HELMHOLTZ MUNICH
AIH Institute of AI for Health

Topological Machine Learning: The (W)Hole Truth

Lecture 4

Bastian Rieck (@Pseudomanifold)

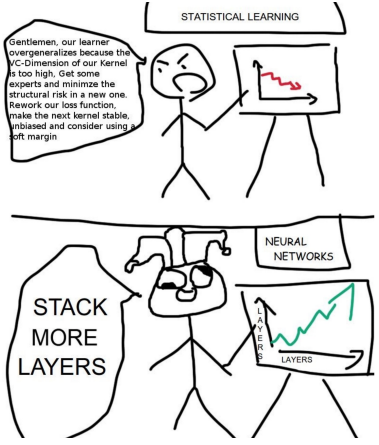
Preliminaries

Do you have feedback or any questions? Write to bastian.rieck@helmholtz-muenchen.de or reach out to @Pseudomanifold on Twitter. You can find the slides and additional information with links to more literature here:

`https://heidelberg.topology.rocks`

What is machine learning?

Two households, both alike in dignity...



Two households, both alike in dignity…

Classical machine learning

- *Somewhat* better interpretable
- Works well in small-sample regime
- Low requirements on compute

Deep learning

- *Often* like a ‘black box’
- Requires large amounts of data
- High compute requirements

Classical models are often not easy to train because they require ‘hand-crafted’ features. Deep models, on the other hand, are harder to ‘investigate’ *a posteriori*.

When to pick one over the other?

Two anecdotes

> In particular we ask whether supervised deep learning can be used for cell annotation, i.e. to predict cell-type labels from single-cell gene expression profiles. *After evaluating 10 classification methods across 14 datasets, we notably find that deep learning does not outperform classical machine-learning methods in the task.*
>
> (N. D. Köhler, M. Büttner, N. Andriamanga and F. J. Theis, 'Deep learning does not outperform classical machine learning for cell-type annotation', Preprint, 2021)

> *One of the most surprising results is the highly-competitive performance of our DTW-KNN ensemble classifier, whose performance for earlier horizons exceeds all of the other methods, despite its conceptual simplicity.* While this is highly relevant for the early detection of sepsis, the DTW-KNN classifier suffers from some practical limitations that impede online monitoring scenarios and its application to very large patient cohorts.
>
> (M. Moor, M. Horn, **B. Rieck**, D. Roqueiro and K. Borgwardt, 'Early Recognition of Sepsis with Gaussian Process Temporal Convolutional Networks and Dynamic Time Warping', *Proceedings of the 4th Machine Learning for Healthcare Conference*, ed. by F. Doshi-Velez et al., Proceedings of Machine Learning Research 106, PMLR, 2019, pp. 2–26, arXiv: `1902.01659 [cs.LG]`)

A brief overview of some tried and tested classical machine learning methods

Starting with the data

‘Wine’ data set

- 178 samples
- 13 attributes
- 3 classes, corresponding to cultivars

Starting with the data

'Wine' data set

- 178 samples
- 13 attributes
- 3 classes, corresponding to cultivars

Normally, we should try to understand all of the attributes: are they *continuous* or *discrete*? What are their ranges?

Starting with the data

‘Wine’ data set

- 178 samples
- 13 attributes
- 3 classes, corresponding to cultivars

Normally, we should try to understand all of the attributes: are they *continuous* or *discrete*? What are their ranges?

Attributes

- Alcohol
- Malic acid
- Ash
- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols
- Proanthocyanins
- Colour intensity
- Hue
- OD280/OD315 value of diluted wines
- Proline

Starting with the data

‘Wine’ data set

- 178 samples
- 13 attributes
- 3 classes, corresponding to cultivars

Normally, we should try to understand all of the attributes: are they *continuous* or *discrete*? What are their ranges?

