CSCI E-89B Introduction to Natural Language Processing

Harvard Extension School

Dmitry Kurochkin

Fall 2024
Lecture 6

Contents

Contents

Introduction to Character Embeddings

- **Character Embeddings:**
  - Each character is encoded into a vector within a continuous vector space, allowing the representation of words based on their individual characters.
  - This approach helps the model learn orthographic and morphological characteristics of a language, which are essential in tasks like spelling correction and morphological analysis.
- **Capturing Sub-word Information:**
  - By focusing on characters, sub-word patterns such as prefixes, suffixes, and stems are captured, aiding in modeling complex word formations.
  - This technique is especially beneficial for languages with rich morphology, supporting the model in recognizing semantic similarities across related word forms, such as "running," "runner," and "ran."
- **Benefits:**
  - Enhances the handling of out-of-vocabulary words and misspellings by deconstructing words into familiar components.
  - Increases robustness and adaptability of NLP systems across various languages and text types.
  - Handles morphological variations, prefixes, and suffixes.

Applications of Character Embeddings

- **Effective in Languages with Rich Inflectional Morphology:**
  - Crucial for processing languages like Finnish and Turkish, where words undergo numerous inflectional modifications with a wide array of morphological variations for tense, case, number, gender, and more.
  - By capturing sub-word patterns, character embeddings allow models to generalize across different word forms from the same root, minimizing vocabulary size and handling out-of-vocabulary issues effectively.
  - For example, in Finnish, which employs 15 grammatical cases, character embeddings enable models to identify the root and distinction among the varieties, improving text processing accuracy.
- **Use in Named Entity Recognition (NER):**
  - Enhance NER by helping models identify named entities with varying morphological or orthographic forms, including handling typos, casing, and inflections. This includes handling typos, different casing, and inflected forms of names.
  - Because NER often involves recognizing words not seen during training, character embeddings provide a robust mechanism for identifying new entity forms reliably by learning about character-level patterns and structures.

Contents

Methods for Character Embeddings

- **Character-Level RNNs/CNNs:**
  - Use Recurrent Neural Networks (RNNs) or Convolutional Neural Networks (CNNs) to process character sequences, providing a dynamic way to capture linguistic nuances.
  - **RNNs:** Ideal for capturing sequential dependencies in text by processing characters one at a time and maintaining a memory of previous inputs, thus effectively modeling the context across varying lengths of text.
  - **CNNs:** Good at identifying local patterns through filters that slide across character sequences, capturing morphological features like prefixes, suffixes, and recurring motifs efficiently by recognizing patterns regardless of their location in the input.
  - These methods excel at tasks needing detailed granularity in text processing, such as text generation or complex morphological languages where understanding of sub-word units is essential.

Methods for Character Embeddings (Continued)

- **Embedding Layer:**
  - Converts characters into dense vectors, reducing vocabularies to manageable sizes for models.
  - Acts as a foundational layer that allows deep learning models to efficiently process and understand the nuances of language by representing even the smallest language units.
  - This layer facilitates the integration of character-level data into broader neural network architectures, offering flexibility and adaptability for various NLP tasks.
- **Hybrid Approaches:**
  - Combine character embeddings with word or subword embeddings like BPE (Byte Pair Encoding) or WordPiece to achieve comprehensive text representation, balancing fine-grained detail with overall semantic context.
  - Balance detail from characters with semantic richness from larger units.
  - Enhance model performance by providing layered understanding, crucial for tasks like machine translation and sentiment analysis.

Contents

Hands-On: Character Embeddings in Python

```
import numpy as np
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, Dense

# Sample text data
texts = ["hello world", "machine learning", "deep learning"]
labels = np.array([1, 0, 0])  # Convert labels to a NumPy array

# Tokenization
tokenizer = Tokenizer(char_level=True)
tokenizer.fit_on_texts(texts)
sequences = tokenizer.texts_to_sequences(texts)

# Character vocabulary
max_vocab_size = len(tokenizer.word_index) + 1
print("Character Index:", tokenizer.word_index)

# Padding sequences
max_sequence_length = max(len(seq) for seq in sequences)
padded_sequences = pad_sequences(sequences, maxlen=max_sequence_length, padding='post')

# Convert padded sequences to a NumPy array
padded_sequences = np.array(padded_sequences)
```

