Multimodal Input

Introduction to Multimodal Inputs

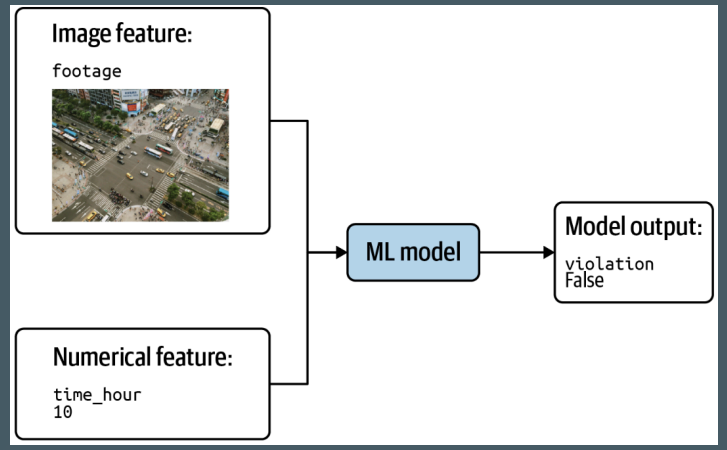
- The **Multimodal Input design pattern** addresses the challenge of representing different types of data or data with complex structures.
- The key idea is to **concatenate all available data representations**—such as text, images, numerical features, or categorical variables—into a single combined input for a model.
- This approach enables machine learning models to make use of diverse information sources within a unified framework.

Problem: Handling Different Data Types

- Most machine learning models are designed for a **single input type**—such as numbers, categories, images, or text.
  - Example: Standard image classifiers like ResNet-50 only process images, not metadata.
- Real-world tasks often require combining different data sources.

Example Scenario: Traffic Violation Detection**

- A camera at an intersection records footage (image data).
- Additional metadata is available:
  - Time of day
  - Day of week
  - Weather conditions
- We need a model that can use **both** the camera footage and the metadata as inputs.



Example Scenario: Predicting Restaurant Ratings with Multimodal Inputs

- Consider the task of predicting a restaurant patron's rating.

Available inputs:

- **Review text:** The customer's written feedback (free-form text).
- **Numerical attributes:** What the customer paid (numeric value).
- **Categorical attributes:** Whether it was lunch or dinner (category).

To make the best prediction, a model should be able to combine insights from text, numbers, and categories. This is a classic multimodal input problem.

Combining Numerical and Categorical Features

- In our restaurant review example, the model can use both numerical and categorical metadata about the meal.
- **Categorical feature:**
  - `meal_type` with three options (e.g., breakfast, lunch, dinner)
  - One-hot encode `meal_type`, e.g., dinner → `[0, 0, 1]`
- **Numerical feature:**
  - `meal_total`, e.g., the price paid for the meal (e.g., `30.5`)
- **Combined feature vector:**
  - Concatenate the one-hot vector and the numerical value: `[0, 0, 1, 30.5]`

This unified vector can now be used as input to a machine learning model.

Adding Text Features: What Does the Flatten Layer Do?

- The **Flatten** layer reshapes multi-dimensional data into a single vector for each input sample.
- This is often needed before concatenating with other features or passing data to dense layers.

Example:
Suppose after an Embedding layer you have:

- Output shape: `(batch_size, 4, 3)`
  - For each sample: 4 tokens, each represented by a 3-dimensional embedding
  - Example for one sample:

```
[
[0.1, 0.2, 0.3],   # Token 1
[0.4, 0.5, 0.6],   # Token 2
[0.7, 0.8, 0.9],   # Token 3
[1.0, 1.1, 1.2]    # Token 4
]
```

Adding Text Features: What Does the Flatten Layer Do?

Flattened output for this sample:

`[0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0, 1.1, 1.2]`

(Length: 4 × 3 = 12)

- Output shape after Flatten: `(batch_size, 12)`

In summary:

The Flatten layer converts each sample from a matrix or tensor into a single vector, making it easier to combine with other features.

Task: Debugging a Multimodal Keras Model

You are given a Keras model built using the Functional API to combine **text(review) input** and **tabular input** (meal type and price) for a restaurant rating prediction task.

Model Description:

- **Text input:**
  - The review is tokenized as a sequence of up to 30 integers (`embedding_input`).
  - The input is passed through an Embedding layer (to create a sequence of embedding vectors).
- **Tabular input:**
  - Meal information (a one-hot encoded vector for meal type plus meal price, total length 4) is provided via `tabular_input`.
  - This passes through a Dense layer with 32 units and ReLU activation for further feature transformation.
- The outputs from both pathways (sentence embedding and the processed tabular features) are concatenated.

