Deep Learning in Python

Lecture 3: Training, Testing & Optimisation

Dr Andreas C. S. Jørgensen

Imperial College London

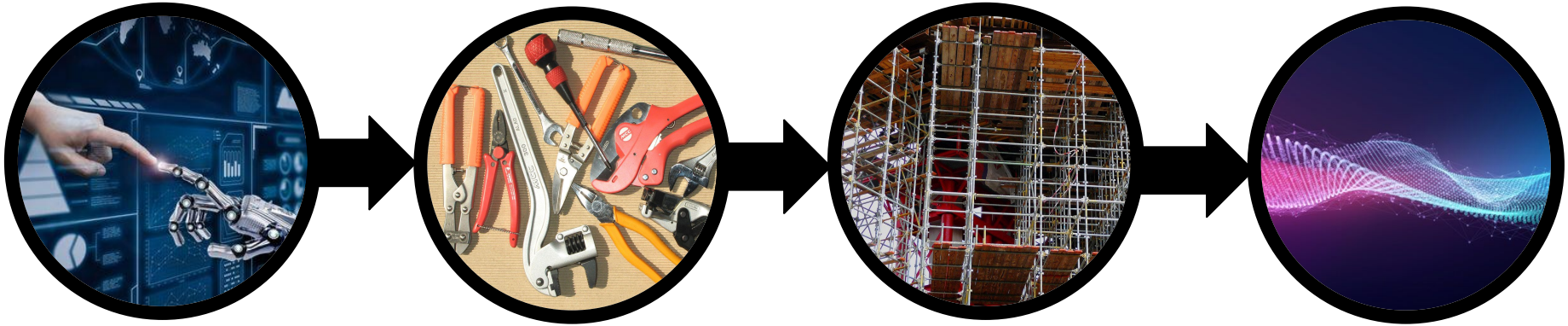
Imperial-X

Outline

Hands-on course in Deep Learning using Python (PyTorch) with focus on computer vision.

Lecture 1	Deep Learning concepts and terminology. PyTorch commands, syntax and libraries (torch, torchvision). Building deep neural networks.
Lecture 2	Convolutional Neural Networks (CNN). Building CNN in PyTorch. Using GPUs with PyTorch.
Lecture 3	Optimisation: How to train and test neural networks robustly. Regularisation and hyper-parameter tuning.

Outline



1) Concepts

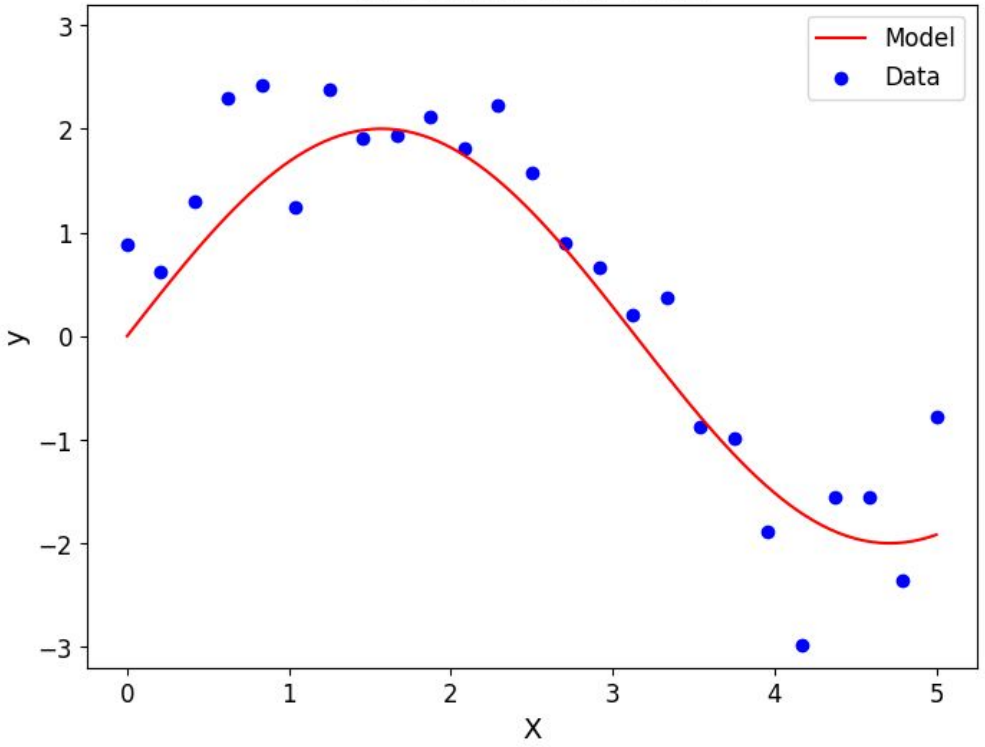
2) Toolkit (PyTorch)