Attributes

- Alcohol
- Malic acid
- Ash
- Alcalinity of ash
- Magnesium
- Total phenols
- Flavanoids
- Nonflavanoid phenols
- Proanthocyanins
- Colour intensity
- Hue
- OD280/OD315 value of diluted wines
- Proline

Source: `https://archive.ics.uci.edu/ml/datasets/wine`

Logistic regression

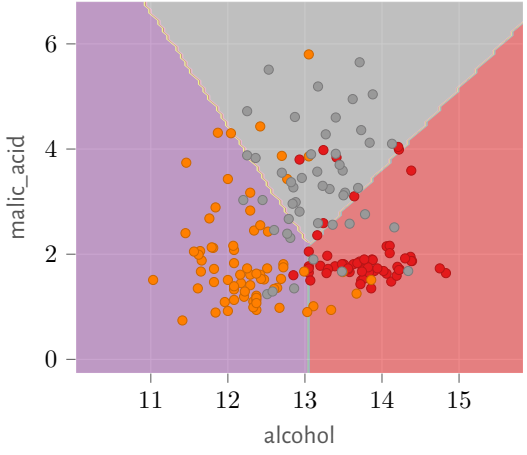
Brief description

Logistic regression models the probability of an event taking place by having the log-odds for the event be a *linear combination* of one or more independent variables. We then estimate the parameters, i.e. the coefficients, of such a model.

Logistic regression

Brief description

Logistic regression models the probability of an event taking place by having the log-odds for the event be a *linear combination* of one or more independent variables. We then estimate the parameters, i.e. the coefficients, of such a model.

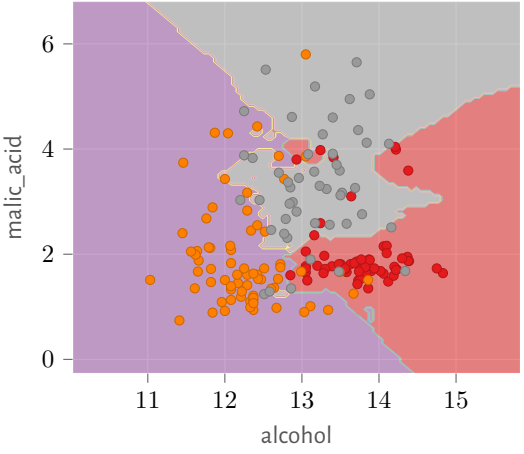


k-nearest neighbours classification and regression

- The quintessential 'lazy learner'
- Requires only a similarity measure and a neighbourhood choice
- Surprisingly effective and efficient

k-nearest neighbours classification and regression

- The quintessential 'lazy learner'
- Requires only a similarity measure and a neighbourhood choice
- Surprisingly effective and efficient



Support vector machines

Given data points $\mathbf{x}_i$ with labels $y_i \in \{+1, -1\}$, find weights $\mathbf{w}$ that minimise $\lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n} \max\bigl(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b)\bigr)\right]$.

Support vector machines

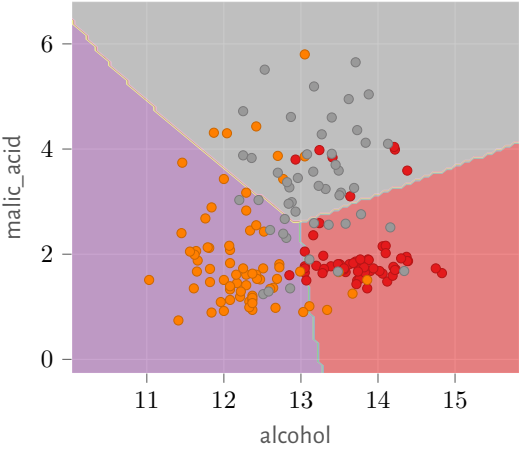
Given data points $\mathbf{x}_i$ with labels $y_i \in \{+1, -1\}$, find weights $\mathbf{w}$ that minimise $\lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n} \max\bigl(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b)\bigr)\right]$.

SVMs also support *kernels*, i.e. domain-specific similarity measures!

