CSCI E-89B Introduction to Natural Language Processing

Harvard Extension School

Dmitry Kurochkin

Fall 2025
Lecture 11

Contents

Contents

Hidden Markov Models (HMMs)

- **Definition:** Hidden Markov Models (HMMs) are statistical models used to describe systems characterized by a sequence of observable events (observations) that are generated by underlying hidden states:
  - States: $\mathbf{Y} = \{Y_1, Y_2, \ldots, Y_T\}$, where each Y_t is a hidden state.
  - Observations: $\mathbf{X} = \{X_1, X_2, \ldots, X_T\}$, where each X_t is an observation.

  These hidden states undergo transitions over time according to specific probabilities:
  - Transition Probabilities: $P(Y_t = j \mid Y_{t-1} = i)$
  - Emission Probabilities: $P(X_t \mid Y_t = j)$
  - Initial State Probabilities: $P(Y_1 = i)$ at time $t = 1$

- **Algorithms:**
  - *Viterbi Algorithm:* A dynamic programming algorithm used to efficiently find the most likely sequence of hidden states (the Viterbi path) given the observed data.
  - *Baum-Welch Algorithm:* An iterative algorithm based on the Expectation-Maximization (EM) framework. It is used to estimate the unknown parameters of an HMM, adapting the model to better fit

Applications and Disadvantages of HMMs

Applications:

- *Speech Recognition:* Transforms spoken language into text by analyzing and modeling speech patterns.
- *Part-of-Speech (POS) Tagging:* Labels each word in a sentence with its part of speech such as noun, verb, or adjective.
- *Bioinformatics:* Analyze biological sequences, such as DNA and proteins.

Disadvantages:

- Simplified Dependencies: The assumptions in HMMs, such as the Markov Property, restrict the model's ability to capture complex, long-range dependencies across states and observations. This simplification can limit the expressiveness of the model when applied to sequences that involve intricate patterns and dependencies.
- Scalability Challenges: High computational costs with large state spaces and sequences due to exponential growth in processing and memory requirements, limiting their efficiency in datasets.

Contents

Conditional Random Fields (CRFs)

- **Definition:** Conditional Random Fields (CRFs) are discriminative probabilistic models used for structured prediction. They model the conditional probability of a label sequence $\mathbf{Y}$ given an observation sequence $\mathbf{X}$, expressed as:

$$P(\mathbf{Y} \mid \mathbf{X}) = \frac{1}{Z(\mathbf{X})} \exp\left(\sum_{t=1}^{T} \sum_{k} \lambda_k f_k(Y_{t-1}, Y_t, \mathbf{X}, t)\right)$$

where $Z(\mathbf{X})$ is the normalization factor calculated over all possible label sequences $\mathbf{Y}$, ensuring that the total probability sums to one given the observation sequence $\mathbf{X}$. The observed sequence $\mathbf{X}$ represents the input features, and $\mathbf{Y}$ is the sequence of labels or states predicted by the CRF.

Feature functions $f_k(\cdot)$ are manually designed by experts. These features are crafted from the observations to capture intricate dependencies relevant to the task.

Applications of CRFs

- *Named Entity Recognition (NER):* CRFs are effective in identifying and classifying entities such as person names, organizations, and locations within text due to their ability to incorporate contextual information from surrounding words, improving precision and recall in entity extraction tasks.
- *Part-of-Speech Tagging:* By leveraging the surrounding words and syntactic context, CRFs enhance the tagging accuracy of parts of speech, allowing for more nuanced linguistic analysis, particularly useful in languages where word meaning can change with context.
- *Segmentation Tasks in Image Processing:* In computer vision, CRFs are applied to label and segment different regions of an image, using pixel-based features and inter-pixel relationships to produce coherent segmentations, crucial for tasks like object detection and scene understanding.

Advantages of CRFs

- **Flexible Feature Integration:** CRFs can incorporate a wide range of features, including overlapping and interdependent ones, allowing for detailed contextual modeling that improves performance in intricate tasks.
- **Structured Dependency Modeling:** CRFs effectively model dependencies between output labels, supporting tasks where the structure of the output sequence is crucial, such as in sequence labeling applications.
- **No Independence Assumptions Between Observations:** Unlike some traditional models, CRFs don't assume independence between input observations, enabling more nuanced analysis by capturing complex relationships across larger contexts.