Hands-On: Character Embeddings in Python (Continued)

```
# Model architecture
embedding_dim = 8
model = Sequential([
    Embedding(input_dim=max_vocab_size,
              output_dim=embedding_dim,
              input_length=max_sequence_length),
    LSTM(64),
    Dense(1, activation='sigmoid')
])

# Compile model
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])
model.summary()

# Train model
model.fit(padded_sequences, labels, epochs=10, batch_size=2)

# Evaluate model
loss, accuracy = model.evaluate(padded_sequences, labels)
print(f"Loss: {loss}, Accuracy: {accuracy}")

# Character Index: {'e': 1, 'l': 2, 'n': 3, ' ': 4, 'r': 5, 'a': 6, 'i': 7,
# 'h': 8, 'o': 9, 'd': 10, 'g': 11, 'w': 12, 'm': 13, 'c': 14, 'p': 15}
```

Contents

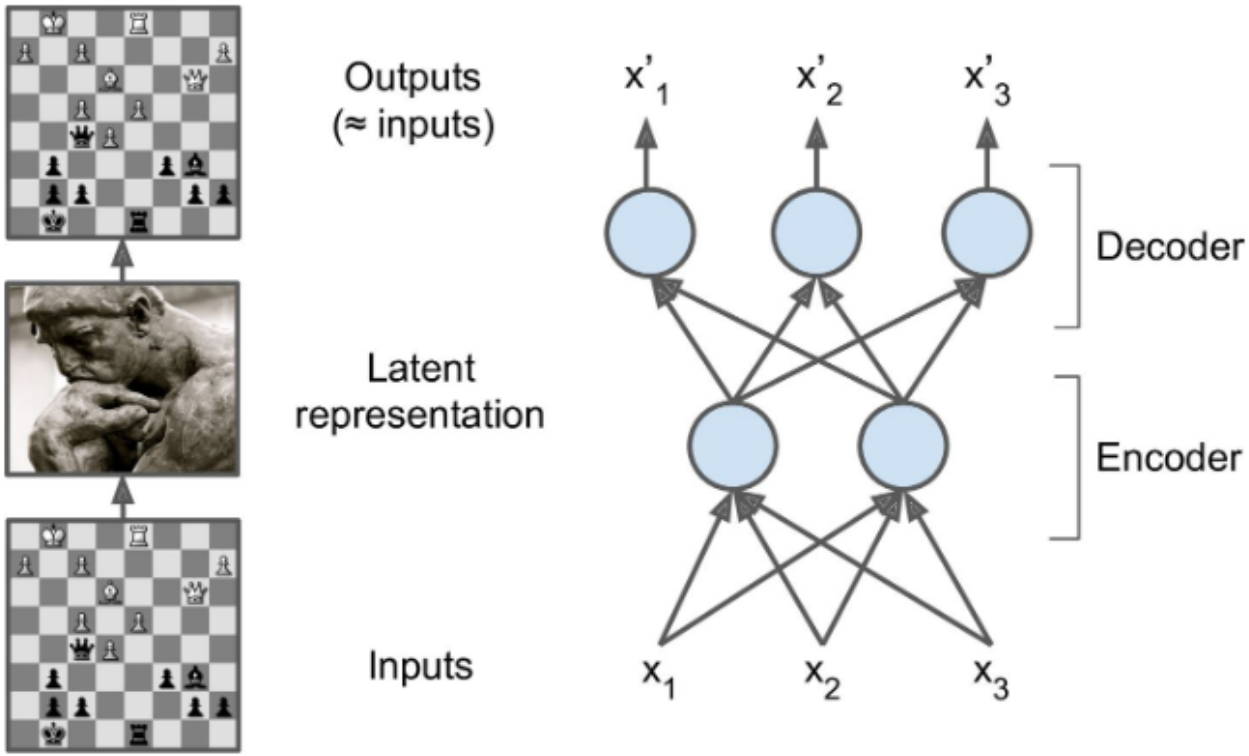
Autoencoders

Example:

- Sequence 1:
  $5, 7, 3, 5, 8, 9, 67, 5, 3, 12, 3, 5, 6$
- Sequence 2:
  $70, 68, 66, 64, 62, 60, 58, 56, 54, 52, 50, 48, 46, 44, 42, 40, 38, 36, 34, 32, 30, 28$

