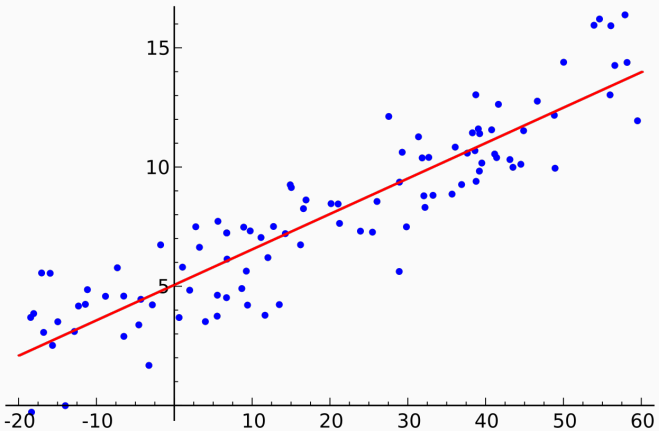
A common cost function is the mean-squared loss:

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} \|\mathbf{f}(\mathbf{x}_n; \boldsymbol{\theta}) - \mathbf{y}_n\|^2.$$

Example: in linear regression,

$$\mathbf{y}_n = \boldsymbol{\beta}^\top \mathbf{x}_n + \text{noise}.$$

In this case, $\boldsymbol{\theta} = \boldsymbol{\beta}$.



In some cases, the minimizer of $J(\boldsymbol{\theta})$ can be found analytically using calculus (or its extension, matrix calculus).

In some cases, the minimizer of $J(\boldsymbol{\theta})$ can be found analytically using calculus (or its extension, matrix calculus).

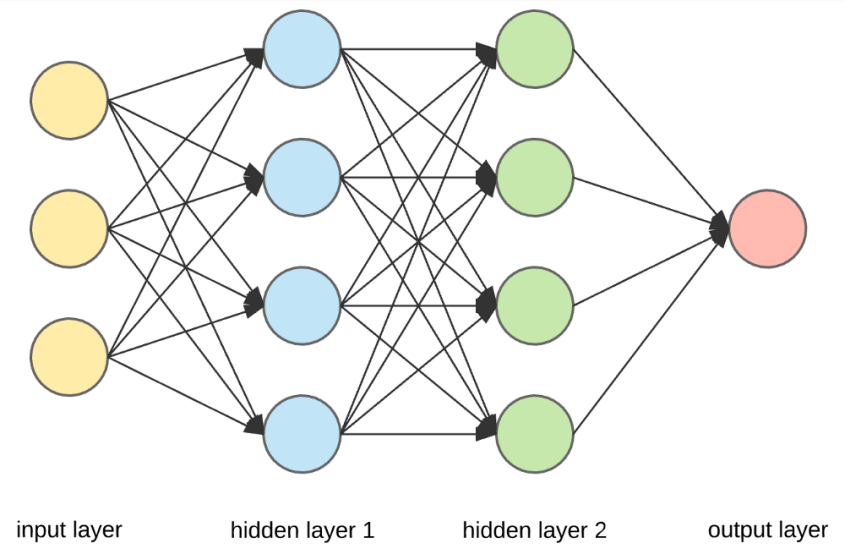
Otherwise, we will need to minimise approximately.

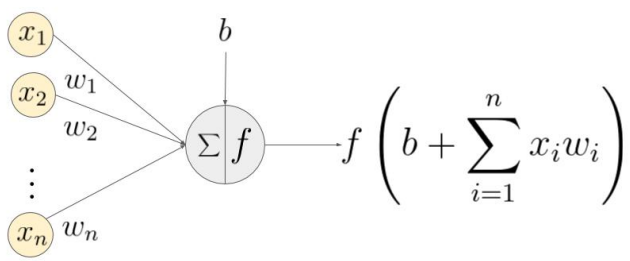
In some cases, the minimizer of $J(\boldsymbol{\theta})$ can be found analytically using calculus (or its extension, matrix calculus).

Otherwise, we will need to minimise approximately.

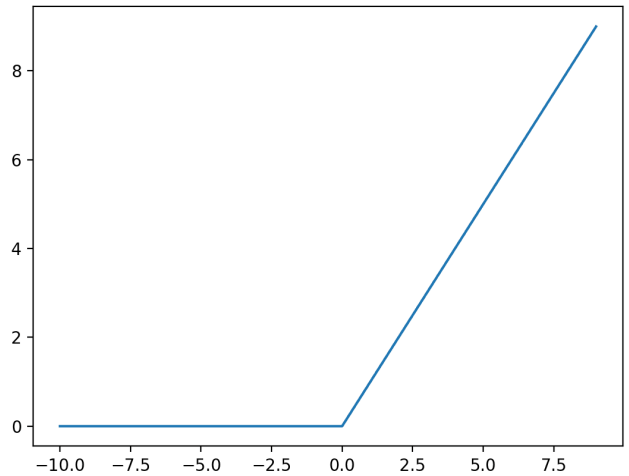
Example: neural networks.

