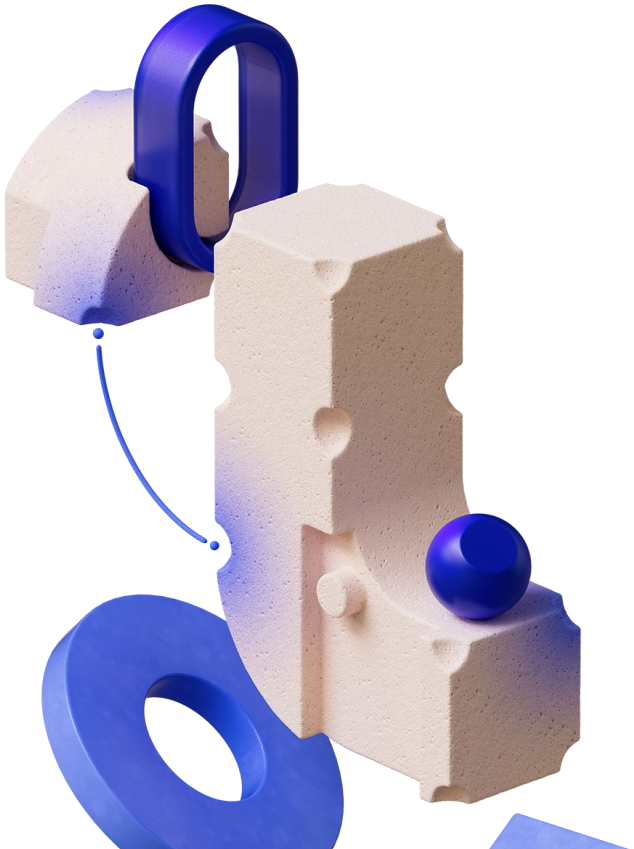
Google DeepMind



Introduction to Transformers with Jax

Mai Giménex
Senior Research engineer

A tutorial for everyone

🌱Beginner

It is my first time being introduced to this work.

 Intermediate

I have done some basic courses/intros on this topic.

🌳Advance

I work in this area/topic daily.

A tutorial for everyone

🛠️ Exercises

Here, you'll need to implement something to demonstrate you understand the concepts described.

⚙️Examples

Examples and code needed to execute the tutorial.

⚠️Use a GPU

For this practical, you will need to use a GPU to speed up training. To do this, go to the "Runtime" menu in Colab, select "Change runtime type", and then in the popup menu, choose "GPU" in the "Hardware accelerator" box.

Contents

An introduction to JAX

- A differentiable numpy-like library that runs on accelerators.
- Python code is traced to an intermediate representation that enables domain-specific compilation.
- Designed to be functional, it allows to transform Python code.

- Common JAX transformations:
  - grad(): computes the gradient from a numerical function.
  - jit(): compiles multiple operations together using XLA.
  - vmap(): performs automatic vectorization or batching.
  - pmap(): performs automatic parallelization.

From numpy to JAX

Numpy

```python
from numpy import np

def predict(params, inputs):
  for W, b in params:
    outputs = np.dot(inputs, W) + b
    inputs = np.tanh(outputs)

def mse_los(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return np.sum((preds - targets) ** 2)
```

JAX

```python
from jax import jnp

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)

def mse_los(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return jnp.sum((preds - targets) ** 2)
```

From an Intro to JAX: Accelerating Machine Learning research

Automatic differentiation

jax.grad()

- Takes a numerical function and returns a new Python function that computes the gradient of the original function.
- Analogous to the ∇ operator from vector calculus.
- Controls which variables to differentiate with respect to.
- If you need the gradient function and the value you can use jax.value_and_grad

```
from jax import jnp
from jax import grad

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)

def mse_los(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return jnp.sum((preds - targets) ** 2)

gradient_fn = grad(mse_loss)
```

From an Intro to JAX: Accelerating Machine Learning research

Compilation

jax.jit()

- Compiling jax code makes it performant.
- The same code can be run on CPU, GPU, or TPU.
- jit transforms Python code into an **intermediate representation language jaxpr using tracing.**
- It doesn't capture side effects.
- The default level of tracing is ShapedArray – that is, each tracer has a concrete shape but no concrete value.

```
from jax import jnp
from jax import grad, jit

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)

def mse_los(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return jnp.sum((preds - targets) ** 2)

gradient_fn = jit(grad(mse_loss))
```

From an Intro to JAX: Accelerating Machine Learning research

Vectorization/Parallelization

jax.vmap()

- Transforms functions to **operate over vector.**
- Using in_axes and out_axes specifies the location of the batch dimension in inputs and outputs

jax.pmap()

- Parallelise operations across multiple XLA devices.

```
from jax import jnp
from jax import grad, jit, pmap, vmap

def predict(params, inputs):
  for W, b in params:
    outputs = jnp.dot(inputs, W) + b
    inputs = jnp.tanh(outputs)

def mse_los(params, batch):
    inputs, targets = batch
    preds = predict(params, inputs)
    return jnp.sum((preds - targets) ** 2)

gradient_fn = jit(grad(mse_loss))
per_examples_gradient = jit(vmap(gradient_fn))
parallel_gradients = jit(pmap(gradient_fn))
```

From an Intro to JAX: Accelerating Machine Learning research

An introduction to Flax

Flax is a high-performance neural network library for JAX, designed for flexibility and reproducibility.