Autoencoders

Example:



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

Autoencoders

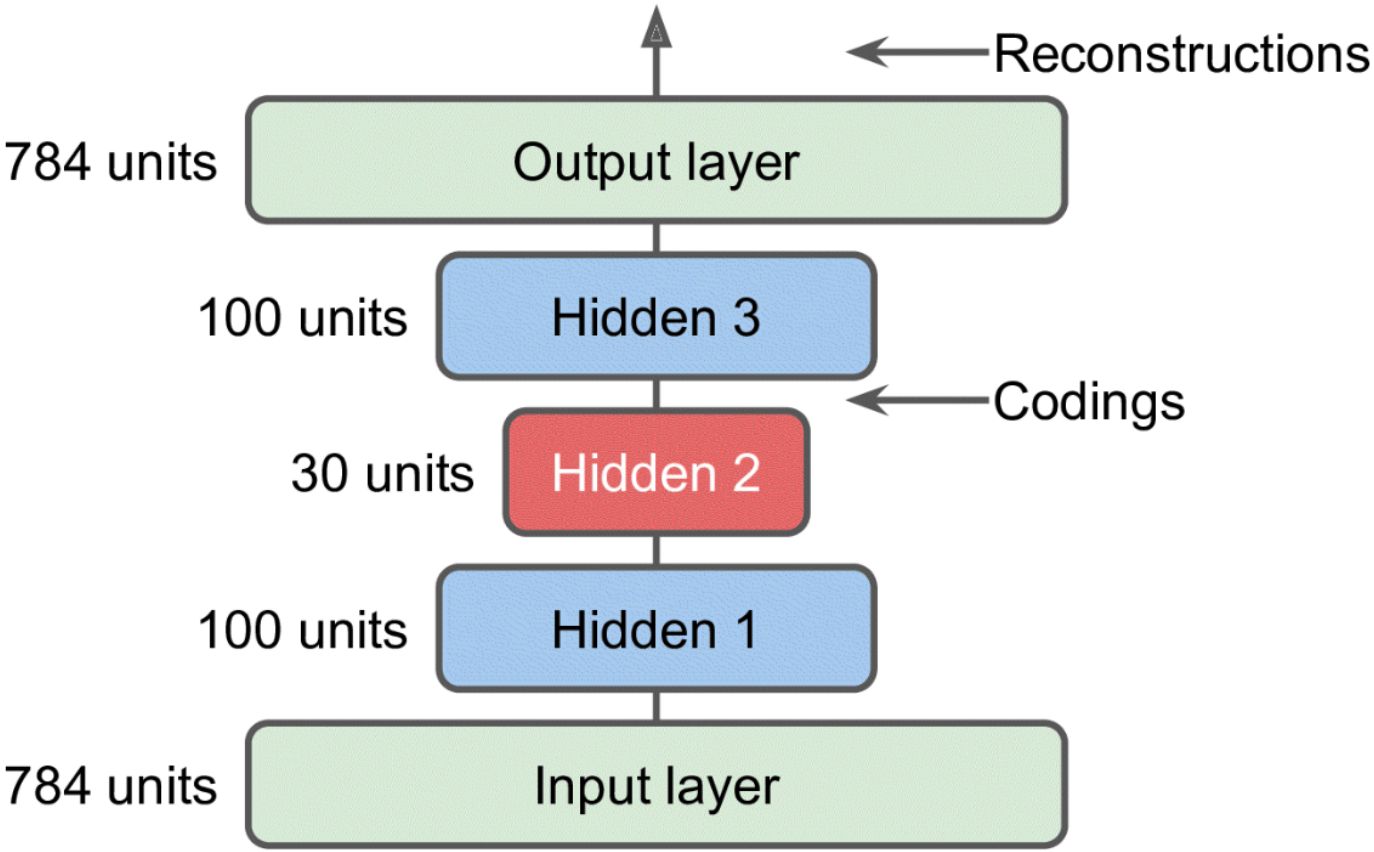
Remarks:

- Outputs are called *reconstructions*
- Cost function is based on *reconstruction loss* that measures deviations of input from output
- Dimensionality of the internal representation is lower - the autoencoder is called "undercomplete" and has to learn the most important features

Contents

Stacked Autoencoders

Example:



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

Stacked Autoencoders

Example (continued):

```
stacked_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(30, activation="selu"),
])
stacked_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[30]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
stacked_ae = keras.models.Sequential([stacked_encoder, stacked_decoder])
```

```
history = stacked_ae.fit(X_train, X_train, epochs=10,
                         validation_data=[X_valid, X_valid])
```

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

Stacked Autoencoders

Example (continued):

```
def plot_image(image):
    plt.imshow(image, cmap="binary")
    plt.axis("off")

def show_reconstructions(model, n_images=5):
    reconstructions = model.predict(X_valid[:n_images])
    fig = plt.figure(figsize=(n_images * 1.5, 3))
    for image_index in range(n_images):
        plt.subplot(2, n_images, 1 + image_index)
        plot_image(X_valid[image_index])
        plt.subplot(2, n_images, 1 + n_images + image_index)
        plot_image(reconstructions[image_index])

show_reconstructions(stacked_ae)
```

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

Stacked Autoencoders

Example (continued):

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

Contents

Visualization via t-SNE

Example (continued):
Let's visualize 30 features using t-distributed stochastic neighbor embedding (t-SNE):

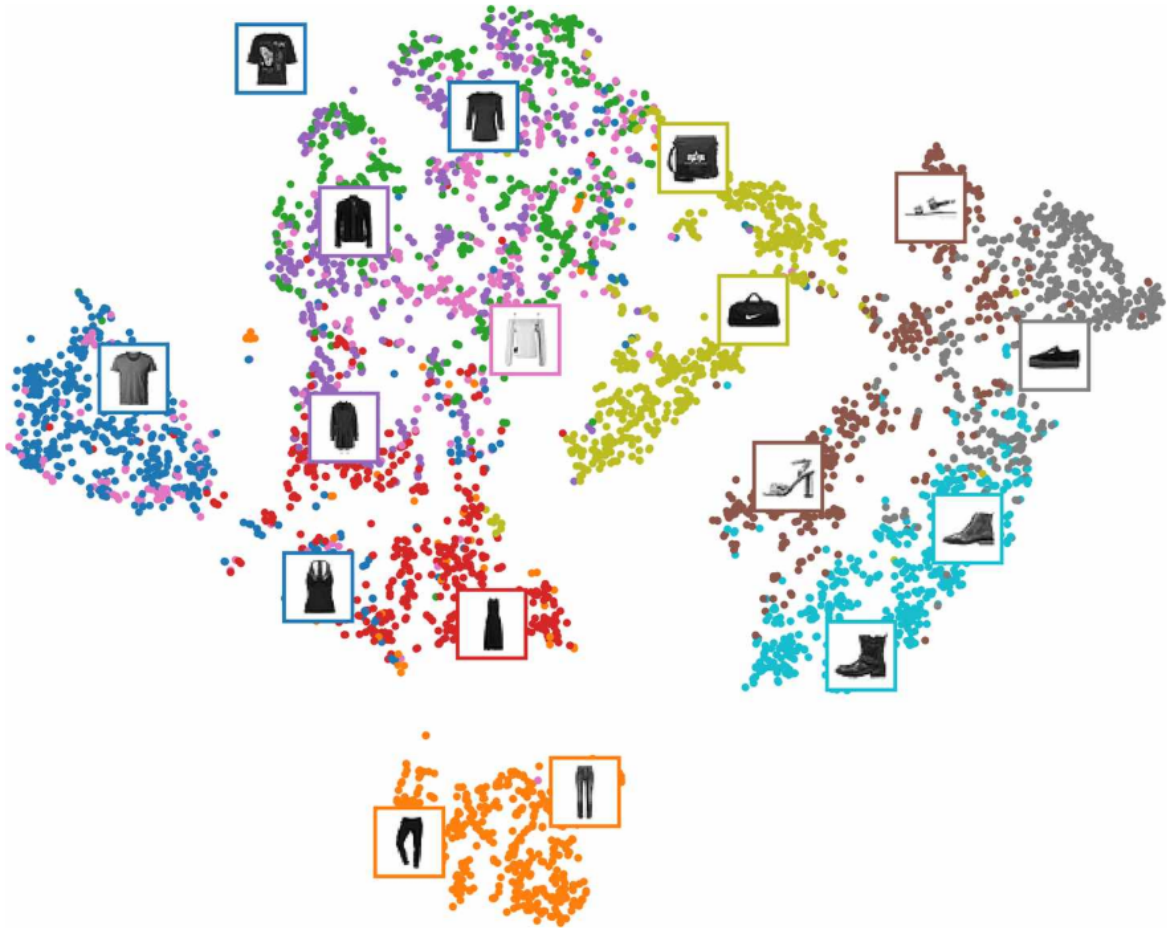
```
X_valid_compressed = stacked_encoder.predict(X_valid)
tsne = TSNE()
X_valid_2D = tsne.fit_transform(X_valid_compressed)
```

```
plt.scatter(X_valid_2D[:, 0], X_valid_2D[:, 1], c=y_valid, s=10, cmap="tab10")
```

