Embeddings

Introduction to Embeddings

- **Embeddings** are learnable data representations that map high-cardinality data into a lower-dimensional space.
- They are designed to preserve information relevant to the learning problem.
- Embeddings are fundamental to modern machine learning, appearing in various forms across the field.

Example: One-Hot Encoding in the Natality Dataset

- **One-hot encoding** is a common way to represent categorical input variables.
- Example: The `plurality` feature in the natality dataset has six categories:
  - `['Single(1)', 'Multiple(2+)', 'Twins(2)', 'Triplets(3)', 'Quadruplets(4)', 'Quintuplets(5)']`
- Each value is mapped to a unit vector in $\mathbb{R}^6$:

Plurality	One-hot Encoding
Single(1)	[1, 0, 0, 0, 0, 0]
Multiple(2+)	[0, 1, 0, 0, 0, 0]
Twins(2)	[0, 0, 1, 0, 0, 0]
Triplets(3)	[0, 0, 0, 1, 0, 0]
Quadruplets(4)	[0, 0, 0, 0, 1, 0]
Quintuplets(5)	[0, 0, 0, 0, 0, 1]

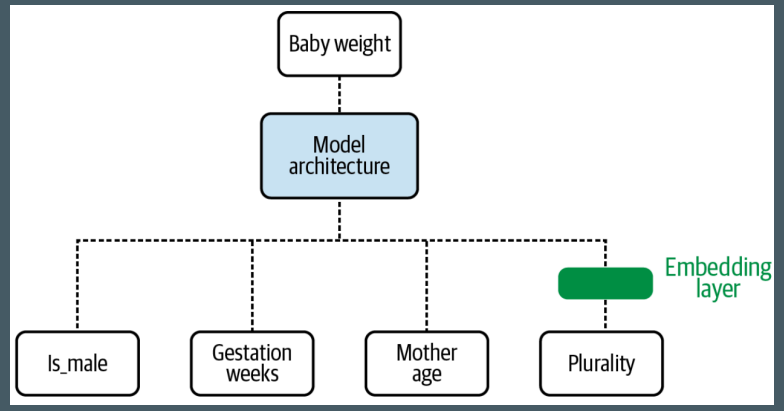
Example: One-Hot Encoding in the Natality Dataset

- One-hot encoding treats categories as independent and equally distant.
- But some categories are naturally **closer** to each other (e.g., twins and triplets).
- **Embeddings** can capture these relationships by placing similar categories closer together in a lower-dimensional space.

Plurality	Candidate Embedding
Single(1)	[1.0, 0.0]
Multiple(2+)	[0.0, 0.6]
Twins(2)	[0.0, 0.5]
Triplets(3)	[0.0, 0.7]
Quadruplets(4)	[0.0, 0.8]
Quintuplets(5)	[0.0, 0.9]

How Are Embeddings Learned?

- The **Embeddings design pattern** uses an embedding layer with trainable weights.
- Categorical inputs are mapped to real-valued vectors in a lower-dimensional space.
- The embedding layer's weights are learned during model training, as part of the overall optimization.
- As a result, the learned embeddings reflect relationships in the data—similar categories are mapped to similar vectors.



Embeddings and Related Techniques

- **Embeddings** capture closeness relationships between categories in a lower-dimensional space.
- Embedding layers can **replace clustering** (e.g., for customer segmentation) and **dimensionality reduction** methods like principal component analysis (PCA).
- Unlike traditional clustering or PCA, embedding weights are learned automatically during model training.
- This eliminates the need to apply clustering or PCA as a separate, pre-processing step.

Your task

- Execute the code supplied in `embedding.ipynb`
- Compare the values of the resulting embedding with the values of your neighbour
- Compare the weights of the embedding used with the values of your neighbour
- Interpret the result

Your task

- Have a closer look at `plurality_class`
- Does it look like a vector of one-hot-encoded values?
- Try to explain why the weights of the embeddinglayer have `shape(6,2)`

Introduction to Text Embeddings

- **Text** is a natural candidate for embeddings due to its high-cardinality vocabulary (often tens of thousands of words).
- One-hot encoding each word would result in extremely large and sparse matrices, which are inefficient for training.
- We want **similar words** to have embeddings that are close together, and **unrelated words** to be far apart in the embedding space.
- Therefore, we use **dense word embeddings** to vectorize text inputs before passing them to the model.

Tokenization: Mapping Words to Indices

- **Tokenization** creates a lookup table that maps each word in the vocabulary to a unique index.
- This is similar to one-hot encoding:
  The tokenized index indicates the position of the “1” in the one-hot vector for that word.
- Building the lookup table requires a pass over the entire dataset (e.g., all article titles).
- Tokenization can be efficiently implemented using libraries like **Keras**.

Example: Tokenization and One-Hot Encoding

Suppose our sentence is: "Machine learning is fun"

Word	One-Hot Encoding
Machine	[1, 0, 0, 0]
learning	[0, 1, 0, 0]
is	[0, 0, 1, 0]
fun	[0, 0, 0, 1]

Word	Token Index
Machine	1
learning	2
is	3
fun	4

The index corresponds to the position of the "1" in the one-hot encoding.

Tokenization in Keras: Code Example

```
from tensorflow.keras.preprocessing.text import Tokenizer

# 1. Create a tokenizer object
tokenizer = Tokenizer()

# 2. Fit the tokenizer on your text data (e.g., article titles)
tokenizer.fit_on_texts(titles_df.title)
```

What does this do?

The Tokenizer builds a vocabulary and assigns an index to each word based on frequency in `titles_df.title`.