Task:
You will receive model code that currently raises an error related to input shapes.
Your job:

- Examine the code carefully.
- Identify what might be wrong in the data flow from the Embedding layer to the Dense and concatenation layers.
- Fix the code so that it runs without shape errors and produces the expected model output.

Tip: Pay special attention to the dimensions of data at each stage, especially before applying Dense and concatenation operations!

Representing the Same Data in Different Ways

- The **Multimodal Input design pattern** is not just about mixing data types (e.g., text and metadata).
- We can also represent **the same data in multiple ways** to help our model learn better patterns.

Example:

- A `ratings` field (1 to 5 stars) can be treated as both:
  - A numeric variable (for regression)
  - A categorical variable (for classification or one-hot encoding)

We'll start by exploring how tabular data can be represented in different ways,
and then move on to text and images.

Example: Multiple Representations for Tabular Data

Suppose **rating** (1 to 5 stars) is an input feature, and we want to predict the review's usefulness (e.g., number of "likes").

How can we represent the `rating` input?

- **As an integer:** The original 1–5 value, used as a numerical feature.
- **As a categorical feature:**
  - We can **bucket** the ratings. Example:
    - "Good" = 4 or 5 stars, enocded as `1`
    - "Bad" = 3 stars or fewer, encoded as `0`

Combined input:
Concatenate the integer rating and the boolean bucket: E.g., `[4, 1]` for a 4-star "good" rating

This gives the model both granular and grouped information to use when finding patterns.

Text Data: Multiple Representations

Text is inherently complex, and there are **many ways to represent it** for machine learning.

- **Embeddings:**
  - Capture similarities and relationships between words.
  - Enable the model to understand context and syntax.
  - Mimic how humans intuitively process language.
- **Bag of Words:**
  - Represents text as a count or presence/absence of words.
  - Ignores order and context, but can be highly effective for certain tasks.

Why use multiple representations?

- Combining different approaches can help the model capture both subtle relationships (via embeddings) and explicit signals (via bag of words).
- Additional tabular features can also be extracted from text (e.g., review length, sentiment score) to further boost performance.

We'll next explore the bag of words approach and how to extract tabular features from text.

Bag of Words (BOW): A Simple Text Representation

- **Bag of Words (BOW)** imagines each text input as a "bag" of words, like Scrabble tiles.
- **Order is ignored**—the model only cares if a word is present or absent.
- BOW is a type of **multi-hot encoding:**
  Each input is an array of 1s and 0s, with each index corresponding to a word in the vocabulary.

Example:
Suppose our vocabulary is: `["delicious", "service", "slow", "friendly"]`

Text:

> "The service was friendly and delicious."

BOW representation:

- "delicious": 1
- "service": 1
- "slow": 0
- "friendly": 1

So the BOW array for this input is: **[1, 1, 0, 1]**

This simple array summarizes which words are present in the review.

Combining Bag of Words and Embedding Representations

- Using the Multimodal Input approach, we can concatenate the BOW and embedding representations for each input before feeding them into the model.

Why combine both?

- **BOW** encoding provides strong signals for the presence of significant words in our vocabulary.
- **Embeddings** capture relationships between words—even for a much larger vocabulary.
- BOW is good for exact matches (e.g., reviews containing "amazing"), while embeddings can capture meaning and context (e.g., distinguishing "amazing" vs. "not amazing").
- BOW treats word presence as boolean, while embeddings can encode frequency and context.

In summary:
By combining both representations, our model can leverage explicit signals and nuanced relationships in the text—often improving predictive performance.

Task: Merging Two Text Representations

You are provided with code that builds both a Bag of Words layer and an Embedding layer for text input.
Each layer processes the text data in a different way, producing two separate outputs.

Your task:

- Write the code needed to combine the outputs of the Bag of Words and Embedding pathways, so that both can be used together by the next layers in the model.
- Make sure the merged result can be passed as input to a Dense layer for further processing.

Tip: Think about how to bring together information from both representations to form a single input for the next step in the network!

Outlook: Enriching Models with Tabular Features from Text

When working with text data, there are often useful characteristics beyond the raw words themselves that can be represented as tabular features.

Example Scenario:

- Suppose we want to predict whether a Stack Overflow question will receive a response.
- Factors beyond the actual words may be important, such as:
  - **Length of the question** (number of words or characters)
  - **Presence of a question mark** (punctuation)
  - **Number of tags** assigned to the question
  - **Day of the week** the question was posted

Why is this valuable?

- Standard text encodings (like embeddings) often **truncate text** or remove punctuation, losing potentially important information.
- By extracting these features and representing them as additional columns, we can provide the model with more context.