Neural networks





An example of a neuron showing the input (x_1 - x_n), their corresponding weights (w_1 - w_n), a bias (b) and the activation function f applied to the weighted sum of the inputs.



Neural networks

Deep learning: many hidden layers, often overparameterised.

Neural networks

Deep learning: many hidden layers, often overparameterised.

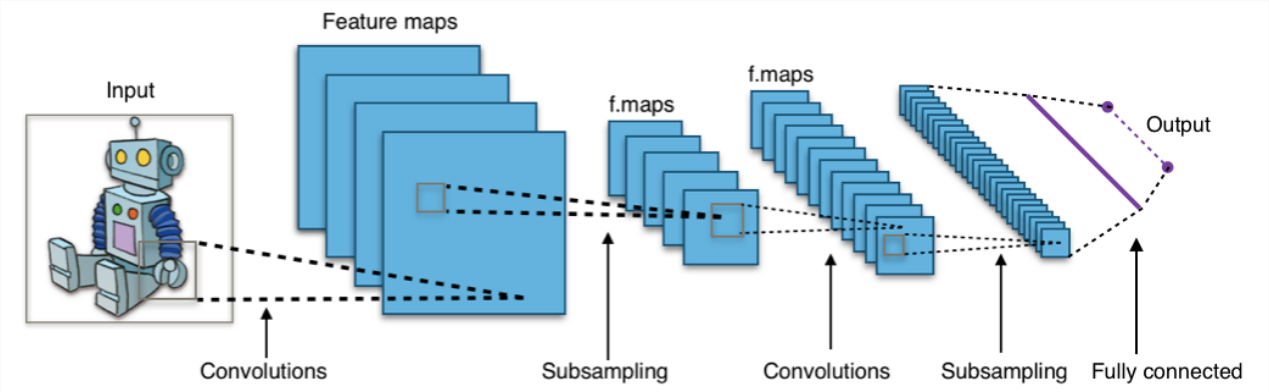
Nonlinearity of activation function allows for universal approximation properties.

Neural networks

Deep learning: many hidden layers, often overparameterised.

Nonlinearity of activation function allows for universal approximation properties.

Can build in spatial structure: convolutional neural networks.



Time-series forecasting

A special case of regression. Assume data is a *time-series*: that is, given by points ordered in time:

$$\{\mathbf{x}(t)\}_{t=1}^{T}.$$

Time-series forecasting

A special case of regression. Assume data is a *time-series*: that is, given by points ordered in time:

$$\{\mathbf{x}(t)\}_{t=1}^{T}.$$

A *Markovian* time-series forecasting model takes

$$\mathbf{x}(t+1) = \mathbf{f}(\mathbf{x}(t); \boldsymbol{\theta}) + \text{noise},$$

or memory can be incorporated as

$$\mathbf{x}(t+1) = \mathbf{f}(\{\mathbf{x}(s)\}_{s=t-\tau}^{t}; \boldsymbol{\theta}) + \text{noise},$$

where τ is the amount of memory.

Time-series forecasting

A special case of regression. Assume data is a *time-series*: that is, given by points ordered in time:

$$\{\mathbf{x}(t)\}_{t=1}^{T}.$$

A *Markovian* time-series forecasting model takes

$$\mathbf{x}(t+1) = \mathbf{f}(\mathbf{x}(t); \boldsymbol{\theta}) + \text{noise},$$

or memory can be incorporated as

$$\mathbf{x}(t+1) = \mathbf{f}(\{\mathbf{x}(s)\}_{s=t-\tau}^{t}; \boldsymbol{\theta}) + \text{noise},$$

where τ is the amount of memory. If $\mathbf{x}$ includes all the relevant variables, memory will in general not be needed. If there are unobserved variables it can be useful.