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

Visualization via t-SNE

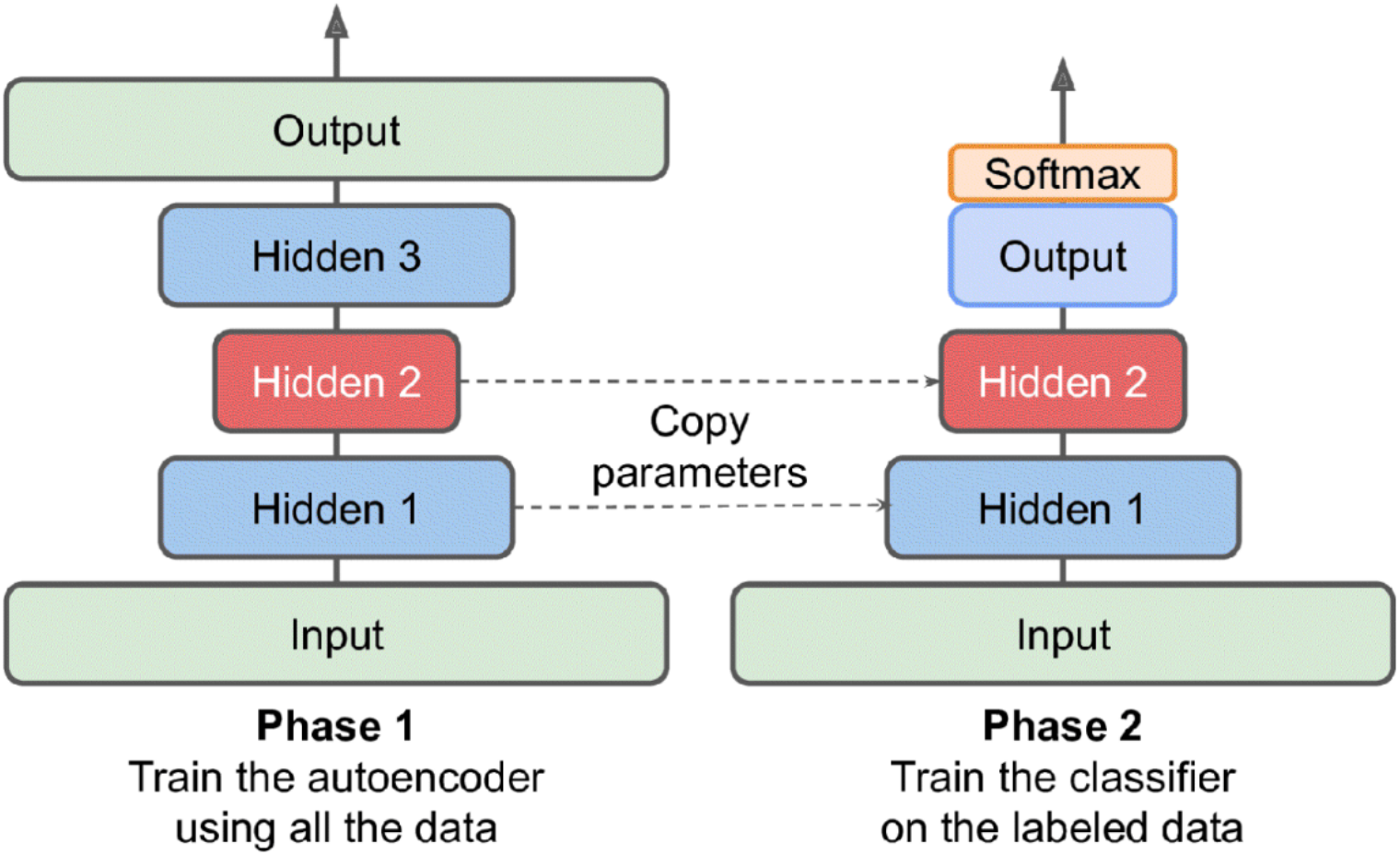
Example (continued):
Let's visualize 30 features using t-distributed stochastic neighbor embedding (t-SNE):