- **Pythonic**: Flax NNX supports the use of regular Python objects, providing an intuitive and predictable development experience.
- **Simple**: Flax NNX relies on Python's object model, which results in simplicity for the user and increases development speed.
- **Expressive**: Flax NNX allows fine-grained control of the model's state via its Filter system.
- **Familiar**: Flax NNX makes it very easy to integrate objects with regular JAX code via the Functional API.

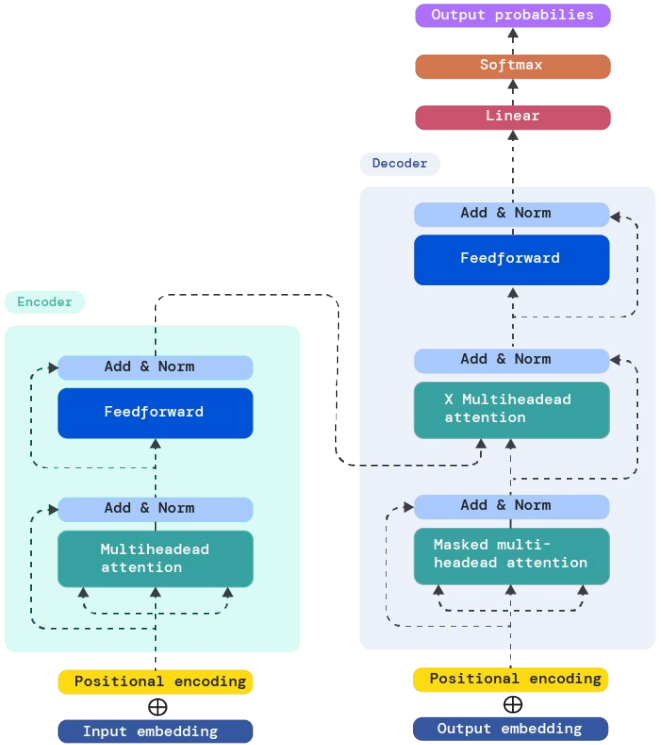
```python
from flax import nnx
import optax


class Model(nnx.Module):
  def __init__(self, din, dmid, dout, rngs: nnx.Rngs):
    self.linear = nnx.Linear(din, dmid, rngs=rngs)
    self.bn = nnx.BatchNorm(dmid, rngs=rngs)
    self.dropout = nnx.Dropout(0.2, rngs=rngs)
    self.linear_out = nnx.Linear(dmid, dout, rngs=rngs)

  def __call__(self, x):
    x = nnx.relu(self.dropout(self.bn(self.linear(x))))
    return self.linear_out(x)

model = Model(2, 64, 3, rngs=nnx.Rngs(0))  # eager initialization
optimizer = nnx.Optimizer(model, optax.adam(1e-3))  # reference
sharing
@nnx.jit  # automatic state management for JAX transforms
def train_step(model, optimizer, x, y):
  def loss_fn(model):
    y_pred = model(x)  # call methods directly
    return ((y_pred - y) ** 2).mean()

  loss, grads = nnx.value_and_grad(loss_fn)(model)
  optimizer.update(grads)  # in-place updates

  return loss
```

The quickest guide to a transformer

Transformers are a family of neural networks architectures.

Key contritributions:

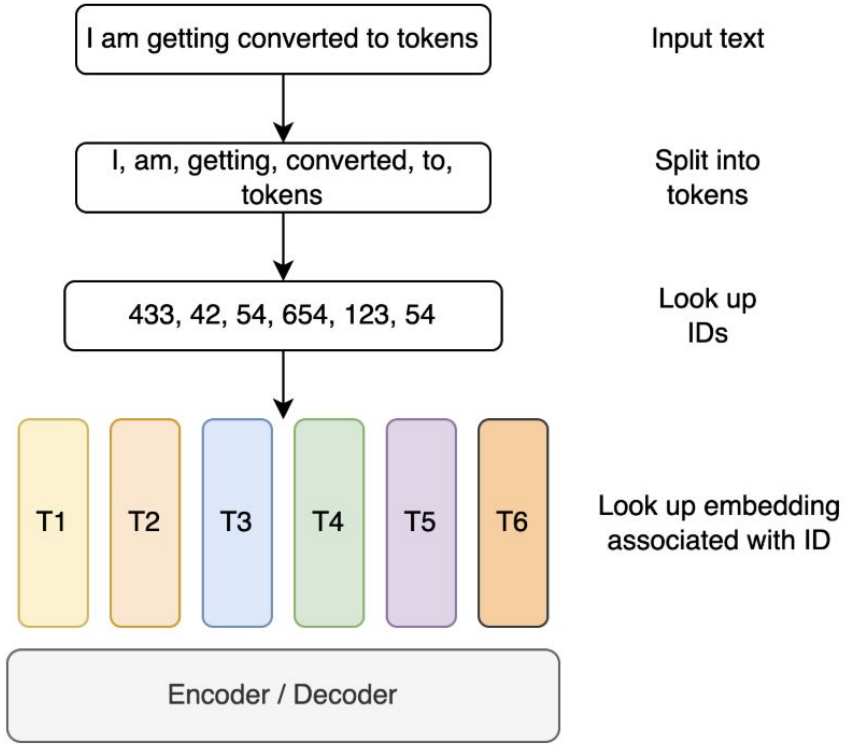
- Parallelization thanks to its non-sequential operations.
- Ability to use attention to capture information from long-range sequences.
- Positional encodings.



From text to embeddings

Text Preparation for LLMs: 3 Key Steps

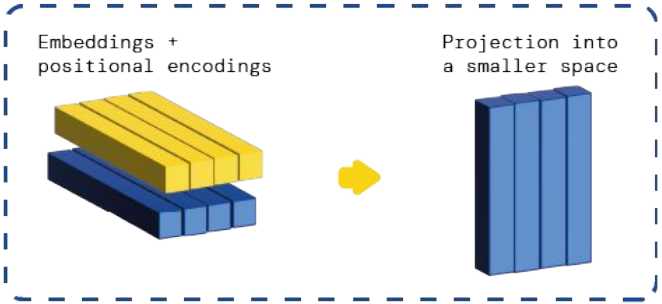
- **Tokenization:** Raw text is split into tokens, like words or subwords, creating the basic units for the LLM.
- **ID Lookup:** Each token is then mapped to a unique numerical ID from a predefined vocabulary, converting textual units into machine-readable numbers.
- **Embedding Generation:** These numerical IDs are used to retrieve corresponding vector representations (embeddings) from an embedding matrix, transforming discrete IDs into dense, meaningful vectors that capture semantic relationships.



Positional encodings

Sequencing problems require to have an understanding of the order of the sequence.

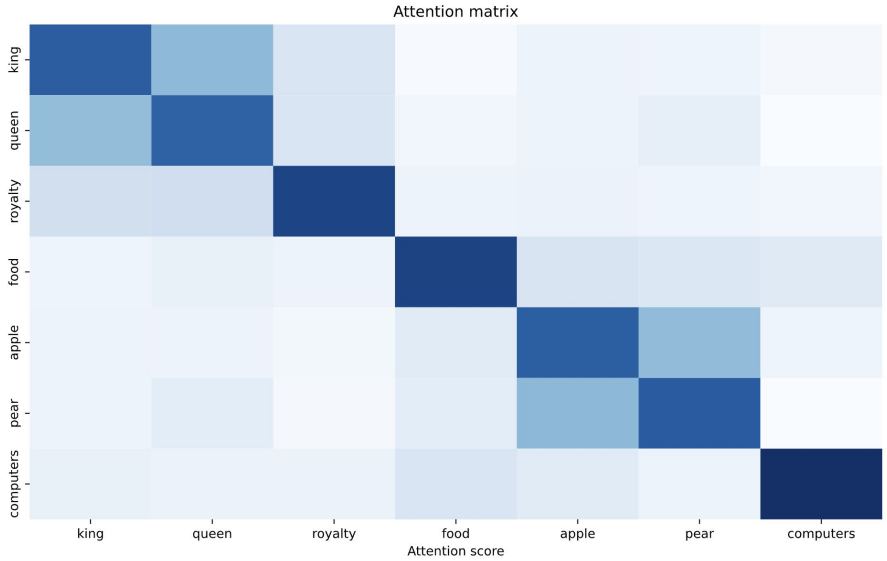
- Transformers encodes the position of every token seen in a fixed-size vector.
- For each word, a sinusoid is used to encode the unique position for each time step.
- Positional encodings are added to the word embedding vector before the first self-attention layer.
- Is completely deterministic.