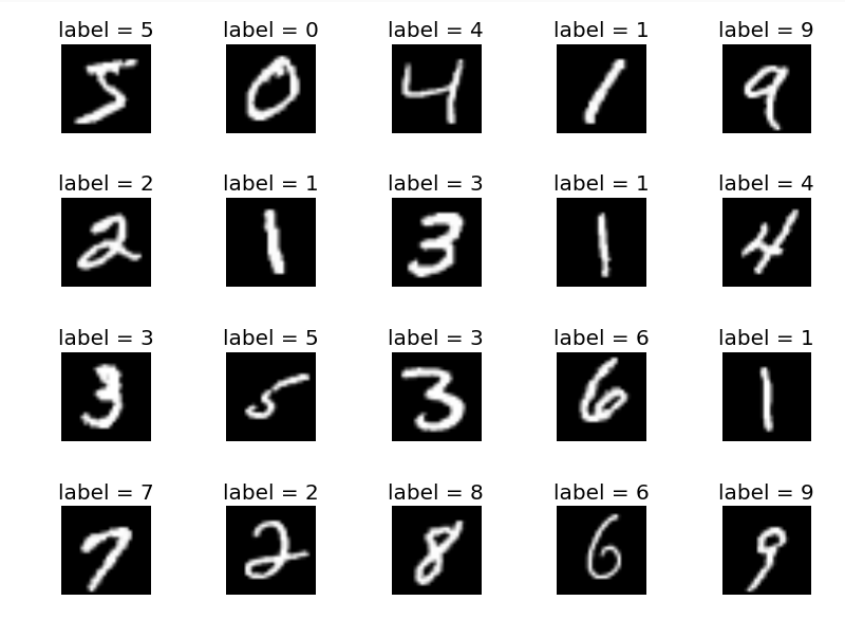
Classification

A type of supervised learning where the outputs $\mathbf{y}_n$ take on discrete values.

Classification

A type of supervised learning where the outputs $\mathbf{y}_n$ take on discrete values.

Example: MNIST



Unsupervised learning

Generative models

Suppose we have a dataset

$$\{\mathbf{x}_n\}_{n=1}^N,$$

where $\mathbf{x}_n \sim p(\mathbf{x}; \boldsymbol{\theta})$. That is, the data points are assumed to have come from a probability distribution $p(\mathbf{x}; \boldsymbol{\theta})$.

Generative models

Suppose we have a dataset

$$\{\mathbf{x}_n\}_{n=1}^N,$$

where $\mathbf{x}_n \sim p(\mathbf{x}; \boldsymbol{\theta})$. That is, the data points are assumed to have come from a probability distribution $p(\mathbf{x}; \boldsymbol{\theta})$.

For generative models, we first estimate $\boldsymbol{\theta}$ in some class, and then produce new samples from this distribution.

Generative models

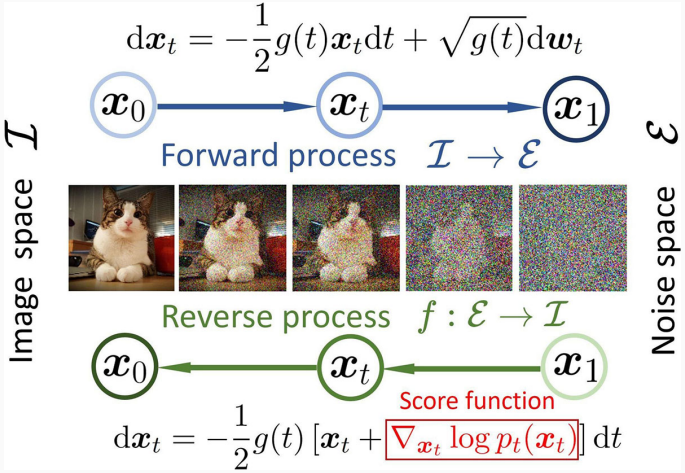
Suppose we have a dataset

$$\{\mathbf{x}_n\}_{n=1}^N,$$

where $\mathbf{x}_n \sim p(\mathbf{x}; \boldsymbol{\theta})$. That is, the data points are assumed to have come from a probability distribution $p(\mathbf{x}; \boldsymbol{\theta})$.

For generative models, we first estimate $\boldsymbol{\theta}$ in some class, and then produce new samples from this distribution.

Example: diffusion models

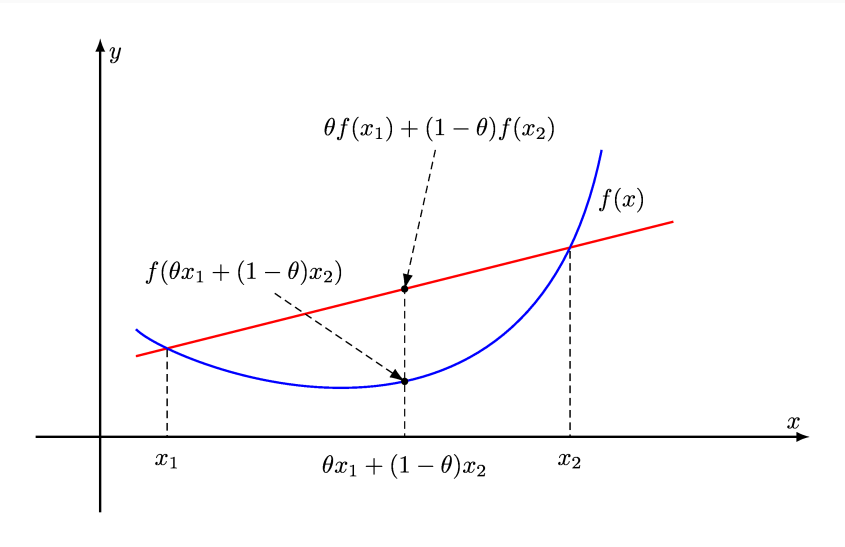


Optimisation

Convexity

A function $f : \mathbb{R}^d \to \mathbb{R}$ is called *convex* if, for all s_1, s_2 and for all $\theta \in [0, 1]$, we have that