Support vector machines

Given data points $\mathbf{x}_i$ with labels $y_i \in \{+1, -1\}$, find weights $\mathbf{w}$ that minimise $\lambda\|\mathbf{w}\|^2 + \left[\frac{1}{n}\sum_{i=1}^{n} \max\left(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b)\right)\right]$.

SVMs also support *kernels*, i.e. domain-specific similarity measures!



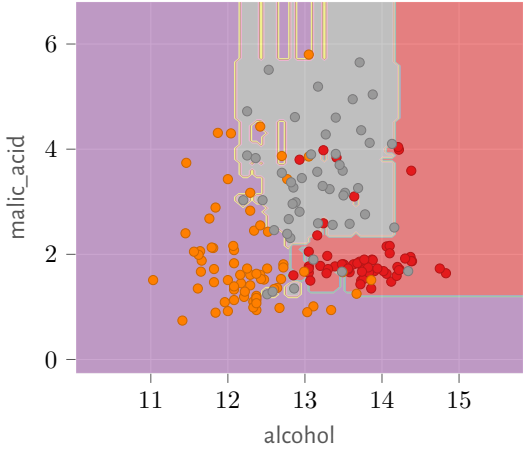
Random forests

- Learn many small *decision trees* and use their joint predictions
- Leads to *highly interpretable* models!
- Still an active area of research[1]

[1] G. Ke et al., 'LightGBM: A Highly Efficient Gradient Boosting Decision Tree', *Advances in Neural Information Processing Systems*, ed. by I. Guyon et al., vol. 30, Curran Associates, Inc., 2017

Random forests

- Learn many small *decision trees* and use their joint predictions
- Leads to *highly interpretable* models!
- Still an active area of research[1]



[1] G. Ke et al., 'LightGBM: A Highly Efficient Gradient Boosting Decision Tree', *Advances in Neural Information Processing Systems*, ed. by I. Guyon et al., vol. 30, Curran Associates, Inc., 2017

How to train such models?

Theory

Ideal scenario

- Split data into training, validation, and test data sets
- Fit model on *training* data, using the *validation* data for hyperparameter tuning, and finally evaluate on *test* data
- Look at *test* data only once!

Data-sparse scenario

- Follow *cross-validation* procedure, which ensures that every split of the data is used exactly once for testing
- Repeat this procedure multiple times to obtain estimates of generalisation performance
- Can also be combined with scenario above

How to train such models?

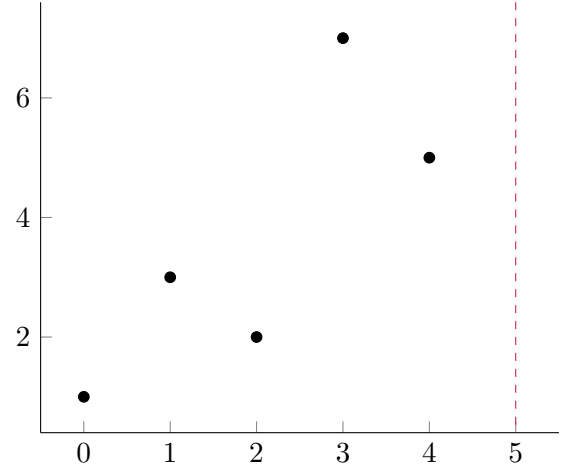
Practice

```
cv = StratifiedKFold(n_splits=n_folds, shuffle=True, random_state=42)
for train_index, test_index in cv.split(X, y):
    X_train, y_train = X[train_index], y[train_index]
    X_test, y_test = X[test_index], y[test_index]

    clf = LogisticRegression(
            solver="saga",
            class_weight="balanced",
            max_iter=5000,
            random_state=42
    )

    grid_search = GridSearchCV(
        clf,
        param_grid=param_grid,
        cv=n_folds,
        scoring="accuracy",
        n_jobs=-1,
    )

    grid_search.fit(X_train, y_train)
    y_pred = grid_search.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)