Disadvantages of CRFs

- **High Computational Demand:** CRFs require considerable computational resources during training due to the complex dependencies modeled, particularly in scenarios involving large datasets or extensive feature spaces. This can lead to slower training times compared to simpler models.
- **Intensive Feature Engineering:** Success with CRFs heavily relies on the construction of comprehensive and relevant feature sets. Crafting these features necessitates deep domain expertise and can involve extensive experimentation, which is both time-consuming and challenging.

Contents

Bidirectional LSTM with CRF (BiLSTM-CRF)

- **Definition:** BiLSTM-CRF is a hybrid model that integrates Bidirectional Long Short-Term Memory networks (BiLSTMs) with Conditional Random Fields (CRFs). This model is particularly effective for sequence labeling tasks.

  Architecture:

  - BiLSTMs capture sequential dependencies from both past (left) and future (right) contexts, producing rich feature representations for each element in the sequence.
  - CRF layers capture and model how output labels depend on each other in a sequence. This helps produce label sequences that are coherent and consistent across the entire input, optimizing the overall structure rather than making independent, disconnected label predictions.

Contents

Applications of BiLSTM-CRF

- *Named Entity Recognition (NER):* BiLSTM-CRF excels in identifying entities within text by leveraging bidirectional context and structured label dependencies.
- *Part-of-Speech (POS) Tagging:* Utilizes bidirectional LSTM features to improve tagging accuracy by considering both previous and next word contexts.
- *Semantic Role Labeling:* Assigns semantic roles to phrases within sentences, benefiting from context-rich BiLSTM features combined with CRF constraints for coherent labeling.

Contents

Advantages of BiLSTM-CRF

- **Comprehensive Contextual Understanding:** BiLSTM-CRF models capture dependencies from both past and future contexts, enriching feature representations.
- **Automatic Feature Learning:** Embedding and LSTM layers autonomously learn complex features from data, reducing the need for extensive manual feature engineering while adapting to intricate data patterns.
- **Structured Output Consistency:** CRF layers ensure the output labels form coherent sequences, effectively leveraging context to improve accuracy.
- **Versatility in NLP Tasks:** Suitable for a wide range of sequence-based NLP tasks, enhancing results where context and sequence structure are essential.

Disadvantages of BiLSTM-CRF

- **High Computational Cost:** The integration of BiLSTM and CRF requires significant computational resources, especially for large datasets or complex problems.
- **Complex Model Tuning:** Fine-tuning this hybrid model involves managing intricate hyperparameters, demanding expertise and experimentation.
- **Training Time:** Training BiLSTM-CRF models can be time-intensive, requiring careful optimization strategies to balance performance and efficiency.

BiLSTM-CRF in Python

```
!pip install torch
!pip install pytorch-crf

import torch
import torch.nn as nn
from TorchCRF import CRF


class BiLSTM_CRF(nn.Module):
    def __init__(self, vocab_size, tagset_size, embedding_dim=100, hidden_dim=256):
        super(BiLSTM_CRF, self).__init__()

        # Embedding layer to convert word indices to embeddings
        self.embedding = nn.Embedding(vocab_size, embedding_dim)

        # BiLSTM layer to process sequences in both directions
        self.lstm = nn.LSTM(embedding_dim, hidden_dim // 2, num_layers=1,
                            bidirectional=True, batch_first=True)

        # Linear layer to map LSTM outputs to tag scores
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)

        # CRF layer to model tag dependencies
        self.crf = CRF(tagset_size, batch_first=True)


    def forward(self, sentences, tags=None, mask=None):
        # Transform input word indices to embeddings
        embeddings = self.embedding(sentences)
```

BiLSTM-CRF in Python (Continued)

```
        # Pass embeddings through the BiLSTM
        lstm_out, _ = self.lstm(embeddings)

        # Convert BiLSTM output to emission scores for each tag
        emissions = self.hidden2tag(lstm_out)

        if tags is not None:
            # Calculate negative log likelihood for CRF training
            log_likelihood = self.crf(emissions, tags, mask=mask)
            return -log_likelihood
        else:
            # Decode the best tag sequence using CRF during inference
            tag_seq = self.crf.decode(emissions, mask=mask)
            return tag_seq
```