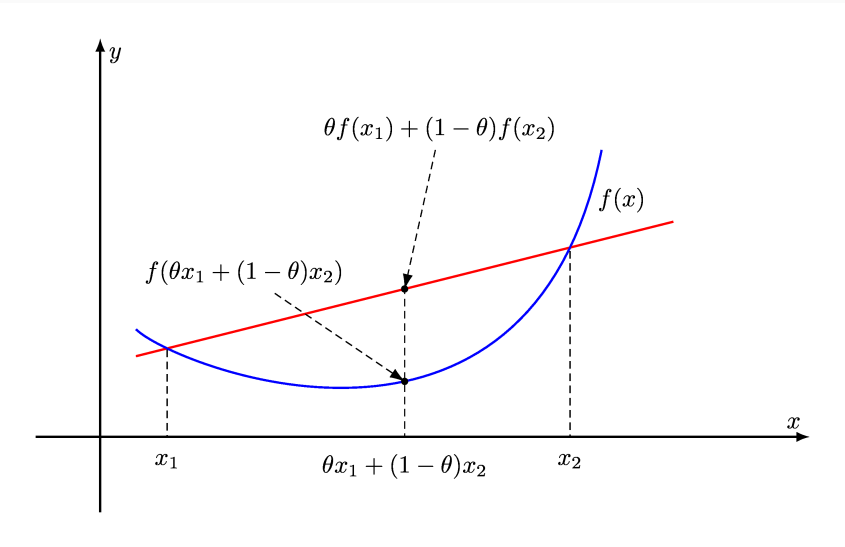
$$f(\theta s_1 + (1 - \theta)s_2) \leq \theta f(s_1) + (1 - \theta)f(s_2).$$



Convexity

A function $f : \mathbb{R}^d \to \mathbb{R}$ is called *convex* if, for all s_1, s_2 and for all $\theta \in [0, 1]$, we have that

$$f(\theta s_1 + (1 - \theta)s_2) \leq \theta f(s_1) + (1 - \theta)f(s_2).$$



A function is called *strictly convex* if the inequality is strict for $s_1 \neq s_2$.

Convexity

Strictly convex functions have a unique global minimum x^*.

Convexity

Strictly convex functions have a unique global minimum x^*.

Positive definite Hessian (positive second derivative) $\implies$ strictly convex, and positive semidefinite Hessian (nonnegative second derivative) $\iff$ convex.

Gradient descent

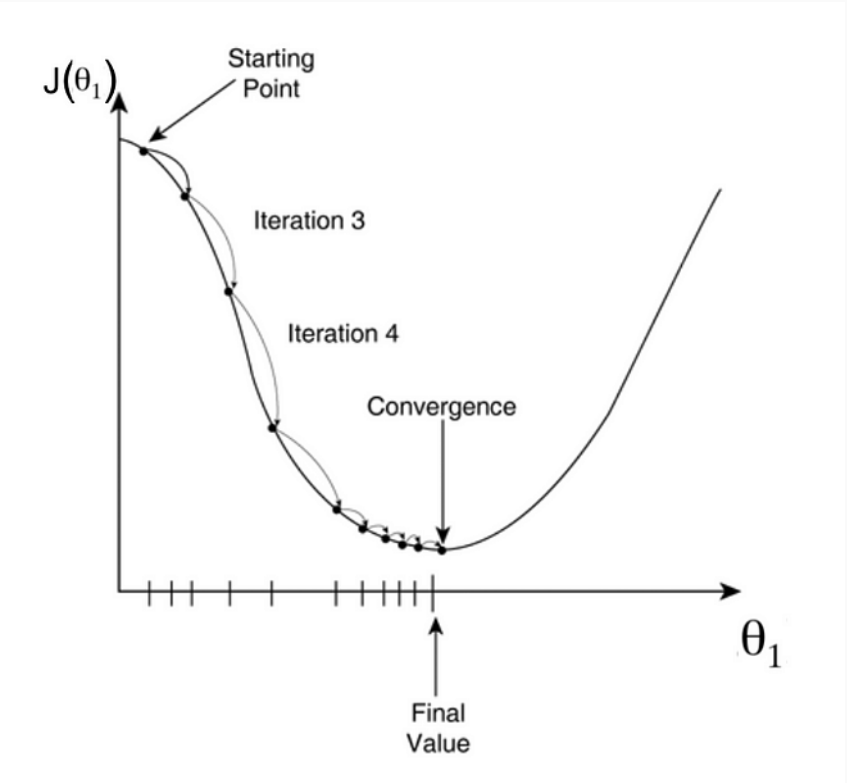
Given $f : \mathbb{R}^d \to \mathbb{R}$, gradient descent is computed from a sequence of step-sizes $\{\alpha_j\}_{j \in \mathbb{Z}^+}$ by picking an $x_0 \in \mathbb{R}^d$ and then iterating as follows:

$$x_{j+1} = x_j - \alpha_j \nabla f(x_j), \qquad j \in \mathbb{Z}^+.$$

Gradient descent

Given $f : \mathbb{R}^d \to \mathbb{R}$, gradient descent is computed from a sequence of step-sizes $\{\alpha_j\}_{j \in \mathbb{Z}^+}$ by picking an $x_0 \in \mathbb{R}^d$ and then iterating as follows:

$$x_{j+1} = x_j - \alpha_j \nabla f(x_j), \qquad j \in \mathbb{Z}^+.$$



Convexity

For strongly convex functions, gradient descent converges to global minimum.

Convexity

For strongly convex functions, gradient descent converges to global minimum.

Typically, cost functions encountered in ML may be highly nonconvex.

Convexity

For strongly convex functions, gradient descent converges to global minimum.

Typically, cost functions encountered in ML may be highly nonconvex.

We generally will not find the global minimum, and may only find a local one.

Convexity

For strongly convex functions, gradient descent converges to global minimum.

Typically, cost functions encountered in ML may be highly nonconvex.

We generally will not find the global minimum, and may only find a local one.

Optimisation algorithms for nonconvex functions try to balance *exploration* and *exploitation*:

- Exploitation: pursue the most promising current avenue (e.g., gradient descent)
- Exploration: venture out to find new avenues in the hope of finding a more promising avenue (e.g., stochasticity)

Stochastic gradient descent (SGD)

Suppose the cost function can be written as a mean over data points, e.g., recall

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} (f(x_n; \boldsymbol{\theta}) - y_n)^2.$$

Stochastic gradient descent (SGD)

Suppose the cost function can be written as a mean over data points, e.g., recall

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} (f(x_n; \boldsymbol{\theta}) - y_n)^2.$$

Computing the gradient,

$$\nabla J(\boldsymbol{\theta}) = \frac{2}{N} \sum_{n=1}^{N} \nabla f(x_n)(f(x_n; \boldsymbol{\theta}) - y_n).$$

Stochastic gradient descent (SGD)

Suppose the cost function can be written as a mean over data points, e.g., recall

$$J(\boldsymbol{\theta}) = \frac{1}{N} \sum_{n=1}^{N} (f(x_n; \boldsymbol{\theta}) - y_n)^2.$$

Computing the gradient,

$$\nabla J(\boldsymbol{\theta}) = \frac{2}{N} \sum_{n=1}^{N} \nabla f(x_n)(f(x_n; \boldsymbol{\theta}) - y_n).$$

It may be expensive to compute this gradient when N is large, such as when f is a large neural network.

Stochastic gradient descent (SGD)

Suppose the cost function can be written as a mean over data points, e.g., recall

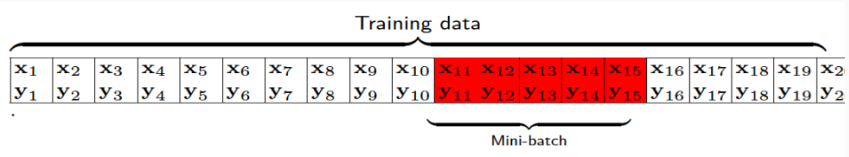
$$J(\boldsymbol{\theta}) = \frac{1}{N}\sum_{n=1}^{N}(f(x_n;\boldsymbol{\theta}) - y_n)^2.$$

Computing the gradient,

$$\nabla J(\boldsymbol{\theta}) = \frac{2}{N}\sum_{n=1}^{N}\nabla f(x_n)(f(x_n;\boldsymbol{\theta}) - y_n).$$

It may be expensive to compute this gradient when N is large, such as when f is a large neural network.

Stochastic gradient descent (SGD) instead computes the gradient with respect to a random subset: a *mini-batch*.



Stochastic gradient descent (SGD)

One full sweep through the data (consisting of multiple SGD steps) is called an *epoch*.

Stochastic gradient descent (SGD)

One full sweep through the data (consisting of multiple SGD steps) is called an *epoch*.

This reduces computational cost but also encourages exploration.

Stochastic gradient descent (SGD)

One full sweep through the data (consisting of multiple SGD steps) is called an *epoch*.

This reduces computational cost but also encourages exploration.

Popular optimisation methods used in ML (e.g., ADAM) are variants of stochastic gradient descent.

Automatic differentiation

Gradients are often difficult to obtain in closed form for complex cost functions.