Using the Multimodal Input pattern:

- Combine these tabular features with encoded text representations (like BOW or embeddings).
- Feed both into the model to help capture patterns that may not be evident from text alone.

Multiple Ways to Represent Image Data

Just as with text, images can be represented in **different ways** when preparing them for machine learning models.

Common image representations:

- **Pixel values:** The most basic approach is to represent the image as an array of raw pixel values (e.g., RGB channels).
- **Sets of tiles:** The image can be divided into smaller, non-overlapping patches or tiles, which may help the model focus on local patterns or features.
- **Windowed sequences:** The image can be represented as a sequence of overlapping patches or "windows," capturing spatial context as the window slides across the image.

The **Multimodal Input design pattern** allows us to combine multiple image representations within a single model, potentially improving performance by capturing different aspects of the data.

Representing Images as Pixel Values

- At the most basic level, images are **arrays of pixel values**.
  - Example: A black-and-white (grayscale) image has integer values from 0 to 255.
  - A 28×28 grayscale image (like those in MNIST) is a 28×28 array of pixel values.
- To use these images as model inputs, we **flatten** the 2D array into a 1D vector.
  - In Keras, this can be done with:

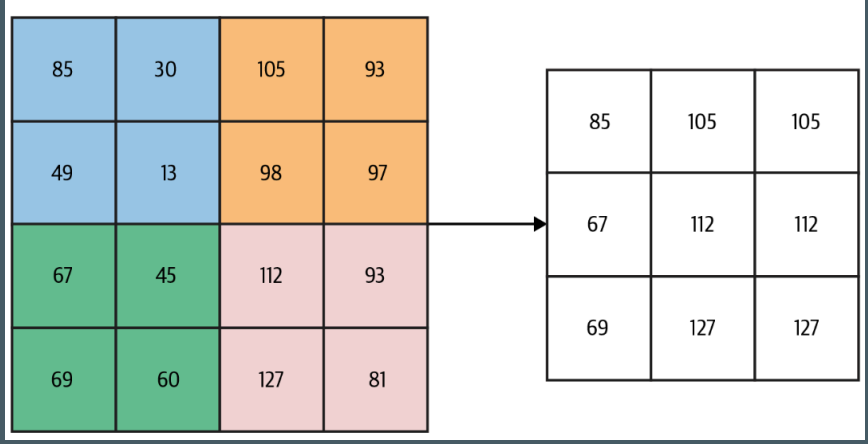
```
layers.Flatten(input_shape=(28, 28))
```

- **Color images** (RGB) add a third dimension for color channels:
  - Each pixel has three values (red, green, blue), and the flattened vector has 2,352 elements (28 × 28 × 3).
- While flattening pixel values is effective for simple images, it can be limiting for more complex images.
- Feeding all pixels at once makes it harder for the model to focus on **local patterns**, such as edges or shapes, that are important for understanding the image.

Images as Tiled Structures

- Real-world images are often too complex for models to understand when represented only as flat arrays of pixel values.
- To help the model **extract meaningful details and recognize patterns**, we can divide images into **small, local pieces**—or "tiles."
- By processing only a **small region of the image at a time**, the model can more easily detect features like **edges** and **spatial gradients** that appear in neighboring pixels.
- This approach is the basis for **convolutional neural networks (CNNs)**, which are designed to learn from local patterns and build up an understanding of the entire image by moving across these tiles.

- **Convolutional neural networks (CNNs)** use a technique called a **sliding window** to process these tiles:
  - A small window (or filter) moves across the image, focusing on a few pixels at a time.
  - The **stride** determines how many pixels the window moves with each step.
    - A stride of 1 means the window moves one pixel at a time, resulting in overlapping regions.
    - A higher stride skips more pixels, resulting in less overlap.
  - This sliding window approach enables the model to detect local features in different parts of the image and to build up a global understanding by combining these local patterns.



Example: Combining Pixel and Tiled Representations

Below is a sample implementation showing how to represent the same image both as raw pixels and as tiles processed by a convolutional layer, then combine both representations:

```
# Define image input layer (same shape for both pixel and tiled representation)
image_input = Input(shape=(28, 28, 3))

# Pixel (flattened) representation
pixel_layer = Flatten()(image_input)

# Tiled (convolutional) representation
tiled_layer = Conv2D(filters=16, kernel_size=3, strides=(2, 2), activation='relu')(image_input)
tiled_layer = MaxPooling2D()(tiled_layer)
tiled_layer = Flatten()(tiled_layer)

# Combine both representations into one layer
merged_image_layers = keras.layers.concatenate([pixel_layer, tiled_layer])
```