```

Gaussian Processes

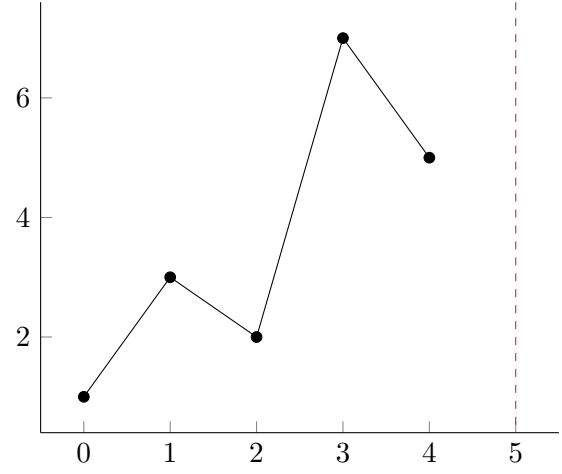
Motivation

How to interpolate these points?



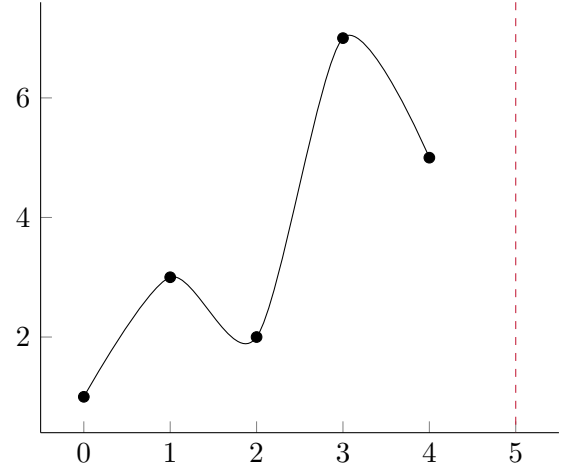
Motivation

How to interpolate these points?



Motivation

How to interpolate these points?



Questions

- Given known data, what is a *likely* value for an unknown x coordinate?
- Can we determine the *confidence* of our prediction?
- What is the *mean* of all possible predictions?

Main building block

Multivariate Gaussian distributions

$$f(x_1, \ldots, x_k \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{\exp\left(-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^\mathsf{T} \boldsymbol{\Sigma}^{-1} (\mathbf{x} - \boldsymbol{\mu})\right)}{\sqrt{(2\pi)^k \det \boldsymbol{\Sigma}}}$$

- $\mathbf{x}$ is a k-dimensional (column) vector
- $\boldsymbol{\mu}$ is a k-dimensional (column) vector representing the *mean* of every variable
- $\boldsymbol{\Sigma}$ is a $k \times k$ *covariance* matrix
- One often writes $\mathbf{x} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$

Conjugate distribution

If $\mathbf{x} = (\mathbf{x_1}, \mathbf{x_2})$ is *jointly Gaussian,* then the *marginals* and the *posterior* are also Gaussian!

Gaussian processes

C. E. Rasmussen and C. K. I. Williams, *Gaussian Processes for Machine Learning*, Adaptive Computation and Machine Learning series, MIT Press, 2005

Definition

A *Gaussian process* generalises a multivariate normal distribution to *infinitely many* variables. The assumption is that every finite number of variables follows some Gaussian distribution.

Properties

Gaussian processes can work with different *kernels*, which serve as covariance functions here.

- $k(x, x') = \sigma_f^2 \exp\left(-\frac{(x-x')^2}{2l^2}\right)$ *(squared exponential kernel)*
- $k(x, x') = \sigma_b^2 + \sigma_f^2 (x - c)(x' - c)$ *(linear kernel)*