3) Combine 1 and 2 to optimise DNNs

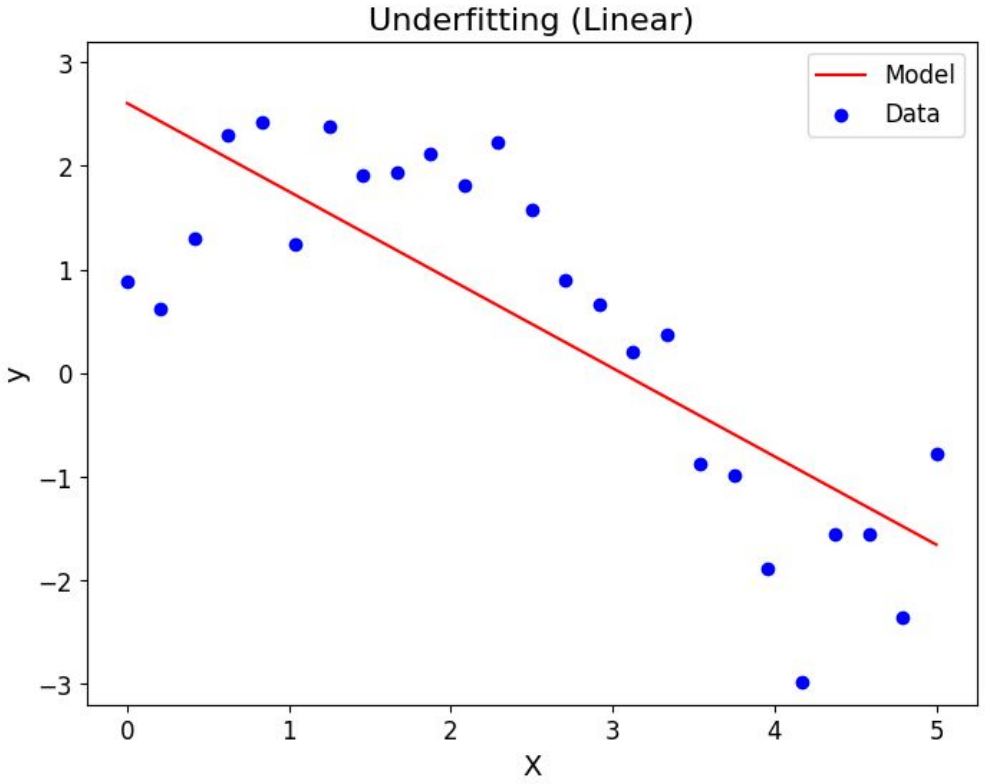
4) Beyond CNNs

Training, validation & testing

Under- & overfitting



Under- & overfitting

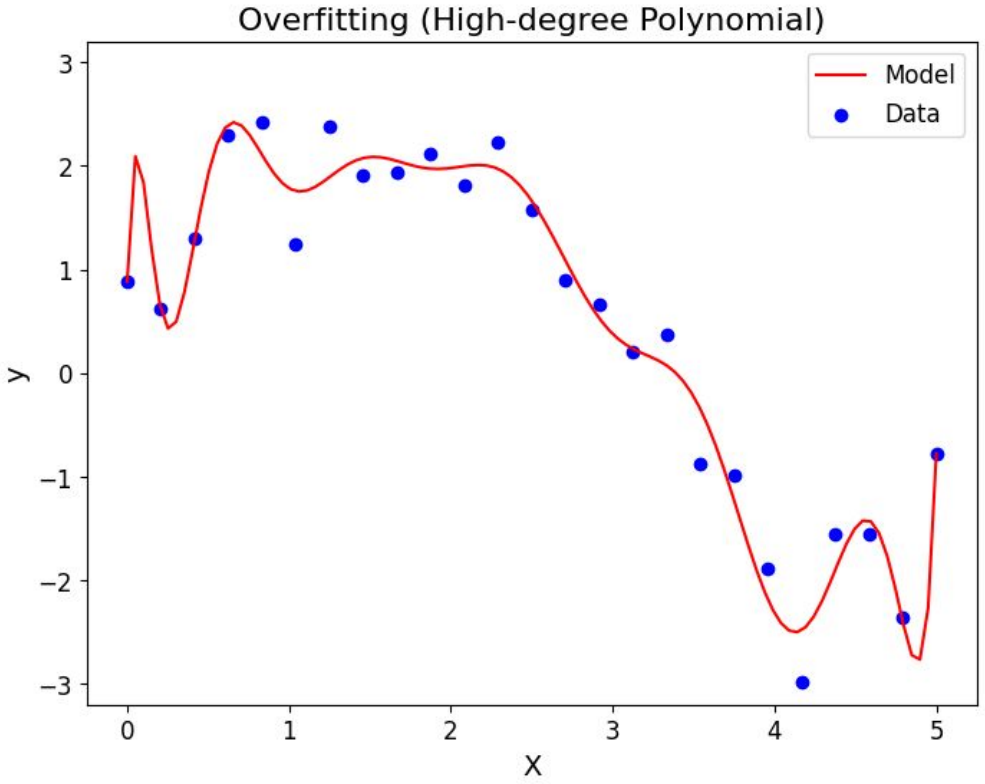


Bias: The difference between the expected prediction of our model and the true value we are trying to predict.

Variance: Sensitivity of the model to the fluctuations in the training dataset.

The polynomial exhibits low bias and high variance.

Under- & overfitting



Bias: The difference between the expected prediction of our model and the true value we are trying to predict.

Variance: Sensitivity of the model to the fluctuations in the training dataset.

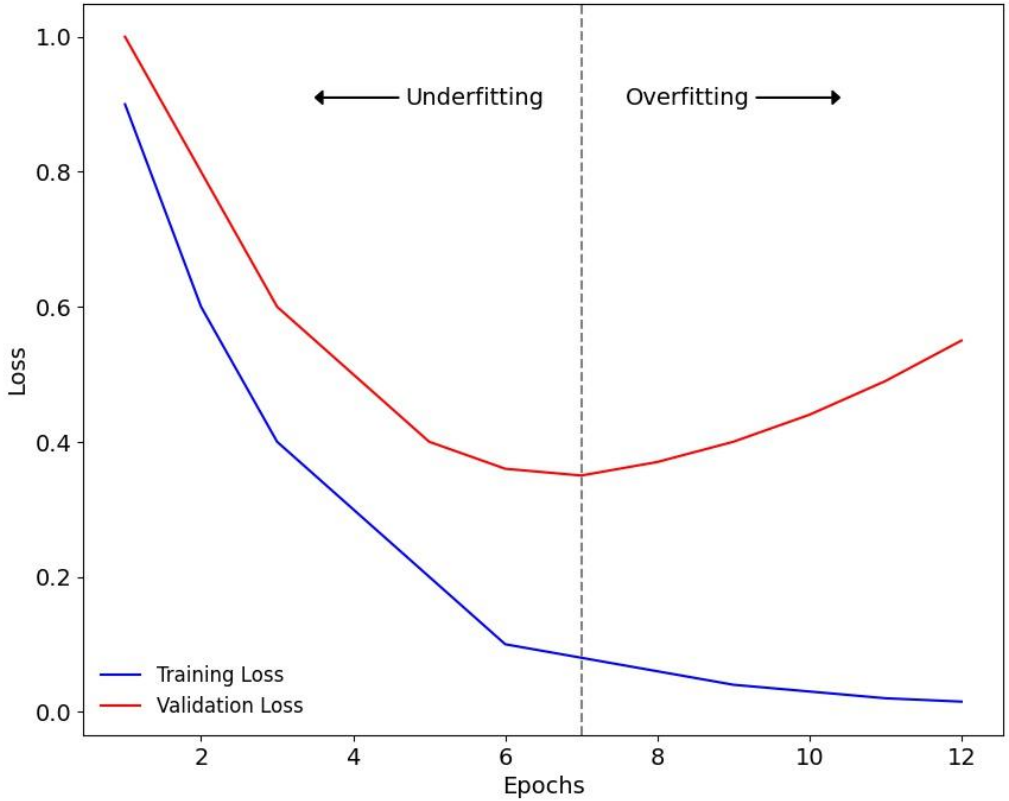
The polynomial exhibits low bias and high variance.

Training curves

Training curves depict the performance of the model over the epochs.

Overfitting: If the training loss decreases while the validation loss increases, the model likely overfits the training data.

Underfitting: When both training and validation losses remain high, the model may underfit the data, suggesting insufficient complexity.



Regularisation

- **Early stopping:** Stop the training if validation loss stalls or worsens for a number of epochs (patience).

Regularisation

- **Early stopping:** Stop the training if validation loss stalls or worsens for a number of epochs (patience).

- **L1 regularisation:** $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \sum_{i=1}^{N} |w_i|$

Loss

Regularisation hyperparameter

parameters

Regularisation

- **Early stopping:** Stop the training if validation loss stalls or worsens for a number of epochs (patience).

- **L1 regularisation:** $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \sum_{i=1}^{N} |w_i|$

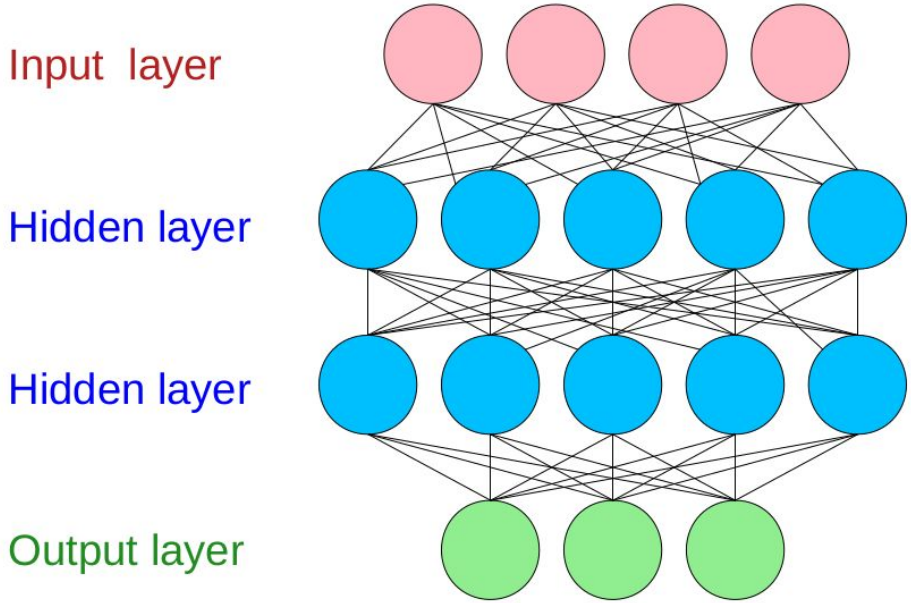
- **L2 regularisation:** $\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{data}} + \lambda \sum_{i=1}^{N} |w_i|^2$