Automatic differentiation

Gradients are often difficult to obtain in closed form for complex cost functions.

Finite difference approximations are often inaccurate and expensive for high-dimensional problems.

Automatic differentiation

Gradients are often difficult to obtain in closed form for complex cost functions.

Finite difference approximations are often inaccurate and expensive for high-dimensional problems.

Automatic differentiation (autodiff) involves repeated application of chain rule on elementary operations that enables computing derivatives accurately to working precision.

Automatic differentiation

Gradients are often difficult to obtain in closed form for complex cost functions.

Finite difference approximations are often inaccurate and expensive for high-dimensional problems.

Automatic differentiation (autodiff) involves repeated application of chain rule on elementary operations that enables computing derivatives accurately to working precision.

This can allow differentiation through model states (adjoint), model parameters, and DA algorithms, and enable use of gradient-based optimization.

Automatic differentiation

Suppose $y = f(x)$, where $y \in \mathbb{R}^{d_n}$ and $x \in \mathbb{R}^{d_0}$, and the derivative of f is not readily available in closed-form.

Automatic differentiation

Suppose $y = f(x)$, where $y \in \mathbb{R}^{d_n}$ and $x \in \mathbb{R}^{d_0}$, and the derivative of f is not readily available in closed-form.

We assume the implementation of f in computer code is made up of n elementary operations $f_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$ (for instance, addition, multiplication, logarithms, etc.), such that

$$f(x) = f_n \circ f_{n-1} \circ \cdots \circ f_1(x), \tag{1}$$

where the Jacobians for these elementary operations, $Df_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i \times d_{i-1}}$, are available in closed form.

Automatic differentiation

Suppose $y = f(x)$, where $y \in \mathbb{R}^{d_n}$ and $x \in \mathbb{R}^{d_0}$, and the derivative of f is not readily available in closed-form.

We assume the implementation of f in computer code is made up of n elementary operations $f_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i}$ (for instance, addition, multiplication, logarithms, etc.), such that

$$f(x) = f_n \circ f_{n-1} \circ \cdots \circ f_1(x), \tag{1}$$

where the Jacobians for these elementary operations, $Df_i : \mathbb{R}^{d_{i-1}} \to \mathbb{R}^{d_i \times d_{i-1}}$, are available in closed form.

Writing the partial evaluations up to $i \leq n$ as

$$g_i = (f_i \circ \cdots \circ f_1)(x),$$

by the chain rule we have that

$$D_x f(x) = (D_{g_{n-1}} f_n(g_{n-1})) \cdots (D_{g_1} f_2(g_1))(D_x f_1(x)). \tag{2}$$

Automatic differentiation

This suggests the following algorithm (*forward mode* automatic differentiation):

1: **Input**: The functions $\{f_i(\cdot)\}_{i=1}^n$, their corresponding Jacobians $\{Df_i(\cdot)\}_{i=1}^n$, and the function input x.
2: Set $g_1 = f_1(x)$ and $J_1 = Df_1(x)$.
3: For $i = 2, \ldots, n$: set $g_i = f_i(g_{i-1})$ and $J_i = (Df_i(g_{i-1}))J_{i-1}$.
4: **Output**: The function output $y = f(x) = g_n$ and the derivative $Df(x) = J_n$.

Automatic differentiation

This suggests the following algorithm (*forward mode* automatic differentiation):

1: **Input**: The functions $\{f_i(\cdot)\}_{i=1}^n$, their corresponding Jacobians $\{Df_i(\cdot)\}_{i=1}^n$, and the function input x.
2: Set $g_1 = f_1(x)$ and $J_1 = Df_1(x)$.
3: For $i = 2, \ldots, n$: set $g_i = f_i(g_{i-1})$ and $J_i = (Df_i(g_{i-1}))J_{i-1}$.
4: **Output**: The function output $y = f(x) = g_n$ and the derivative $Df(x) = J_n$.

Reverse mode autodiff requires a backwards pass to compute the derivative, and is more efficient when $d_0 \gg d_n$ (many inputs), whereas forward mode is more efficient when $d_n \gg d_0$ (many outputs).