How to perform the regression

Given known values for the function f, we want to predict new values f_*. We thus have

$$\begin{pmatrix} f \\ f_* \end{pmatrix} = \mathcal{N}\left(\begin{pmatrix} \boldsymbol{\mu} \\ \boldsymbol{\mu}_* \end{pmatrix}, \begin{pmatrix} \mathbf{K} & \mathbf{K}_* \\ \mathbf{K}_*^T & \mathbf{K}_{**} \end{pmatrix}\right),$$

where we write $\mathbf{K}_*$ to denote the matrix containing all $k(x, x_*)$ (and similarly for the other matrices). According to the rules of Gaussian distributions, the *posterior* is given by

$$p(f_* \mid f) = \mathcal{N}(f_* \mid \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*),$$

where:

$$\boldsymbol{\mu}_* = \boldsymbol{\mu}(\mathbf{x}_*) + \mathbf{K}_*^T \mathbf{K}^{-1}(f - \boldsymbol{\mu}(\mathbf{x}))$$
$$\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{K}_*$$

Parameter estimation

We can estimate the parameters that make our observed data most likely. Suppose we have $k(x, x') = \sigma_f^2 \exp\left(-\frac{(x-x')^2}{2l^2}\right)$, we can find good values for σ_f and l by maximising *marginal likelihood*:

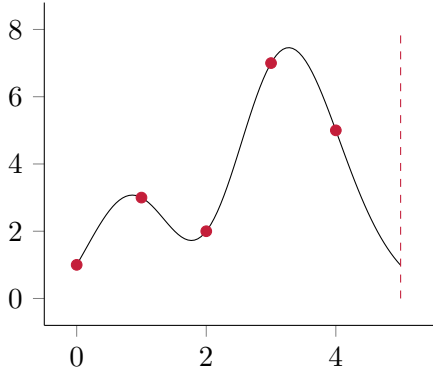
$$\log p(\mathbf{y} \mid \mathbf{X}) = -\frac{1}{2}\mathbf{y}\mathbf{K}_y^{-1}\mathbf{y} - \frac{1}{2}\log\det\mathbf{K}_y - \frac{n}{2}\log(2\pi)$$

This is possible using *gradient descent* or *Nelder–Mead*, for example.

How to use this?

Sampling from the *posterior*

Recall that $p(f_* \mid f) = \mathcal{N}(f_* \mid \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$, where $\boldsymbol{\mu}_* = \boldsymbol{\mu}(\mathbf{x}_*) + \mathbf{K}_*^T \mathbf{K}^{-1}(f - \boldsymbol{\mu}(\mathbf{x}))$ and $\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{K}_*$. In our simple example, let's assume that $\boldsymbol{\mu} = 0$, so everything reduces to evaluating some matrices.

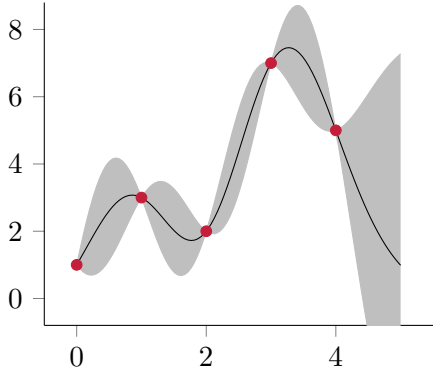


Notice that this is still a *random* sample!

How to use this?

Sampling from the *posterior*

Recall that $p(f_* \mid f) = \mathcal{N}(f_* \mid \boldsymbol{\mu}_*, \boldsymbol{\Sigma}_*)$, where $\boldsymbol{\mu}_* = \boldsymbol{\mu}(\mathbf{x}_*) + \mathbf{K}_*^T \mathbf{K}^{-1}(f - \boldsymbol{\mu}(\mathbf{x}))$ and $\boldsymbol{\Sigma}_* = \mathbf{K}_{**} - \mathbf{K}_*^T \mathbf{K}^{-1} \mathbf{K}_*$. In our simple example, let's assume that $\boldsymbol{\mu} = 0$, so everything reduces to evaluating some matrices.



Notice that this is still a *random* sample!

Why kernels are *still* relevant

Recap

Kernel

Given a set $\mathcal{X}$, a function $k\colon \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ is a *kernel* if there is a Hilbert space $\mathcal{H}$ (an inner product space that is also a complete metric space) and a map $\Phi\colon \mathcal{X} \to \mathcal{H}$, such that $k(x, y) = \langle \Phi(x), \Phi(y) \rangle_{\mathcal{H}}$ for all $x, y \in \mathcal{X}$.