Loss

Regularisation hyperparameter

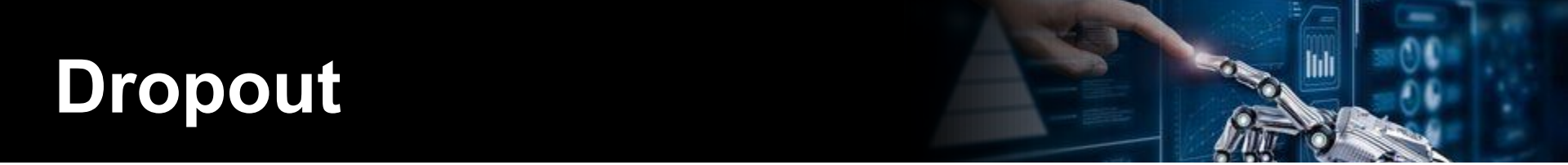
parameters

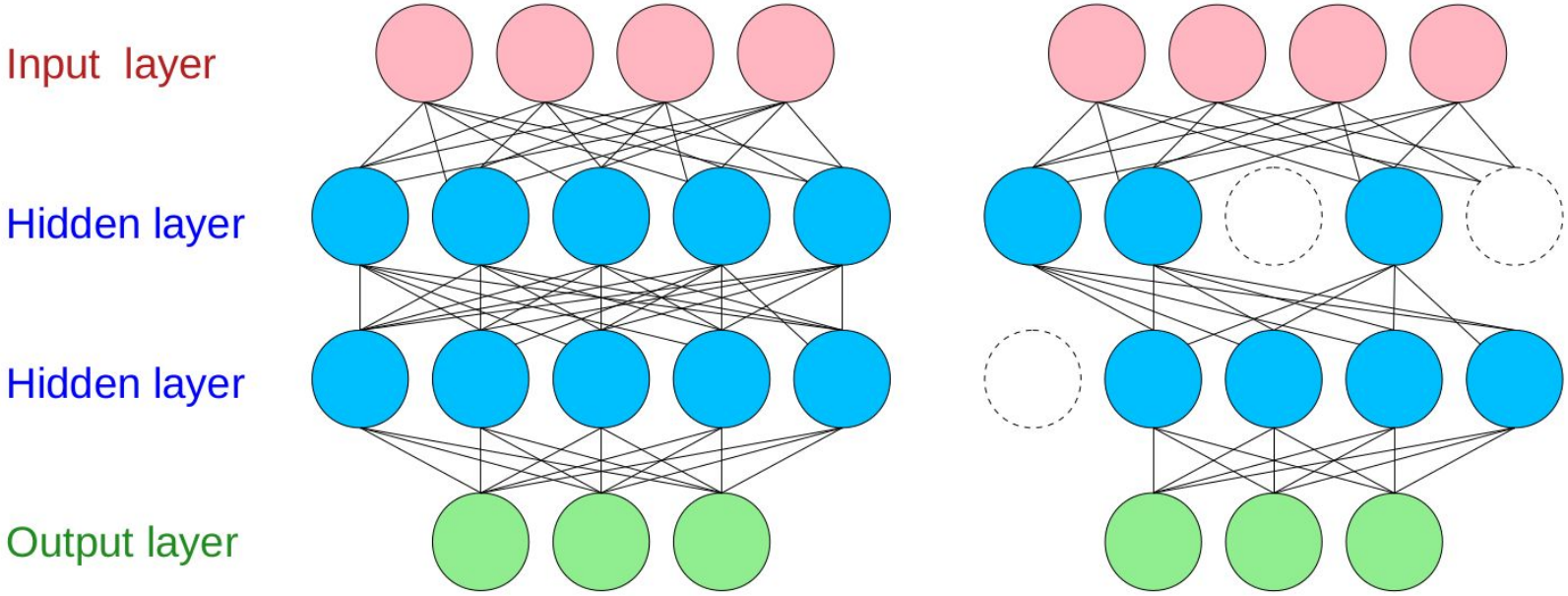
Dropout



Normal neural network

Dropout

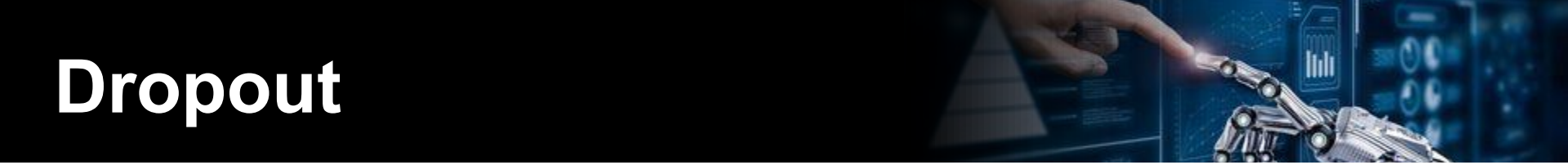


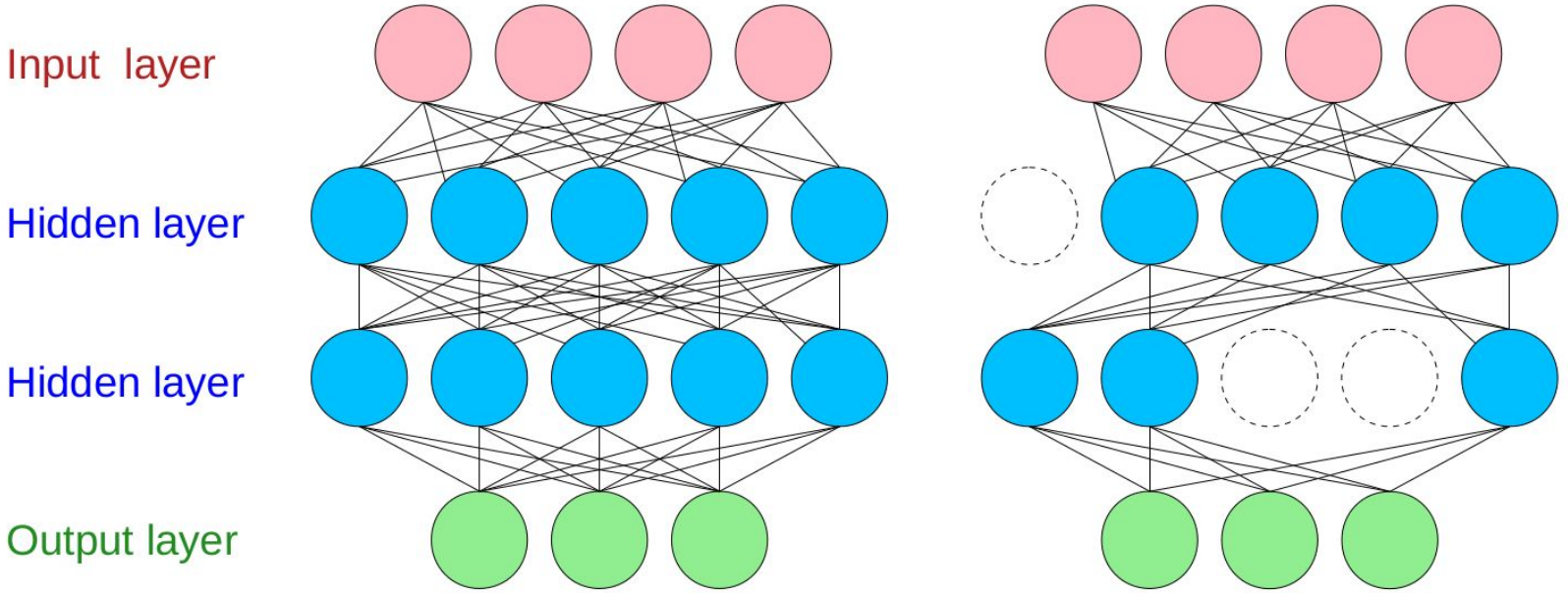


Normal neural network

Neural network with dropout

Dropout





Normal neural network

Neural network with dropout

Hyper-parameter tuning

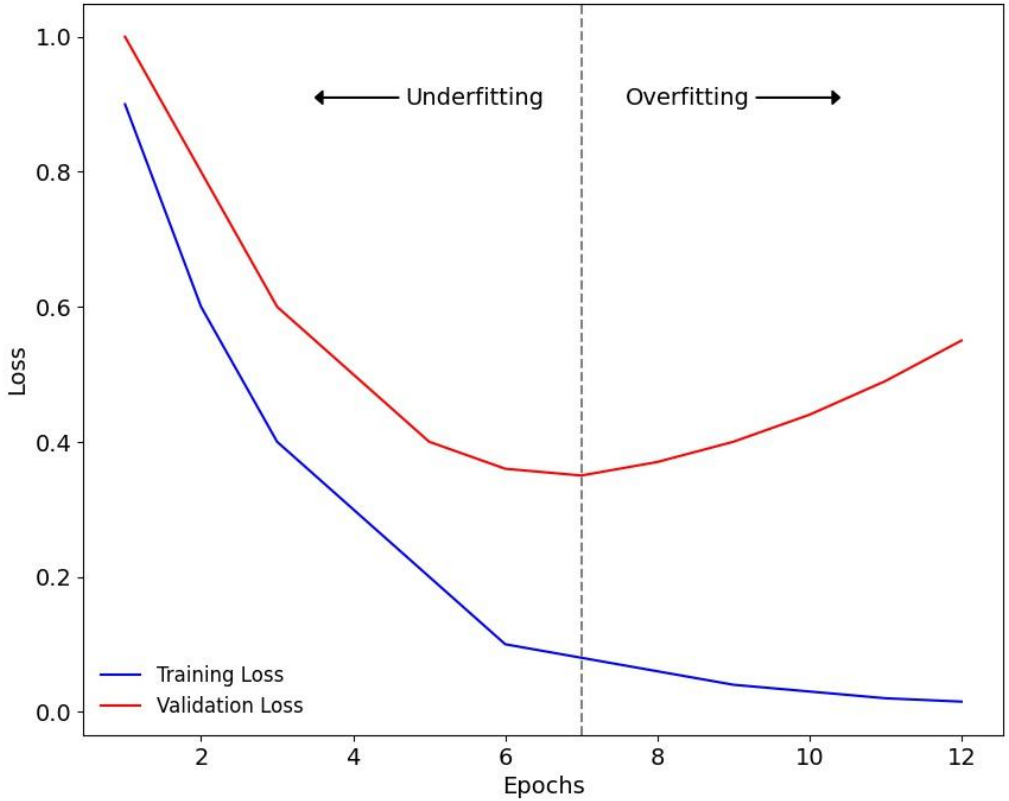
The presented neural networks contain several hyper-parameters (e.g., learning rate, λ, number of epochs…).