Contents

Unsupervised Pretraining using Autoencoders

What if most of the labels are missing?



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

Contents

Training Autoencoders

- One can impose constraints by tying weights of encoder/decoder
- One can train one encoder at a time and then stuck all autoencoders into one stacked autoencoder (sandwich) - this way a very deep stakced autoencoder can be built

Contents

Sparse Autoencoders

L1 Activity Regularization

```
sparse_l1_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid"),
    keras.layers.ActivityRegularization(l1=1e-3)
])
sparse_l1_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_l1_ae = keras.models.Sequential([sparse_l1_encoder, sparse_l1_decoder])
```

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

Contents

Sparse Autoencoders

Kullback–Leibler Divergence (or Relative Entropy)

The Kullback–Leibler divergence is defined as follows:

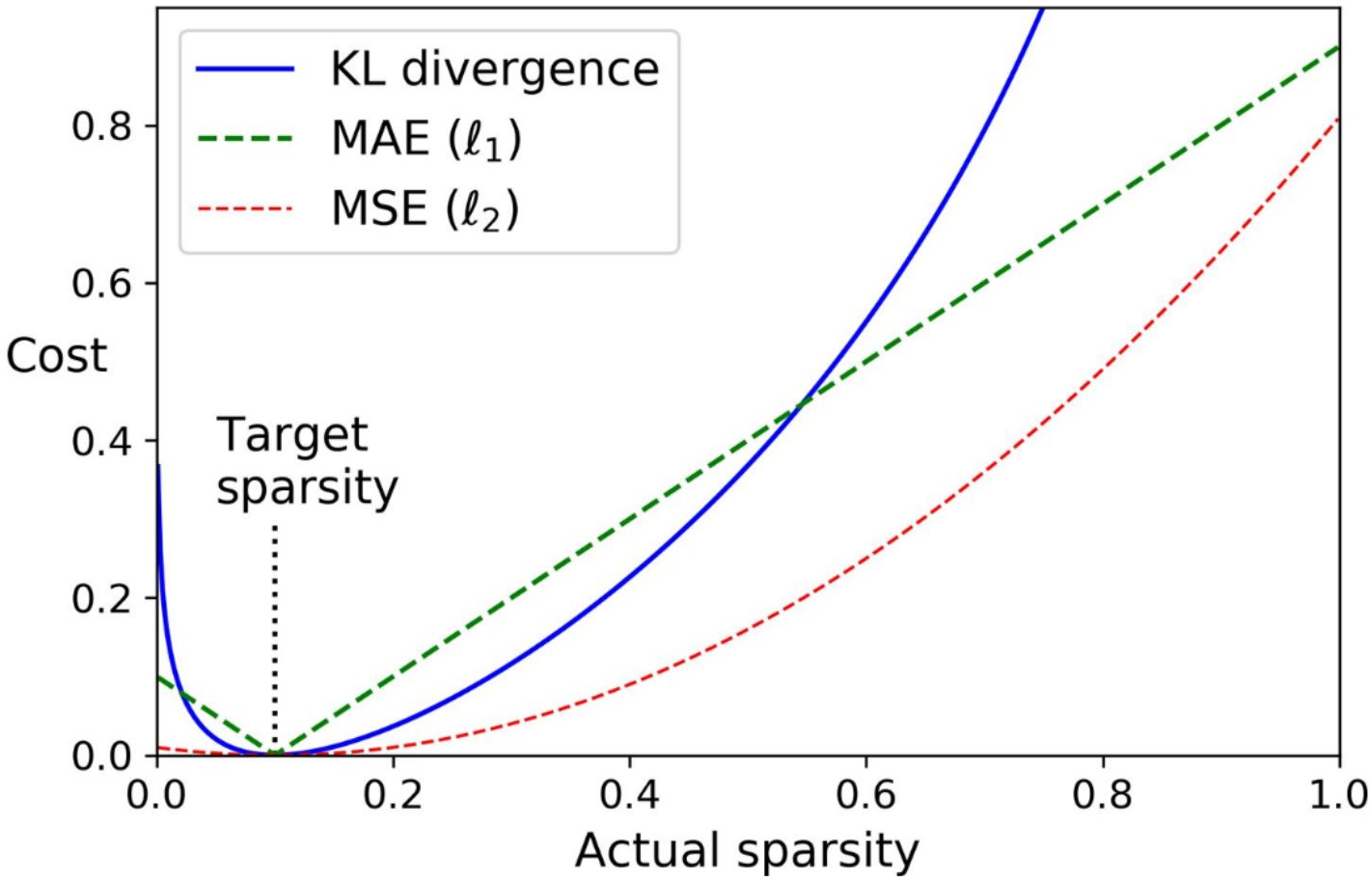
$$D_{\mathsf{KL}}(p \,\|\, \hat{p}) = p \ln \frac{p}{\hat{p}} + (1-p) \ln \frac{1-p}{1-\hat{p}},$$