Generalisation

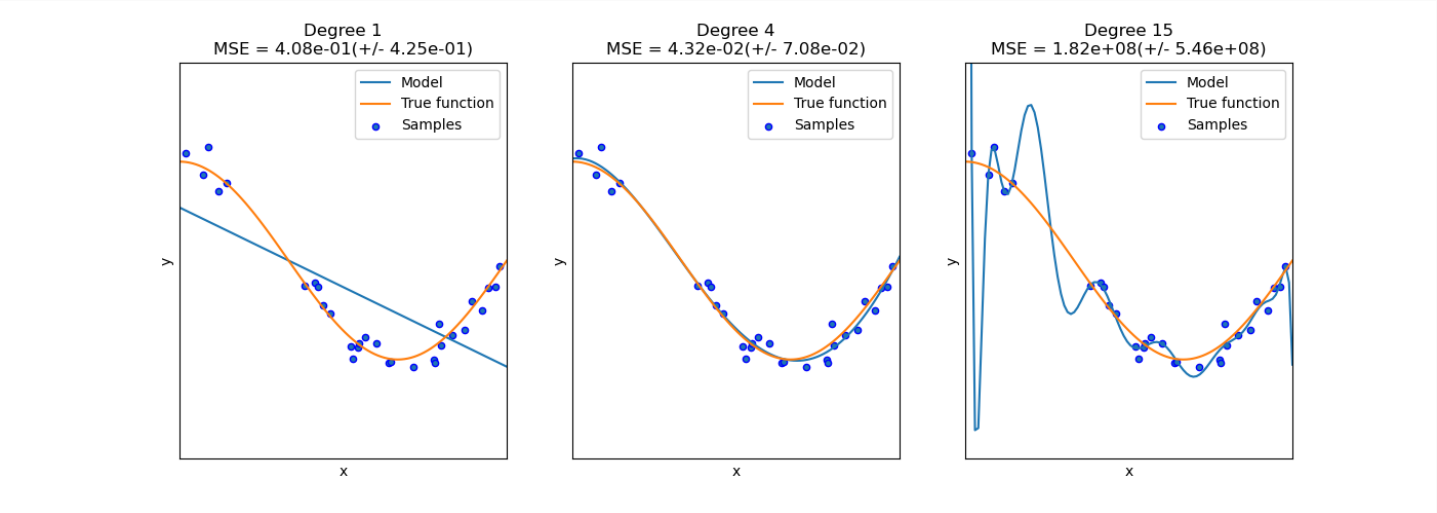
Under- and over-fitting

ML models usually have parameters that can be adjusted, called *hyperparameters*, to vary the level of complexity; e.g., depth of a neural network, degree of polynomial fit.

Under- and over-fitting

ML models usually have parameters that can be adjusted, called *hyperparameters*, to vary the level of complexity; e.g., depth of a neural network, degree of polynomial fit.

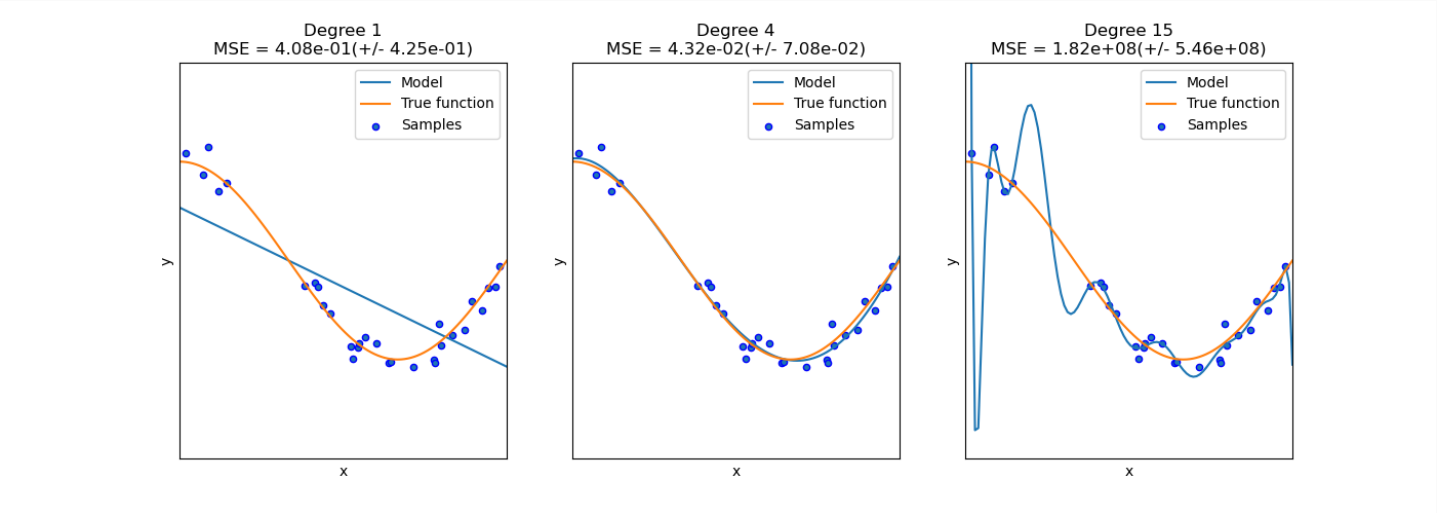
Models that are complex enough will often be able to do arbitrarily well on the training data. However, this may not do well with new data points.



Under- and over-fitting

ML models usually have parameters that can be adjusted, called *hyperparameters*, to vary the level of complexity; e.g., depth of a neural network, degree of polynomial fit.