Choose the hyper-parameter combination that leads to the best performance! E.g., perform a **grid search.**

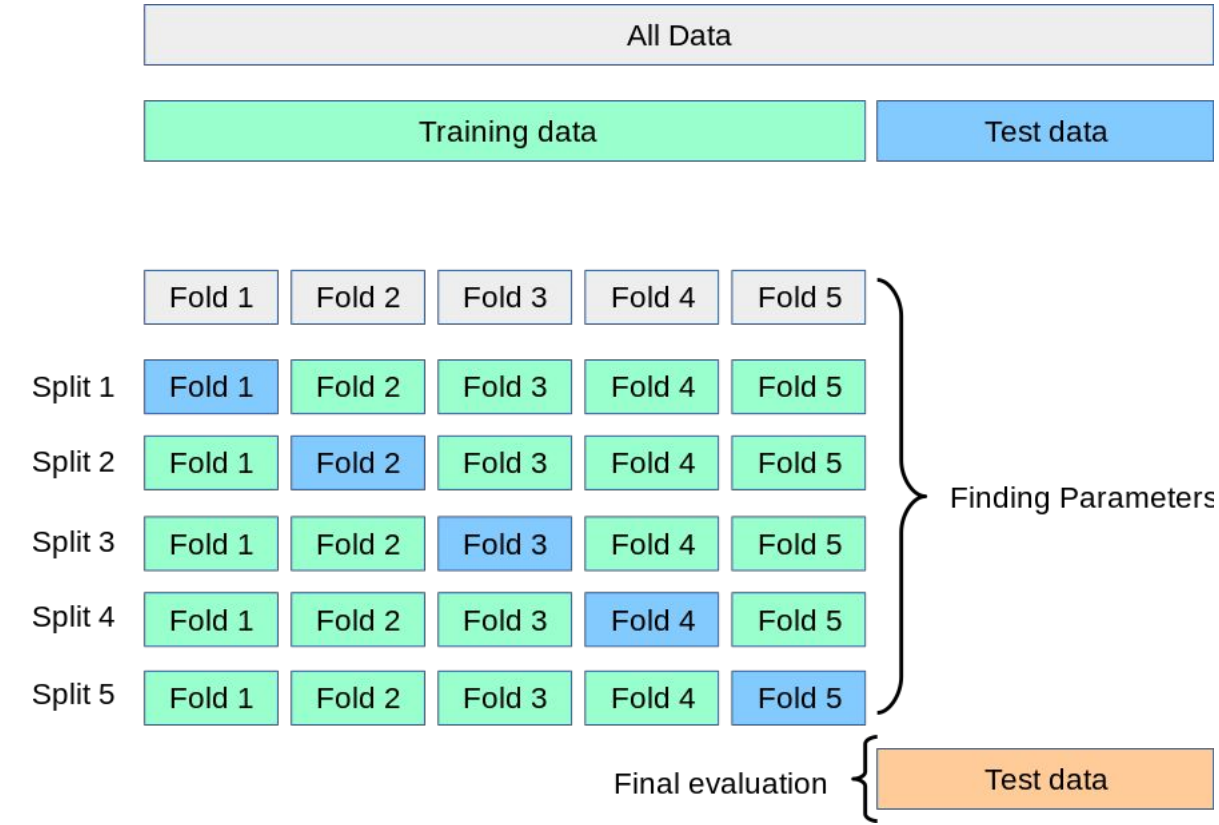
Hyper-parameter tuning

The presented neural networks contain several hyper-parameters (e.g., learning rate, λ, number of epochs…).

Choose the hyper-parameter combination that leads to the best performance! E.g., perform a **grid search.**



K-fold cross validation



We split data into ***K* subsets**, using each as a validation set in tandem with the remaining *K*-1 subsets for training.

We then evaluate the **average performance** for each combination of hyper parameters.

Finally, we train the model on the entire training set with the **optimal hyperparameters**.

Cf. scikit-learn.org

Metrics

Regression:

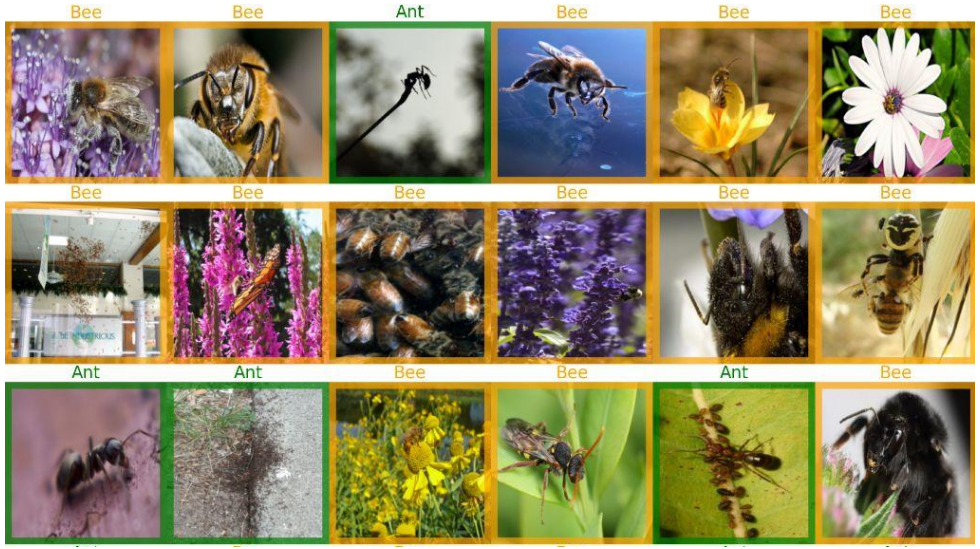
- Loss
- Coefficient of Determination (R^2)
- Mean square error
- Mean Absolute Error

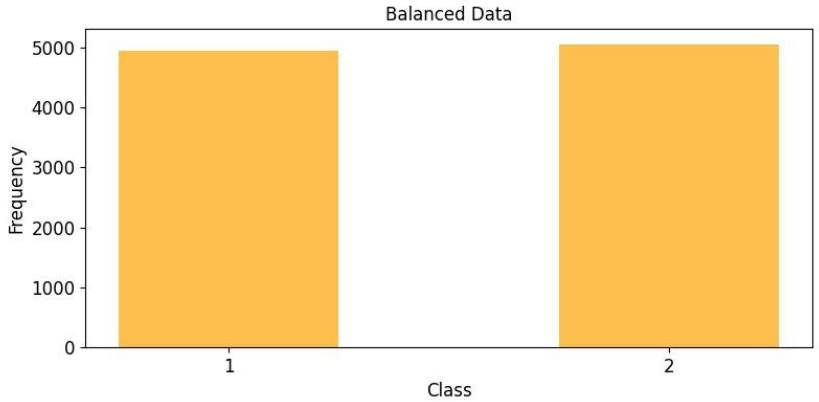
Classification:

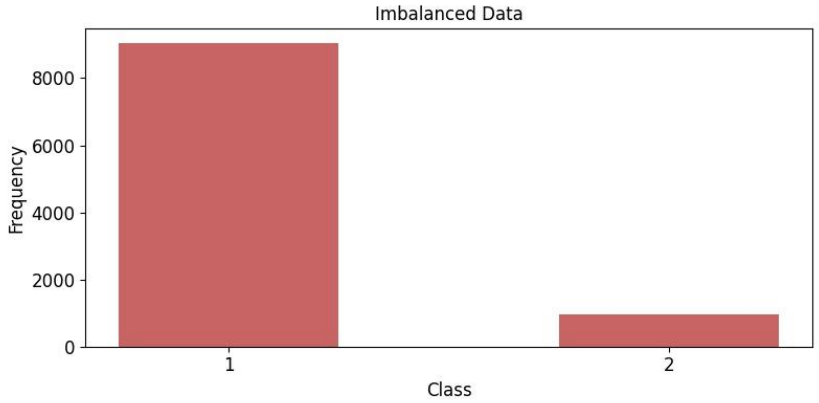
- Loss
- Accuracy and precision
- F1 score
- ROC curve (AUC)
- Recall (Sensitivity) and specificity

Balanced data

Imbalanced data can lead to biased model predictions, favoring majority classes and ignoring minority ones.