where

p and $\hat{p}$ are the target and actual (observed) sparsity, respectively.

Sparse Autoencoders

Kullback–Leibler Divergence (or Relative Entropy)



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

Sparse Autoencoders

Kullback–Leibler Divergence (or Relative Entropy)

Example:

```
K = keras.backend
kl_divergence = keras.losses.kullback_leibler_divergence

class KLDivergenceRegularizer(keras.regularizers.Regularizer):
    def __init__(self, weight, target=0.1):
        self.weight = weight
        self.target = target
    def __call__(self, inputs):
        mean_activities = K.mean(inputs, axis=0)
        return self.weight * (
            kl_divergence(self.target, mean_activities) +
            kl_divergence(1. - self.target, 1. - mean_activities))
```

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

Sparse Autoencoders

Kullback–Leibler Divergence (or Relative Entropy)

Example (continued):

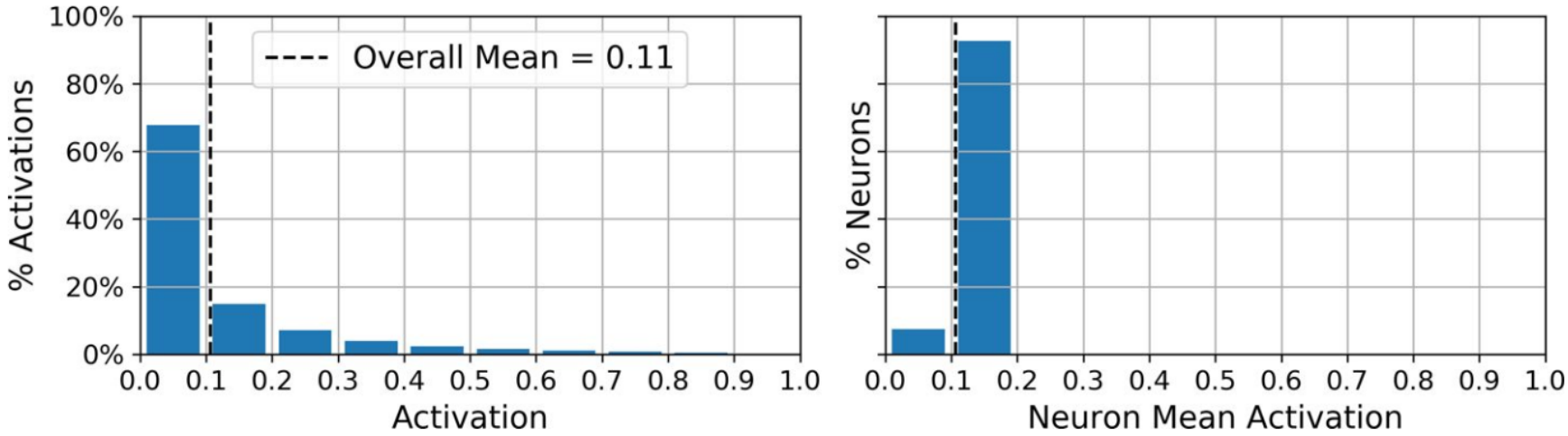
```
kld_reg = KLDivergenceRegularizer(weight=0.05, target=0.1)
sparse_kl_encoder = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[28, 28]),
    keras.layers.Dense(100, activation="selu"),
    keras.layers.Dense(300, activation="sigmoid", activity_regularizer=kld_reg)
])
sparse_kl_decoder = keras.models.Sequential([
    keras.layers.Dense(100, activation="selu", input_shape=[300]),
    keras.layers.Dense(28 * 28, activation="sigmoid"),
    keras.layers.Reshape([28, 28])
])
sparse_kl_ae = keras.models.Sequential([sparse_kl_encoder, sparse_kl_decoder])
```