BiLSTM-CRF Model in Python (Continued)

```
# Example setup
vocab_size = 5000
tagset_size = 9
model = BiLSTM_CRF(vocab_size, tagset_size)

# Optimizer setup
optimizer = torch.optim.SGD(model.parameters(), lr=0.01, momentum=0.9)

# Dummy dataset
sentences = torch.tensor([[1, 2, 3, 0], [4, 5, 6, 7]])
tags = torch.tensor([[0, 1, 2, 0], [1, 2, 3, 4]])
mask = sentences.gt(0)

# Training loop
num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    loss = model(sentences, tags, mask=mask)
    loss.backward()
    optimizer.step()
    print(f'Epoch {epoch + 1}/{num_epochs} - Loss: {loss.item()}')
```

```
Epoch 1/5 - Loss: 15.381282806396484
Epoch 2/5 - Loss: 14.884138107299805
Epoch 3/5 - Loss: 13.962076187133789
Epoch 4/5 - Loss: 12.697538375854492
Epoch 5/5 - Loss: 11.17860221862793
```

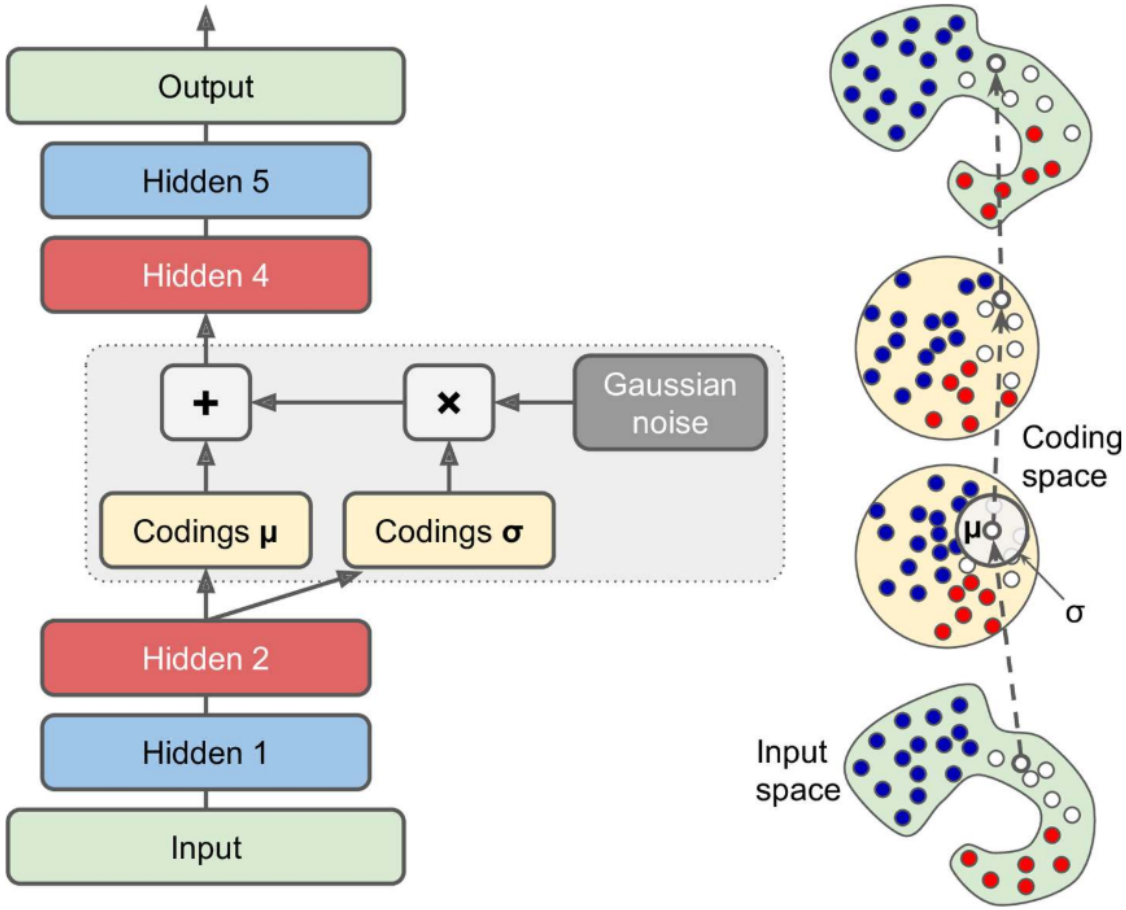
BiLSTM-CRF Model in Python (Continued)

```
# Prediction
model.eval()  # Switch to evaluation mode
predicted_tags = model(sentences, mask=mask)
print("Predicted Tags:", predicted_tags)

Predicted Tags: [[0, 1, 2], [1, 2, 3, 4]]
```