Representative data

Ensure that you split the data so that both the training and test data are representative.

Further optimisation

To further optimise your network, you could use, e.g.,

- Weight initialization
- Batch normalisation (normalise the output of each layer)
- Learning rate scheduling (adjust the learning rate during training)
- Momentum (include past gradients in gradient descent)
- Ensemble methods involve training multiple models and combine their predictions

New tools required?

Scikit learn (sklearn)

An open-source machine-learning library available for Python. Also offers, e.g.,

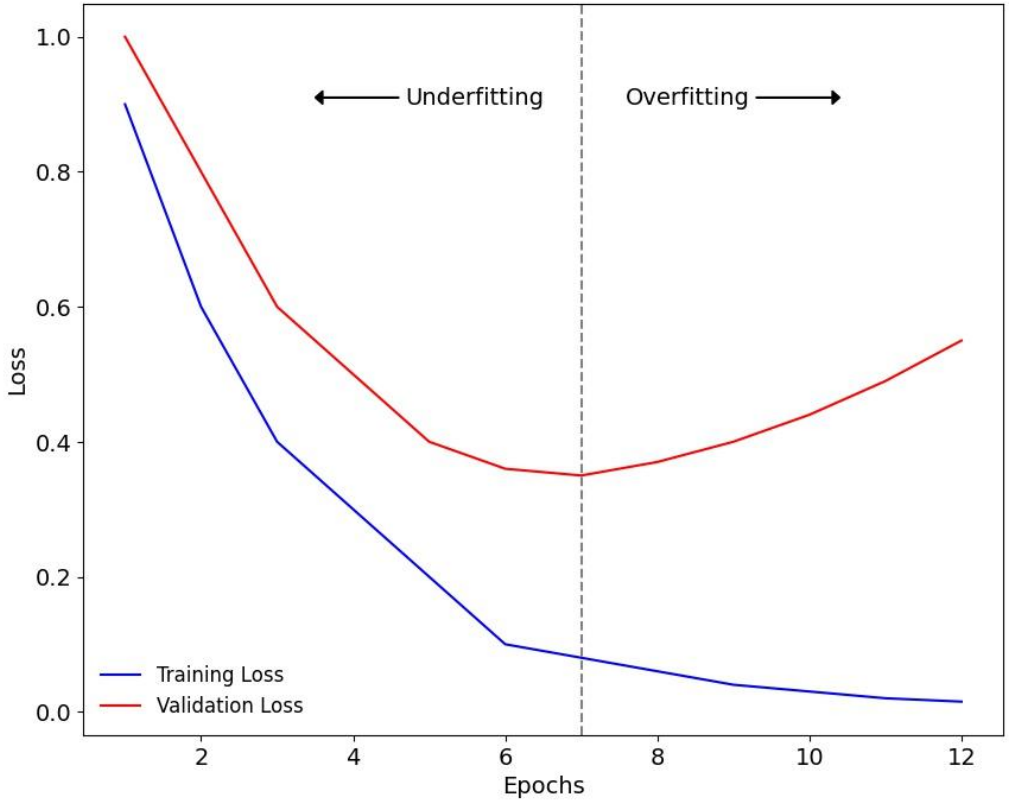
- Neural networks
- Nearest neighbours
- Gaussian processes
- Linear, Ridge, and Lasso Regression
- Logistic Regression
- Decision trees and random forests
- KMeans clustering
- Support vector machines

Also contains packages for model selection and evaluation.

Optimisation in practice

Splitting data

We need validation data to check for overfitting and for cross validation and it *cannot* be the test data…



Splitting data

We need validation data to check for overfitting and for cross validation and it *cannot* be the test data…

```
from torch.utils.data import random_split

# Load the MNIST dataset for training
mnist_dataset = datasets.MNIST(root='./data', train=True, transform=transform, download=True)

# Define the size of your validation set
validation_set_size = int(0.2 * len(mnist_dataset))  # Adjust the percentage as needed

# Split the dataset into training and validation sets
train_dataset, validation_dataset = random_split(mnist_dataset, [len(mnist_dataset) - validation_set_size, validation_set_size])

# Create DataLoader for training and validation sets
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
validation_loader = DataLoader(validation_dataset, batch_size=64, shuffle=False)
```

Including validation

```
for epoch in range(num_epochs):
        model.train()
        for inputs, labels in train_loader:
                inputs, labels = inputs.to(device), labels.to(device)
                optimizer.zero_grad()
                outputs = model(inputs)
                loss = criterion(outputs, labels)
                loss.backward()
                optimizer.step()
                training_loss += loss.item()

        model.eval()
        with torch.no_grad():
                for inputs, labels in validation_loader:
                        inputs, labels = inputs.to(device), labels.to(device)
                        outputs = model(inputs)
                        loss = criterion(outputs, labels)
                        val_loss += loss.item()
```

We need to evaluate the performance of the model on the validation set for every epoch.

Early stopping

Early stopping: Stop the training if validation loss stalls or worsens for a number of epochs (patience).

The implementations of other regularisation techniques are left as exercises.

```
# Early stopping
if avg_val_loss < best_val_loss:
        best_val_loss = avg_val_loss
        patience_counter = 0
else:
        patience_counter += 1
        if patience_counter >= patience:
                break
```

K-fold cross validation

```
from sklearn.model_selection import KFold
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)
```

K-fold cross validation

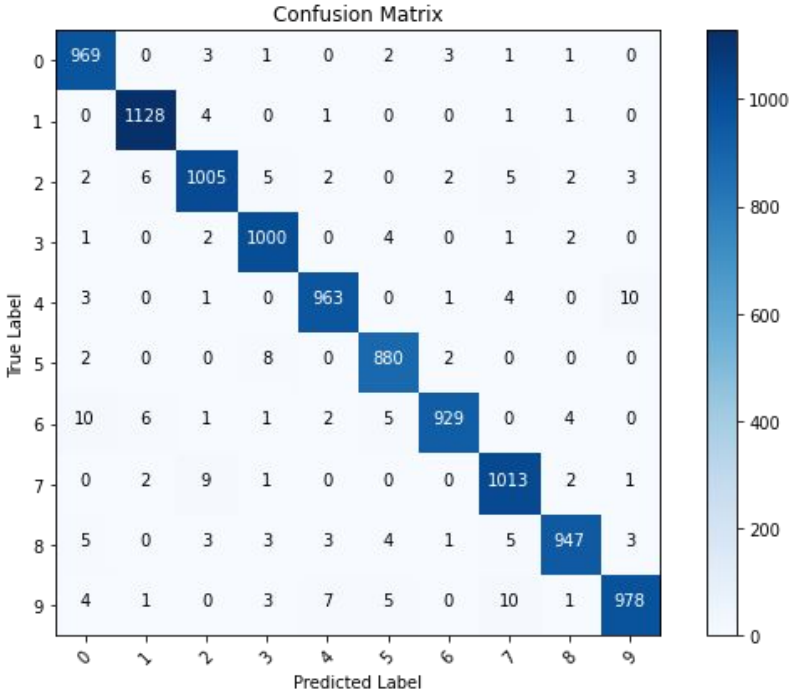
```
from sklearn.model_selection import KFold
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)

for lr in learning_rates:
    model = LinearRegressionModel()       # Initialize the model …
    optimizer = torch.optim.SGD(model.parameters(), lr=lr) # … and optimiser for each learning rate
```

K-fold cross validation

```
from sklearn.model_selection import KFold
kf = KFold(n_splits=num_folds, shuffle=True, random_state=42)

for lr in learning_rates:
    model = LinearRegressionModel()      # Initialize the model …
    optimizer = torch.optim.SGD(model.parameters(), lr=lr) # … and optimiser for each learning rate

    for train_indices, val_indices in kf.split(X_np):
        # For each iteration of the loop, we define training and validation sets
        X_train, y_train = X_np[train_indices], y_np[train_indices]
        train_dataset = TensorDataset(torch.tensor(X_train, dtype=torch.float32), torch.tensor(y_train, dtype=torch.float32))
        train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True)
        X_val, y_val = X_np[val_indices], y_np[val_indices]
        val_dataset = TensorDataset(torch.tensor(X_val, dtype=torch.float32), torch.tensor(y_val, dtype=torch.float32))
        val_loader = DataLoader(val_dataset, batch_size=16, shuffle=False)

        for epoch in range(num_epochs):
            …
```