Tokenization in Keras: Code Example

```
# 3. Convert the text to sequences of token indices
integerized_titles = tokenizer.texts_to_sequences(titles_df.title)
```

What does this do?

Transforms each title into a list of integers, where each integer represents the index of a word in the vocabulary.

Adding an Embedding Layer to a DNN in Keras

- We can build a deep neural network (DNN) in Keras with an **embedding layer** to transform word indices into dense vectors.
- The `Embedding` layer maps each integer index (for a word) to a dense, trainable vector—its embedding.
- Key arguments:
  - `input_dim`: Size of the vocabulary.
  - `output_dim`: Dimensionality of the embedding vectors.
  - `input_shape`: Length of each input sequence (e.g., after padding).
- Typically, input sequences (like padded article titles) are fed to the model for classification or further processing.

Example: Simple Keras DNN with Embedding Layer

```
model = models.Sequential([
    layers.Embedding(input_dim=VOCAB_SIZE + 1,  # Size of vocabulary (+1 for padding token)
                     output_dim=embed_dim,      # Dimension of embedding vectors
                     input_shape=[MAX_LEN]),    # Length of input sequences
    layers.Lambda(lambda x: tf.reduce_mean(x, axis=1)),  # Average embeddings (simple pooling)
    layers.Dense(N_CLASSES, activation='softmax')        # Output layer for classification
])
```

This model:

- Converts integerized text to dense embeddings,
- Averages (pools) the embeddings for each input,
- Outputs class probabilities via a dense softmax layer.

Task: Understanding Output Shapes in a DNN with Embeddings

Assume the following:

- Input batch shape: `(26, 100)` (26 is max sequence length, 100 is your vocabulary size)
- Embedding dimension: `8`
- Number of classes: `3`

Fill in the output shape after each layer in the table below.

Layer	Output Shape
Embedding	
Lambda (reduce_mean)	
Dense	

Work with a neighbor and be prepared to share your reasoning!

Follow Up Tasks:

- How many parameters does the embedding layer have?
- How many does the lambda layer have?
- How many does the dense layer have?
- In this setup, why do we need a layer to reduce to the mean?

Trade-Offs and Choosing Embedding Dimensions

- **Trade-Offs with Embeddings**
  - Going from high-cardinality to lower-dimensional embeddings means some information is lost.
  - In exchange, embeddings capture **closeness** and **contextual relationships** between categories.
- **Choosing Embedding Dimension**
  - The embedding size is a tunable parameter with its own trade-offs:
    - **Too small:** Not enough capacity, important context is lost.
    - **Too large:** Model may memorize categories (like one-hot), losing generalization.

Trade-Offs and Choosing Embedding Dimensions

- **Rules of Thumb:**
  - Embedding dimension ≈ 4th root of the number of unique categories.
  - Alternative: $1.6 \cdot \sqrt{\text{number of unique categories}}$, but no less than 600.
  - **Example:** For 625 categories:
    - 4th root rule: 5
    - Square root rule: 40
- **Best Practice:**
  The optimal embedding dimension is found by experimentation (hyperparameter tuning).

Autoencoders: Learning Embeddings Without Labels

- **Supervised embeddings** (e.g., from image classifiers) require massive labeled datasets, which are often hard to obtain.
- **Autoencoders** provide an unsupervised alternative:
  - No need for labeled data—just input examples.
- **Architecture:**
  - The **encoder** maps high-dimensional input to a lower-dimensional "bottleneck" layer (the embedding).
  - The **decoder** reconstructs the input from this embedding.
- The network is trained to minimize **reconstruction error**:
  - The output is forced to be as similar as possible to the input.
- The bottleneck layer **acts as an embedding layer**, capturing a compressed, meaningful representation of the input.

Autoencoders for Nonlinear Dimensionality Reduction

- Since the input equals the output, **no extra labels are needed** for training autoencoders.
- The **encoder** learns an optimal **nonlinear dimensionality reduction** of the input.
  - This is similar to PCA (Principal Component Analysis), but PCA is limited to linear reduction.
  - The autoencoder's bottleneck layer can capture **nonlinear relationships** in the data.
- **Practical Workflow:**
  1. **Train the autoencoder** on all available (unlabeled) data to learn a low-dimensional embedding.
  2. Use this **embedding as input** to your supervised task (e.g., classification), where labeled data is limited.
  3. This reduces the complexity of the supervised task—fewer weights to learn, potentially improving performance.

Word2Vec and BERT: Learning Powerful Text Embeddings

- **Word2Vec**
  - Uses shallow neural networks with two main architectures:
    - **CBOW (Continuous Bag of Words):** Predicts a word from its surrounding context.
    - **Skip-gram:** Predicts context words from a target word.
  - Trained on large text corpora (e.g., Wikipedia).
  - Learns low-dimensional embeddings that capture semantic relationships—meaningful distances and directions in embedding space.
  - Embedding for a word is **the same in any context** (static embedding).

Word2Vec and BERT: Learning Powerful Text Embeddings

- **BERT**
  - Trained using two auxiliary tasks:
    - **Masked Language Model:** Randomly masks words and predicts the missing words.
    - **Next Sentence Prediction:** Predicts if two sentences are consecutive in the original text.
  - Trained on large corpora (Wikipedia, BooksCorpus).
  - Produces **contextual embeddings**—the embedding for a word changes depending on how it is used in a sentence.
  - Embeddings from BERT are highly effective for many downstream NLP tasks.

Word2Vec and BERT: Learning Powerful Text Embeddings

Summary:
Both Word2Vec and BERT create word embeddings useful for transfer learning, but Word2Vec yields static embeddings, while BERT produces embeddings that adapt to context.