Properties

Model similarity between structured and unstructured objects. Many existing machine learning algorithms can be rephrased in terms of kernels using the *kernel trick*.[2]

[2] B. Schölkopf and A. J. Smola, *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*, Adaptive Computation and Machine Learning series, MIT Press, 2018

Maximum mean discrepancy (MMD)

Comparing distributions of objects using kernels

Given n samples $X = \{x_1, \ldots, x_n\} \subseteq \mathcal{X}$ and m samples $Y = \{y_1, \ldots, y_m\} \subseteq \mathcal{X}$, the biased empirical estimate of the MMD between X and Y is

$$\mathrm{MMD}^2(X, Y) := \frac{1}{n^2} \sum_{i,j=1}^{n} k(x_i, x_j) + \frac{1}{m^2} \sum_{i,j=1}^{m} k(y_i, y_j) - \frac{2}{nm} \sum_{i=1}^{n} \sum_{j=1}^{m} k(x_i, y_j).$$

This formulation is due to Gretton et al.;[3] under some mild conditions, MMD is a metric on the space of probability distributions.

[3] A. Gretton, K. Borgwardt, M. Rasch, B. Schölkopf and A. J. Smola, 'A Kernel Method for the Two-Sample-Problem', *Advances in Neural Information Processing Systems*, ed. by B. Schölkopf, J. C. Platt and T. Hoffman, vol. 19, MIT Press, 2007, pp. 513–520

Deep Learning

Some terminology

- *Batch* or *Mini-batch*: a subsample of the data set
- *Epoch*: how many times the entire data set was processed
- *Loss*: a function that determines how well the network fits the data

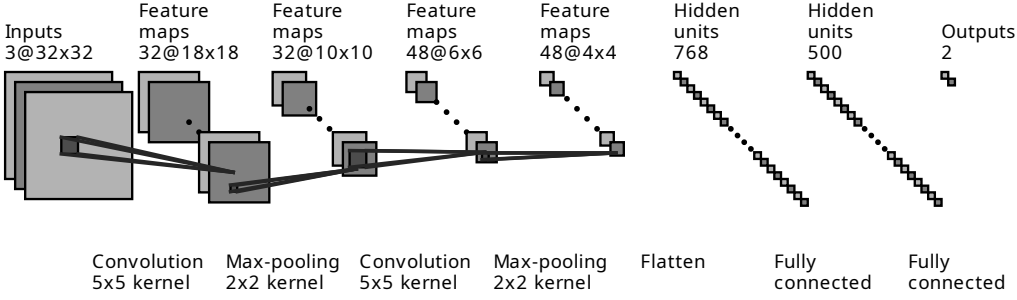
Backpropagation

Adjust weights—connection strengths—in the neural network to minimise the error: compute gradient of the loss function for this batch and update weights.

Some common architectures

- Fully-connected neural networks (FCNs)
- Convolutional neural networks (CNNs)
- Autoencoders (AEs)
- Recurrent neural networks (RNNs)

Convolutional neural networks



The ecosystem of deep learning

- ☆ `Keras`
- ☆ `PyTorch`
- ☆ `TensorFlow`

Example

Getting some data

```python
import torch
from torch import nn
from torch.utils.data import DataLoader
from torchvision import datasets
from torchvision.transforms import ToTensor

training_data = datasets.FashionMNIST(
    root="data", train=True, download=True,
    transform=ToTensor()
)

test_data = datasets.FashionMNIST(
    root="data", train=False, download=True,
    transform=ToTensor()
)

train_dataloader = DataLoader(training_data, batch_size=64)
test_dataloader = DataLoader(test_data, batch_size=64)

```

(pytorch.org)