Testing

```
from sklearn.metrics import confusion_matrix
```



In a binary classification task, the confusion matrix would include the true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN):

Actual class \ Predicted class	1	0
1	TP	FN
0	FP	TN

Beyond CNNs

Image generators

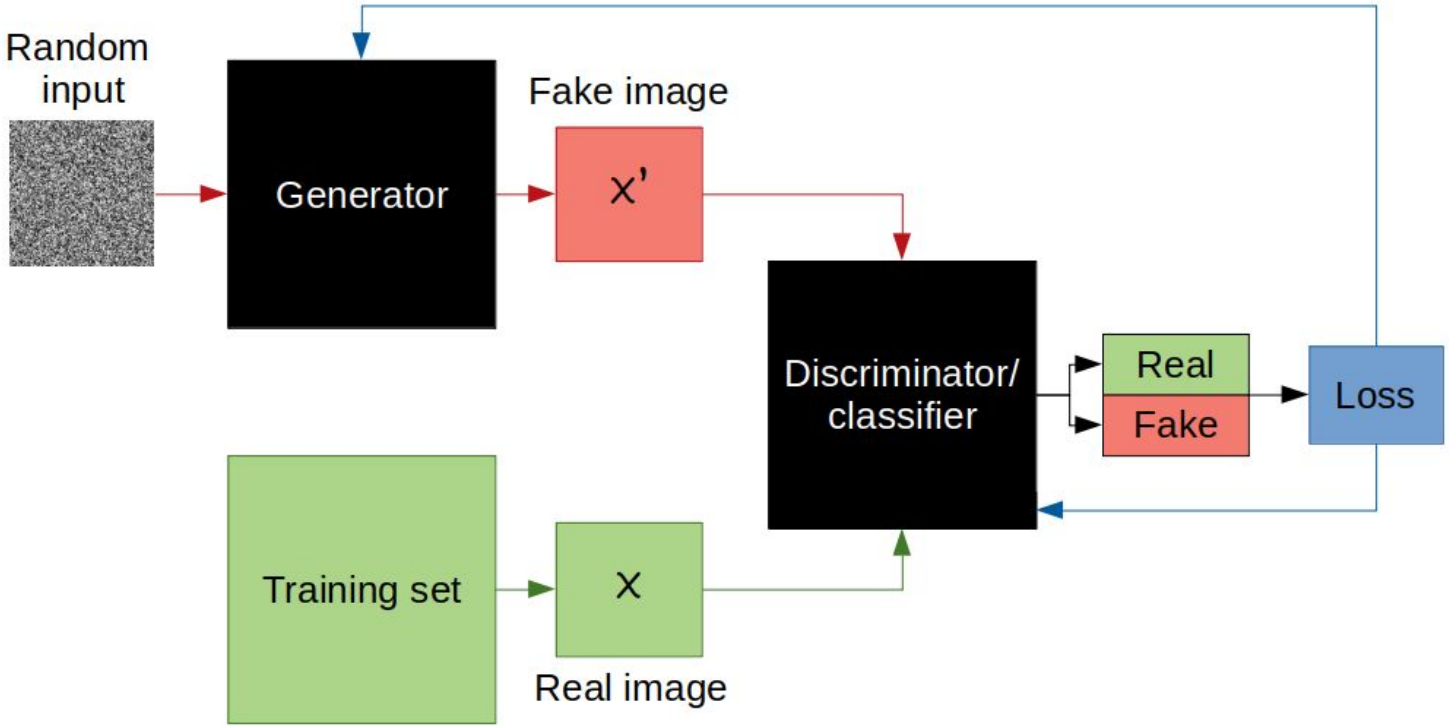
What if you wanted to generate images?

1) Generative adversarial networks (GANs)

2) Variational autoencoders (VAEs)

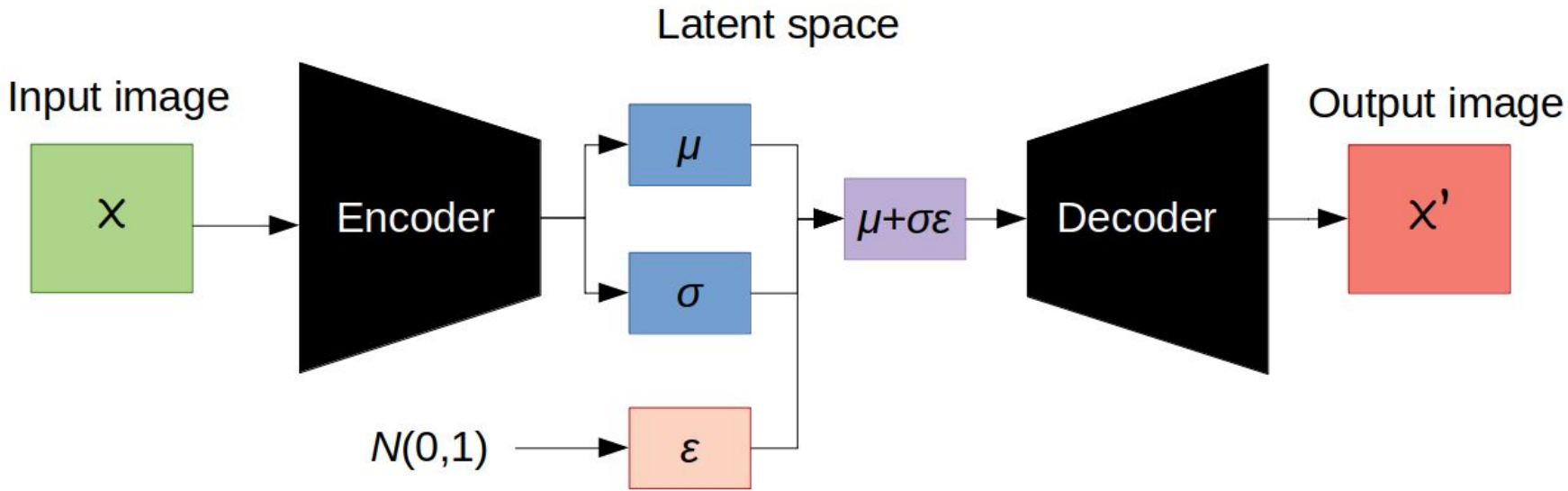
3) Flow-based generative models

GANs



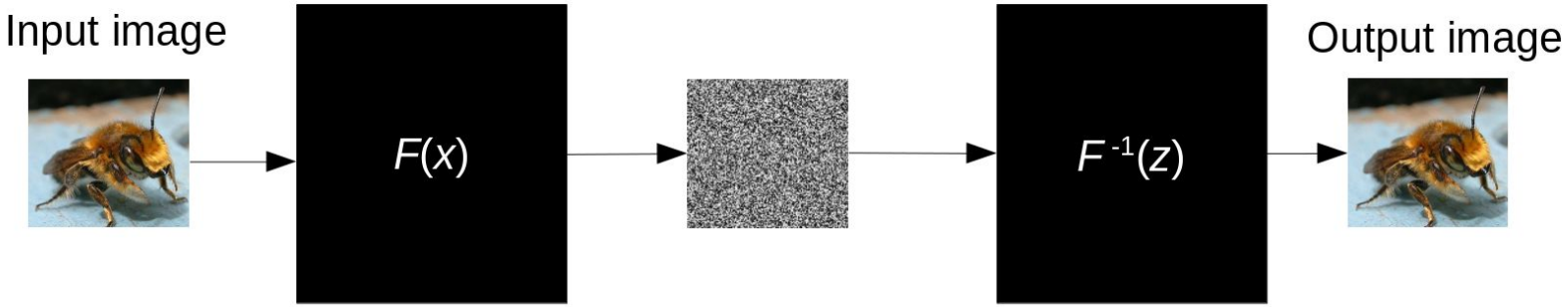
The generator and discriminator are adversaries.

VAEs



VAEs strive to create output images that match the input (minising the reconstruction loss) by maximising the ELBO.