Contents

Variational Autoencoders



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

Variational Autoencoders: Latent Loss

The loss function is defined as reconstruction loss (for example, cross-entropy based on inputs and reconstructions) plus the following *latent loss*:

$$\mathscr{L} = -\frac{1}{2}\sum_{i=1}^{n}\left[1 + \underbrace{\ln \sigma_i^2}_{\gamma_i^2} - \underbrace{\sigma_i^2}_{e^{\gamma_i^2}} - \mu_i^2\right],$$

where μ_i and σ_i^2 denote mean and variance of i-th coding, respectively.

Here, n is the total number of codings.

Remark: $\mathscr{L} = D_{\mathsf{KL}}\left(\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2) \,\|\, \mathcal{N}(\mathbf{0}, \mathbf{1})\right)$

Contents

Generative Adversarial Networks (GANs)

Generative Adversarial Nets

Ian J. Goodfellow, Jean Pouget-Abadie*, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair†, Aaron Courville, Yoshua Bengio‡
Département d'informatique et de recherche opérationnelle
Université de Montréal
Montréal, QC H3C 3J7

Abstract

We propose a new framework for estimating generative models via an adversarial process, in which we simultaneously train two models: a generative model G that captures the data distribution, and a discriminative model D that estimates the probability that a sample came from the training data rather than G. The training procedure for G is to maximize the probability of D making a mistake. This framework corresponds to a minimax two-player game. In the space of arbitrary functions G and D, a unique solution exists, with G recovering the training data distribution and D equal to $\frac{1}{2}$ everywhere. In the case where G and D are defined by multilayer perceptrons, the entire system can be trained with backpropagation. There is no need for any Markov chains or unrolled approximate inference networks during either training or generation of samples. Experiments demonstrate the potential of the framework through qualitative and quantitative evaluation of the generated samples.

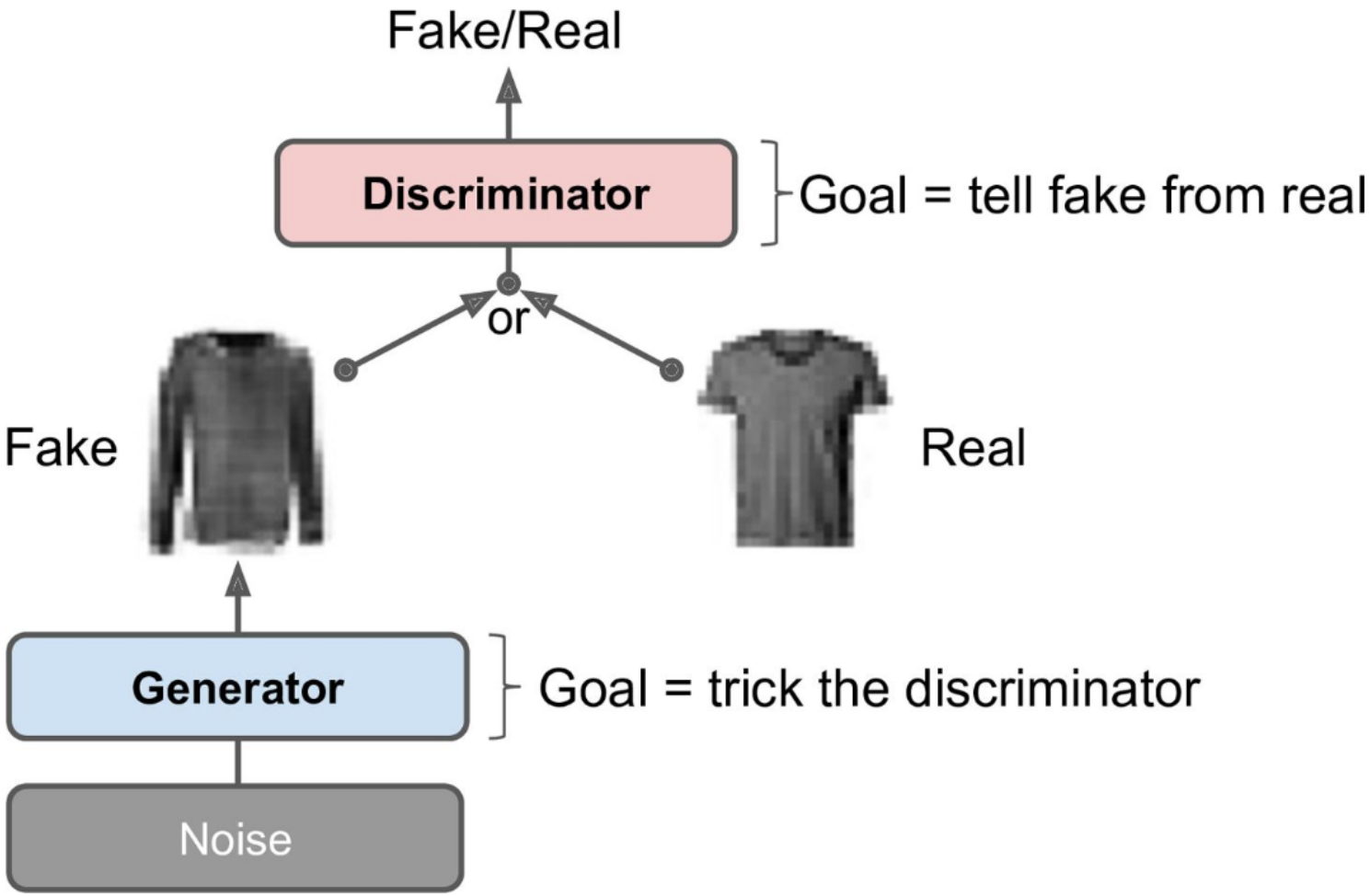
Source: *Generative Adversarial Networks*, 2014 by Ian J. Goodfellow et al.