Example

Defining a neural network

```python
class NeuralNetwork(nn.Module):
    def __init__(self):
        super(NeuralNetwork, self).__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = nn.Sequential(
            nn.Linear(28*28, 512),
            # Non-linear operation!
            nn.ReLU(),
            nn.Linear(512, 512),
            nn.ReLU(),
            nn.Linear(512, 10),
        )

    def forward(self, x):
        x = self.flatten(x)
        logits = self.linear_relu_stack(x)
        return logits

model = NeuralNetwork()

```

(pytorch.org)

Example

Train loop

```python
def train_loop(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    for batch, (X, y) in enumerate(dataloader):
        # Compute prediction and loss
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

```

(pytorch.org)

Example

Test loop

```python
def test_loop(dataloader, model, loss_fn):
    size = len(dataloader.dataset)
    num_batches = len(dataloader)
    test_loss, correct = 0, 0

    with torch.no_grad():
        for X, y in dataloader:
            pred = model(X)
            test_loss += loss_fn(pred, y).item()
            correct += (pred.argmax(1) == y).type(torch.float).sum().item()

    test_loss /= num_batches
    correct /= size

```

(pytorch.org)

Example

Running the training and testing process

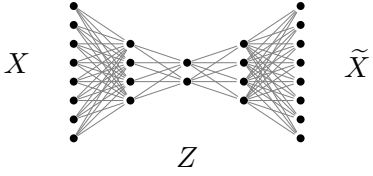
```
learning_rate = 1e-3
batch_size = 64
epochs = 10

loss_fn = nn.CrossEntropyLoss()
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate)

for t in range(epochs):
    train_loop(train_dataloader, model, loss_fn, optimizer)
    test_loop(test_dataloader, model, loss_fn)

```

(pytorch.org)

A lot of moving parts

- Architecture
- Loss function
- Optimiser
- Additional elements: batch norm, attention, …

Autoencoders



Terminology

- $X \in \mathbb{R}^D$: input data
- $Z \in \mathbb{R}^d$: latent representation
- $\widetilde{X} \in \mathbb{R}^D$: reconstructed data

Properties

- Typically, $D \gg d$.
- Use loss function $\mathcal{L}\left(X, \widetilde{X}\right)$ to measure quality of reconstruction.

Why autoencoders?

- Encoder–decoder architecture (we learn the *identity* function).
- 'Middle' layer serves as *bottleneck* or *latent representation*.
- Latent representations can be used for visualisation, interpolation, clustering, and much more.

A simple autoencoder

- Encoder: linear transformation $\mathbb{R}^D \to \mathbb{R}^2$
- Decoder: linear transformation $\mathbb{R}^2 \to \mathbb{R}^D$
- Loss function: mean squared error, $\mathcal{L}\left(X, \widetilde{X}\right) := \left\| X - \widetilde{X} \right\|_2^2$

A simple autoencoder

Some reconstructions

0 epochs

A simple autoencoder

Some reconstructions

10 epochs

A simple autoencoder

Some reconstructions

20 epochs

A simple autoencoder

Some reconstructions

30 epochs

A simple autoencoder

Some reconstructions

40 epochs

A simple autoencoder

Some reconstructions

50 epochs

PyTorch Lightning

Overall training loop

```python
from .data import MNISTDataModule
from .models import LinearAutoencoder
from pytorch_lightning import Trainer

if __name__ == '__main__':
    dm = MNISTDataModule()
    dm.prepare_data()
    dm.setup()

    model = LinearAutoencoder(input_dim=dm.input_dim, bottleneck_dim=2)

    # There's a lot more parameters for a `trainer` class, but this is
    # sufficient for our brief example.
    trainer = Trainer(max_epochs=50)
    trainer.fit(model, dm)

```

PyTorch Lightning

Defining a model

```python
import torch.nn as nn
import torch.nn.functional as F
import pytorch_lightning as pl

class LinearAutoencoder(pl.LightningModule):
  def __init__(self, input_dim, bottleneck_dim=2, lr=0.01):
      super().__init__()

      self.encoder = nn.Linear(input_dim, bottleneck_dim)
      self.decoder = nn.Linear(bottleneck_dim, input_dim)

      self.lr = 0.01
      self.loss_fn = F.mse_loss