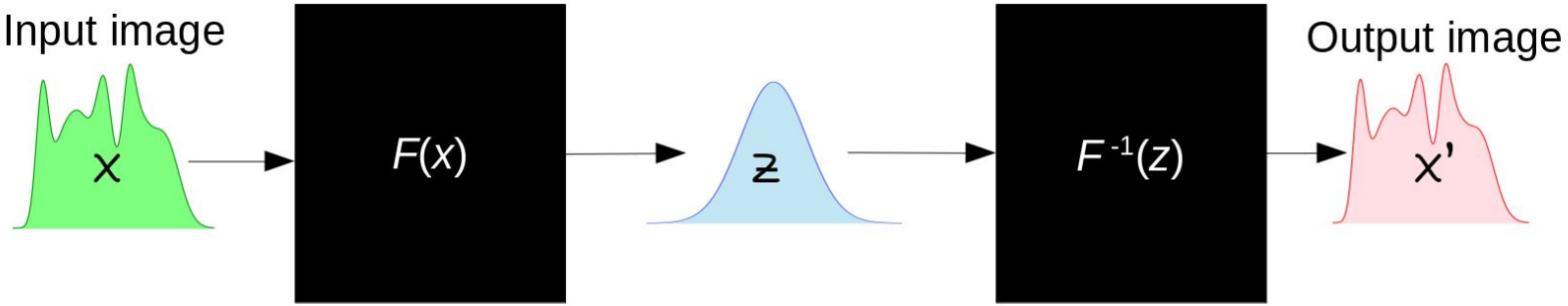
Flow-based generative models



The flow transforms the image (a complex distribution) to noise (a simple distribution) via an **invertible function** *F* in a stepwise manner:

$$z = F(x) = f_K(f_{K-1}(\dots f_2(f_1(x))\dots))$$

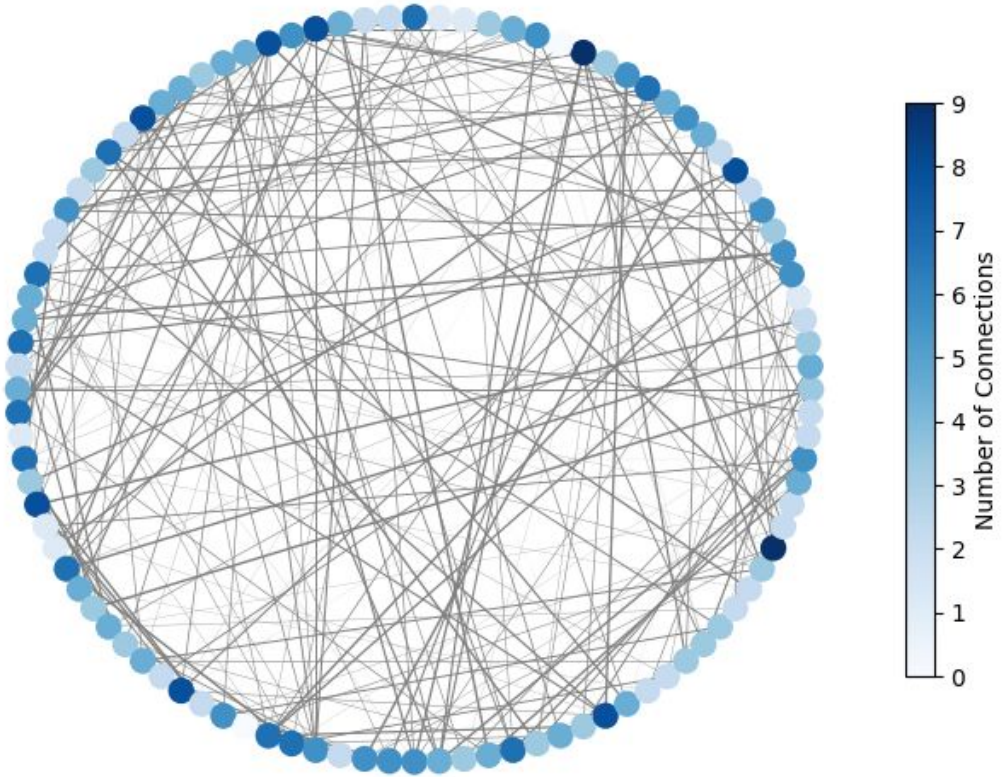
Flow-based generative models



The flow transforms the image (a complex distribution) to noise (a simple distribution) via an **invertible function** *F* in a stepwise manner:

$$z = F(x) = f_K(f_{K-1}(\dots f_2(f_1(x))\dots))$$

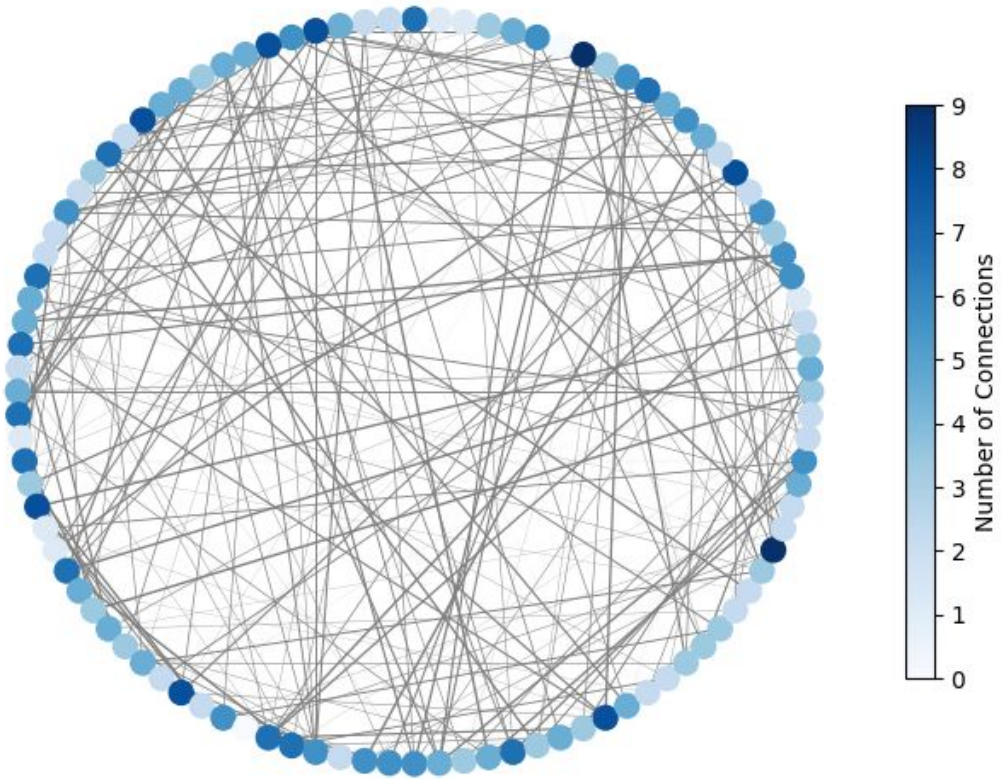
GNNs

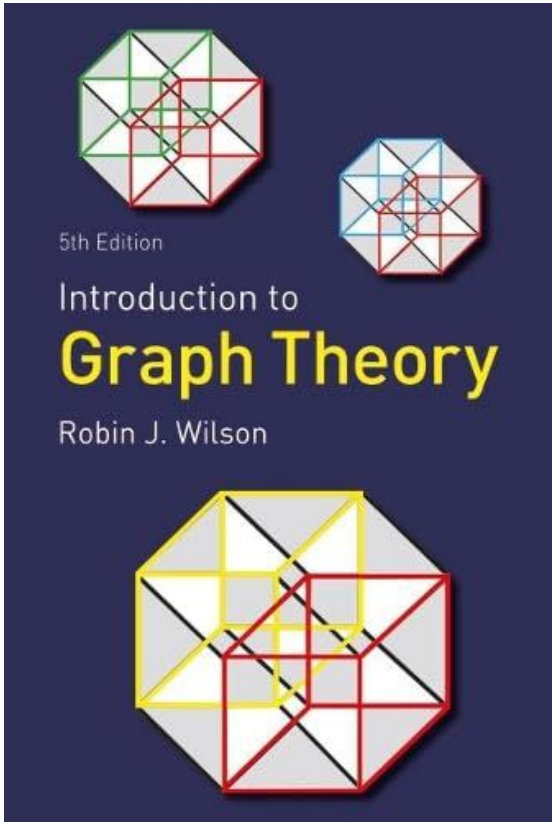


Some data can be represented as graphs (e.g. social networks). For these data, we can train a **graph neural network** (GNN).

We can use GNN to learn about patterns and relationships in the graph.