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

Sparse Autoencoders

Kullback–Leibler Divergence (or Relative Entropy)

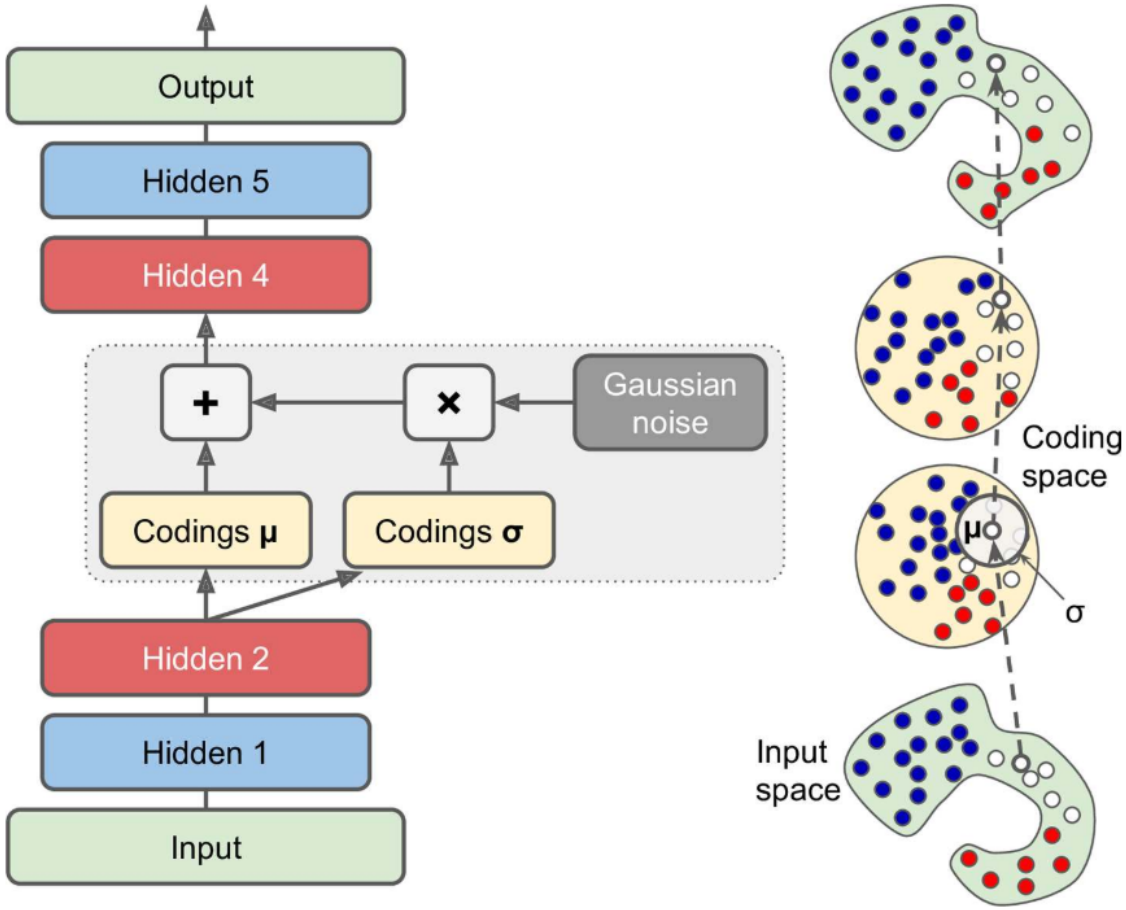
Example (continued):



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

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