```

PyTorch Lightning

Defining a model, continued

```
    def forward(self, x):
        z = self.encoder(x)
        x_hat = self.decoder(z)
        loss = self.loss_fn(x_hat, x)
        return x_hat, z, loss

    def training_step(self, batch, batch_idx):
        x, y = batch

        x = x.view(x.size(0), -1)
        _, _, loss = self(x)

        self.log('train_loss', loss)
        return loss

    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.lr)
        return optimizer

```

PyTorch Lightning

More complex operations

Some highlights

- Early stopping
- Learning rate adjustments
- Model checkpoints
- Distributed training

Variational autoencoders

We can make the autoencoder learn the *parameters* of our input distribution to sample *new* data points from it.

Variational autoencoders

The basic skeleton

```python
class BetaVAE(pl.LightningModule):
    def __init__(self, input_dim, bottleneck_dim=32, beta=1, lr=1e-3):
        super(BetaVAE, self).__init__()

        self.bottleneck_dim = bottleneck_dim

        # The specific choice of encoder/decoder is irrelevant here.
        self.encoder = ...
        self.decoder = ...

        # Final layer for encoding the parameters of the distribution
        self.fc_mu = nn.Linear(16, bottleneck_dim)
        self.fc_logvar = nn.Linear(16, bottleneck_dim)

        # For sampling values that parametrise our distribution
        self.fc_z = nn.Linear(bottleneck_dim, 16)

```

Variational autoencoders

Encoding and decoding

```
    def encode(self, x):
        x = self.encoder(x)
        x = x.view(x.shape[0], -1)
        return self.fc_mu(x), self.fc_logvar(x)

    def sample(self, mu, logvar):
        std = torch.exp(0.5 * logvar)  # e^(1/2 * log(std^2))
        eps = torch.randn_like(std)    # random ~ N(0, 1)
        return eps.mul(std).add_(mu)

    def decode(self, z):
        z = self.fc_z(z)
        z = z.view(z.shape[0], -1)
        return self.decoder(z)

    def forward(self, x):
        mu, logvar = self.encode(x)
        z = self.sample(mu, logvar)
        x_hat = self.decode(z)
        return x_hat, mu, self.loss_fn(x_hat, x, mu, logvar)

```

Variational autoencoders

Loss function

```
1       def loss_fn(self, x_hat, x, mu, logvar):
2           reconstruction_loss = F.mse_loss(
3               x_hat,
4               x,
5               reduction='sum' # We reduce it ourselves later on!
6           )
7
8           # This follows the loss provided by Kingma and Welling in
9           # 'Auto-Encoding Variational Bayes', Appendix B. The loss
10          # assumes that all distributions are Gaussians.
11          kl = -0.5 * torch.sum(1 + logvar - mu.pow(2) - logvar.exp())
12
13          return (reconstruction_loss + self.beta * kl) / x.shape[0]
14
```

The loss term boils down to a Kullback–Leibler divergence between the prior distribution and the posterior distribution:

$$\mathrm{KL}\left(q_\phi(\mathbf{z}) \parallel p_\theta(\mathbf{z})\right)$$

Variational autoencoders

Some reconstructions

0 epochs

Variational autoencoders

Some reconstructions

10 epochs

Variational autoencoders

Some reconstructions

20 epochs

Variational autoencoders

Some reconstructions

30 epochs

Variational autoencoders

Some reconstructions

40 epochs

Variational autoencoders

Some reconstructions

50 epochs

Sampling from the learned distribution

Pick $z \sim \mathcal{N}(0, 1)$ and *decode* it. This yields a valid image of the distribution!

Brief summary

- Taming the dragon: PyTorch Lightning makes working with neural networks easy
- Autoencoders are available in many flavours (variational, topological, …)
- *Acta non verba*: try them out yourselves!

```
https://github.com/Pseudomanifold/bestiary-autoencoders
```