GNNs


9
8
7
6
5
4
3
2
1
0
Number of Connections

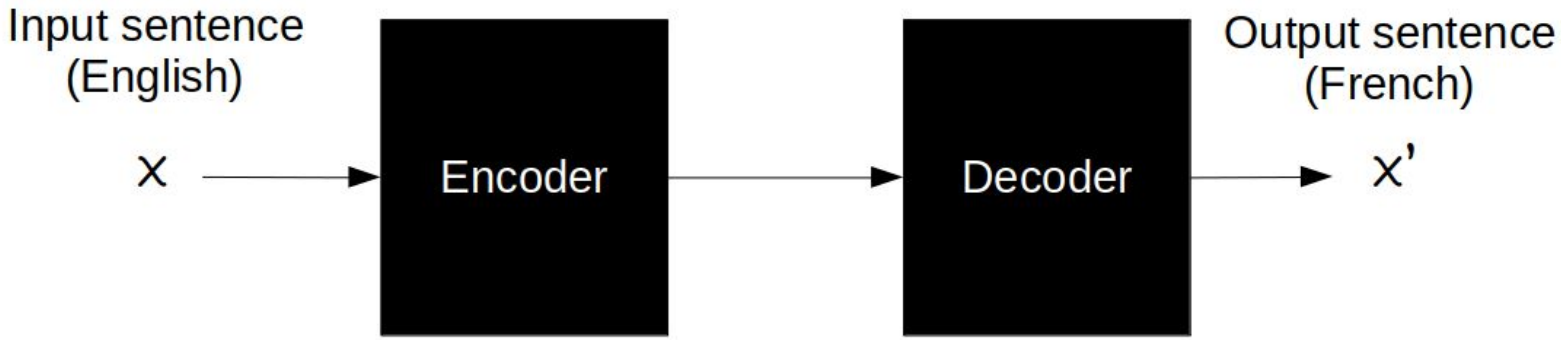

5th Edition
Introduction to
Graph Theory
Robin J. Wilson

NLP

Natural language processing, covers

- Text and speech processing (e.g., speech recognition).
- Lexical semantics (e.g., determine the meaning of a word in a context).
- Relational semantics (e.g., determine relationships between named entities).
- Morphological analysis (e.g., identify morphemes)
- Syntactic analysis (e.g., determine the grammar of a language)
- Discourse (e.g., identify topics of segments)

Transformers



Encoder: encodes words into numbers, the position of the words and the relations among words within the sentence (self-attention). The encoder and **decoder** are similar in structure. Encoder-decoder attention keeps track between input and output.

Transformers are **highly parallelizable**!

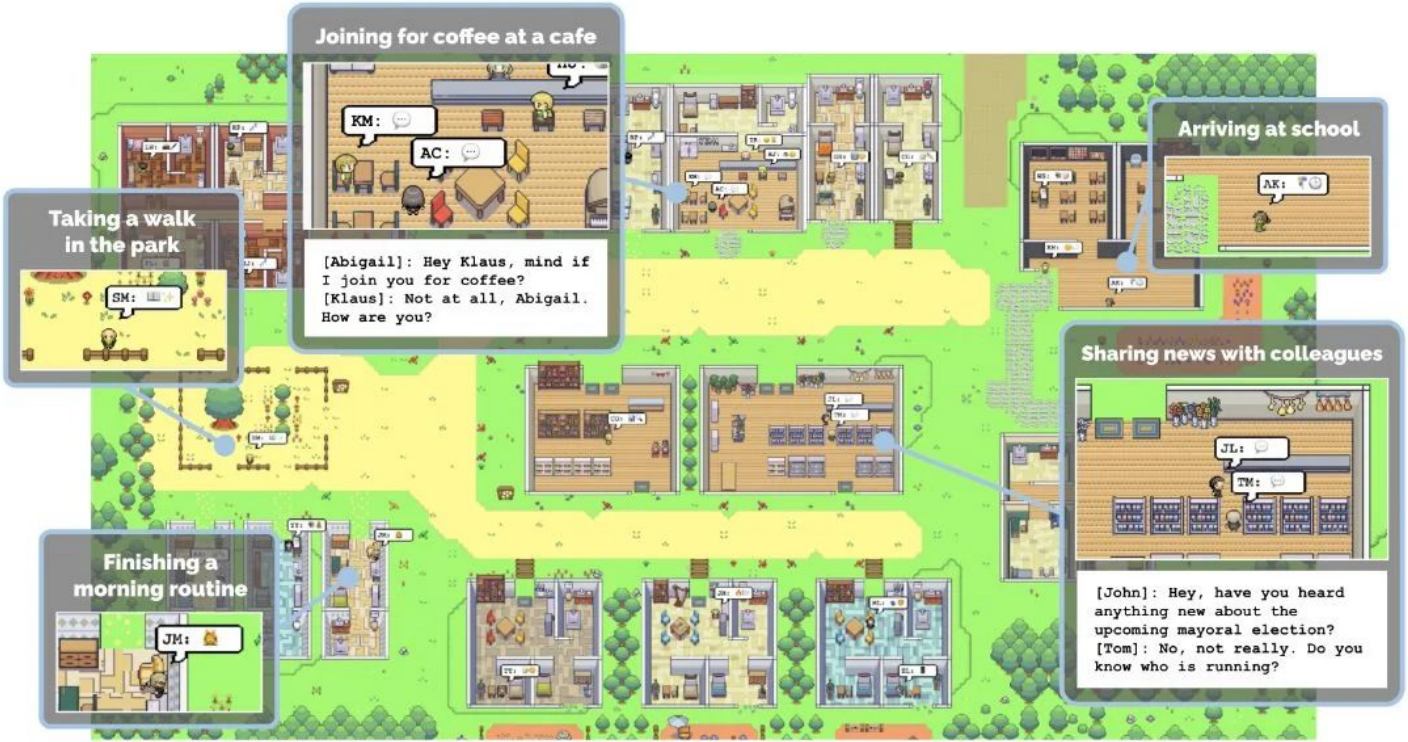
LLMs

Large Language Models:

BERT: Bidirectional Encoder Representations from *Transformers* (a encoder. 340 million parameters trained on 3.3 Billion words, e.g. from Wikipedia).

GPT: Generative Pre-trained *Transformer* (a decoder. GPT-3 has 175 million parameters and was trained on 45 TB of data). Using fine-tuning and reinforcement learning from human feedback, it can follow instructions.

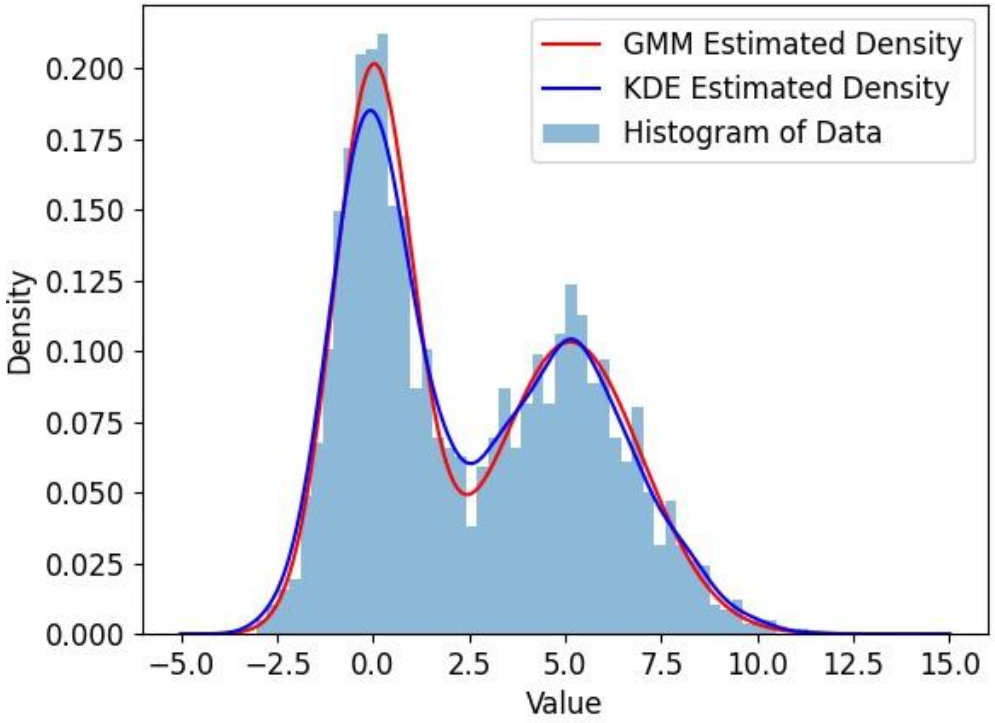
LLM



J. S. Park et al., 2023, arXiv

NDEs

A neural density estimator can be used to estimate an unobservable probability distribution based on samples.



Tensorflow

An open-source machine-learning library (by Google) available for Python.

Check out Tensorflow's **Keras** for building neural networks.

TensorFlow

Summary

Based on this lecture, you should better be able to

1) Evaluate the performance of your model and choose suitable metrics.

2) Explain regularisation strategies and apply the concept in Python.

3) Understand how to optimise the hyper-parameters of your model and do so in Python.

4) List key algorithms in Deep Learning beyond CNNs.