Generative Adversarial Networks (GANs)

References:

- NIPS 2016 Tutorial: Generative Adversarial Networks, Ian Goodfellow, 2017, `https://arxiv.org/abs/1701.00160`
- Chapter 8, Deep Learning with Python by François Chollet, O'Reilly 2017, and Jupyter notebooks associated with that chapter
- Image Style Transfer Using CNNs, Leon A. Gatys, Alexander S. Ecker and Matthias Bethge, 2015, IEEE Xplore, `https://arxiv.org/abs/1508.06576`
- Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks, Alec Radford, Luke Metz, Soumith Cintala, 2016, `https://arxiv.org/pdf/1511.06434.pdf`

Generative Adversarial Networks (GANs)



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

Generative Adversarial Networks (GANs)

```
codings_size = 30

generator = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[codings_size]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])

discriminator = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(150, activation="selu"),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Generative Adversarial Networks (GANs)

```
discriminator.compile(loss="binary_crossentropy", optimizer="rmsprop")
discriminator.trainable = False
gan.compile(loss="binary_crossentropy", optimizer="rmsprop")
```

```
batch_size = 32
dataset = tf.data.Dataset.from_tensor_slices(X_train).shuffle(1000)
dataset = dataset.batch(batch_size, drop_remainder=True).prefetch(1)
```

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Generative Adversarial Networks (GANs)

```
def train_gan(gan, dataset, batch_size, codings_size, n_epochs=50):
    generator, discriminator = gan.layers
    for epoch in range(n_epochs):
        for X_batch in dataset:
            # phase 1 - training the discriminator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            generated_images = generator(noise)
            X_fake_and_real = tf.concat([generated_images, X_batch], axis=0)
            y1 = tf.constant([[0.]] * batch_size + [[1.]] * batch_size)
            discriminator.trainable = True
            discriminator.train_on_batch(X_fake_and_real, y1)
            # phase 2 - training the generator
            noise = tf.random.normal(shape=[batch_size, codings_size])
            y2 = tf.constant([[1.]] * batch_size)
            discriminator.trainable = False
            gan.train_on_batch(noise, y2)

train_gan(gan, dataset, batch_size, codings_size)
```

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Generative Adversarial Networks (GANs)

Generated images after 1 epoch of training:

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

Generative Adversarial Networks (GANs)