Models that are complex enough will often be able to do arbitrarily well on the training data. However, this may not do well with new data points.



This failure to generalise is called *overfitting*. The performance on the training set is misleading.

Train and test set

How do we get a better estimate of the performance of the model on unseen data?

Train and test set

How do we get a better estimate of the performance of the model on unseen data?

Split dataset into a *training set* and *test set*. Train on the training set and only validate model performance on test data

Train and test set

How do we get a better estimate of the performance of the model on unseen data?

Split dataset into a *training set* and *test set*. Train on the training set and only validate model performance on test data

If the test error is significantly higher than the training error, this is a symptom of overfitting. Need to make the model simpler.

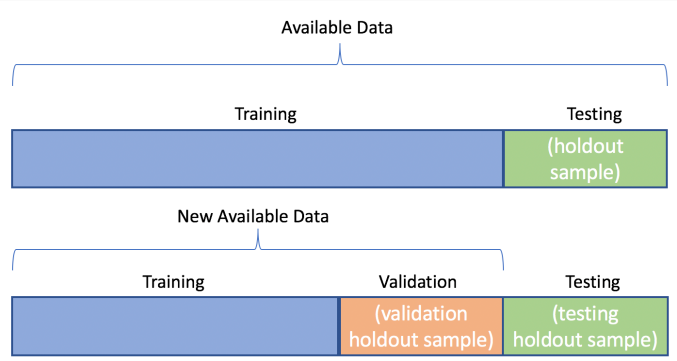
Train and test set

How do we get a better estimate of the performance of the model on unseen data?

Split dataset into a *training set* and *test set*. Train on the training set and only validate model performance on test data

If the test error is significantly higher than the training error, this is a symptom of overfitting. Need to make the model simpler.

How to choose hyperparameters? Split data again, and validate performance on a *validation set*.



Cross-validation

We can still overfit on the validation data!

Cross-validation

We can still overfit on the validation data!

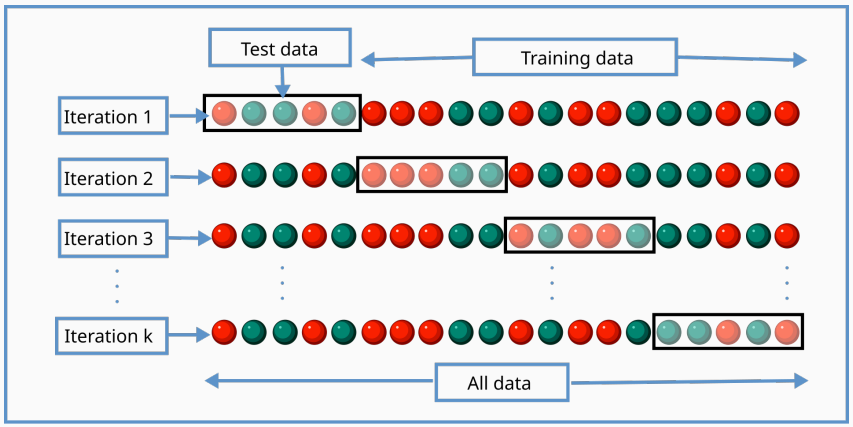
Cross-validation is a popular technique to try to fix this.

Cross-validation

We can still overfit on the validation data!

Cross-validation is a popular technique to try to fix this.

Set aside test set, and rotate training and validation sets:

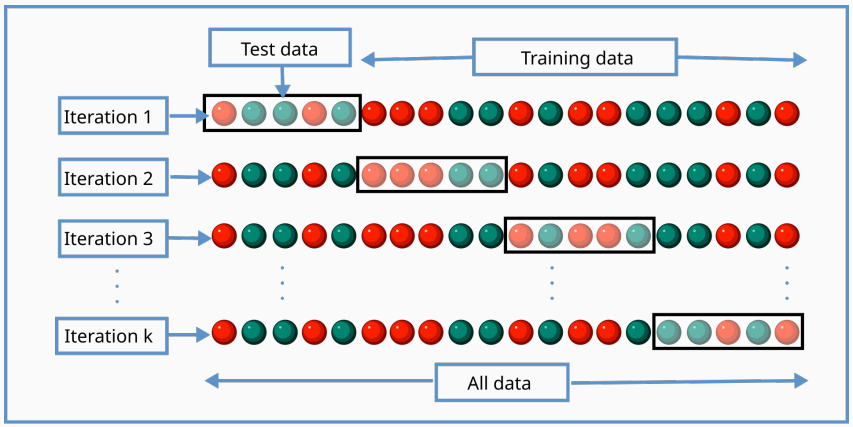


Cross-validation

We can still overfit on the validation data!

Cross-validation is a popular technique to try to fix this.

Set aside test set, and rotate training and validation sets:



With k splits, called *k-fold* cross-validation.

References i