```
codings_size = 100

generator = keras.models.Sequential([
    keras.layers.Dense(7 * 7 * 128, input_shape=[codings_size]),
    keras.layers.Reshape([7, 7, 128]),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(64, kernel_size=5, strides=2, padding="same",
                                 activation="selu"),
    keras.layers.BatchNormalization(),
    keras.layers.Conv2DTranspose(1, kernel_size=5, strides=2, padding="same",
                                 activation="tanh")
])
```

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Generative Adversarial Networks (GANs)

```
discriminator = keras.models.Sequential([
    keras.layers.Conv2D(64, kernel_size=5, strides=2, padding="same",
                        activation=keras.layers.LeakyReLU(0.2),
                        input_shape=[28, 28, 1]),
    keras.layers.Dropout(0.4),
    keras.layers.Conv2D(128, kernel_size=5, strides=2, padding="same",
                        activation=keras.layers.LeakyReLU(0.2)),
    keras.layers.Dropout(0.4),
    keras.layers.Flatten(),
    keras.layers.Dense(1, activation="sigmoid")
])
gan = keras.models.Sequential([generator, discriminator])
```

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

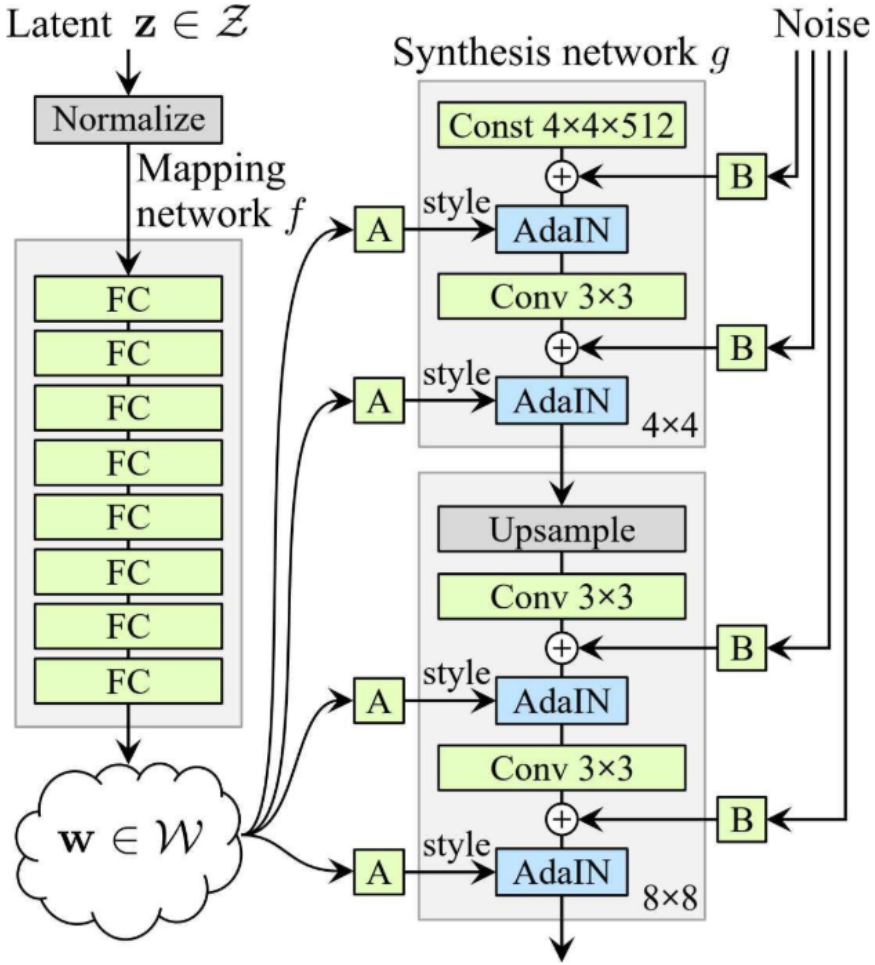
Generative Adversarial Networks (GANs)

Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

Contents

StyleGANs

Generator:



Source: *Hands-On Machine Learning with Scikit-Learn and TensorFlow* by A. Géron

StyleGANs

This cat does not exist:

Source: `https://thiscatdoesnotexist.com`

StyleGANs

This person does not exist:

Source: `https://thispersondoesnotexist.com`