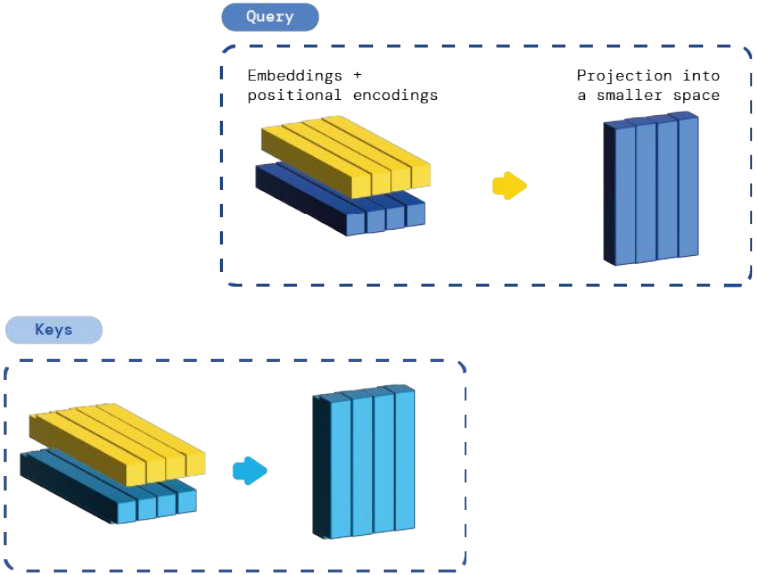
Understanding attention

Key Functions of Attention:

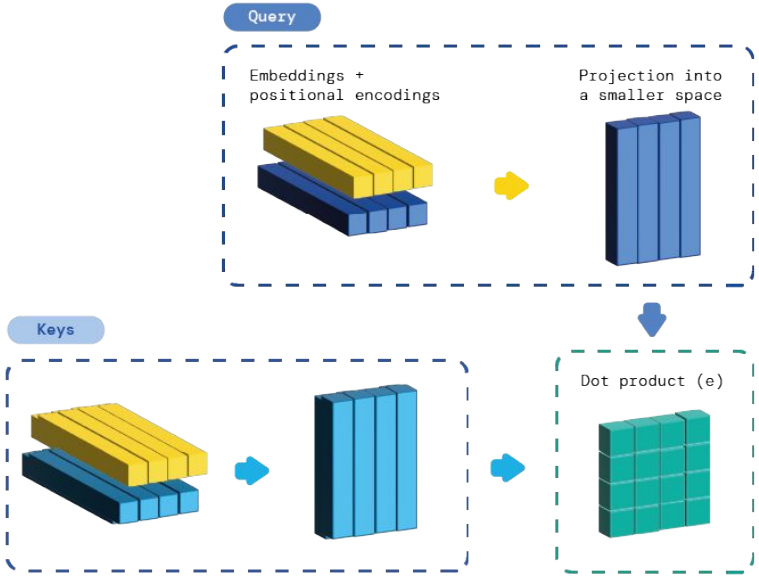
- **Selective Focus**: It allows the model to prioritize specific parts of the input.
- **Variable Importance**: It assigns different levels of importance to various elements.
- **Improved Understanding**: It enhances the model's ability to grasp complex relationships within the data.



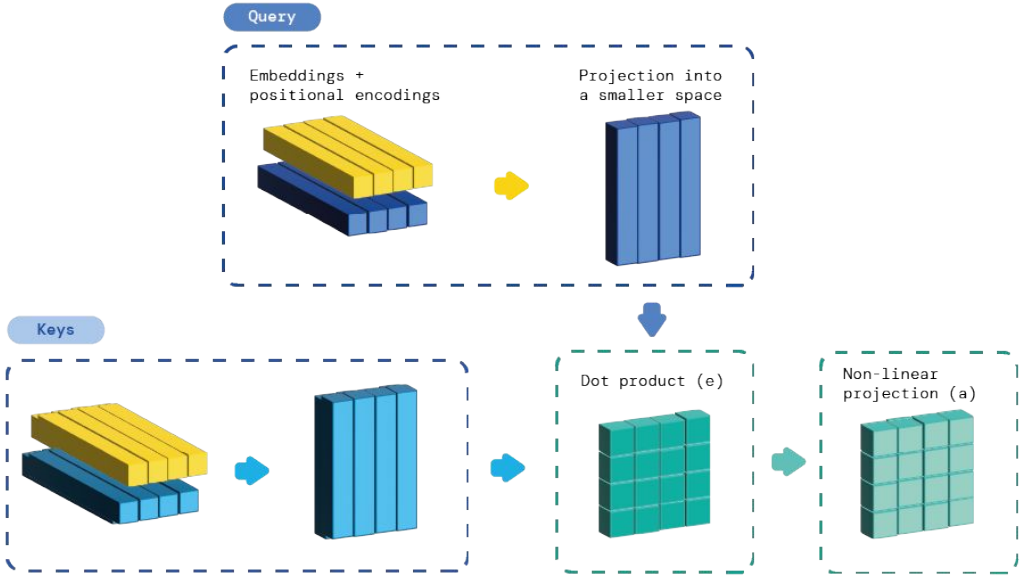
Attention mechanism



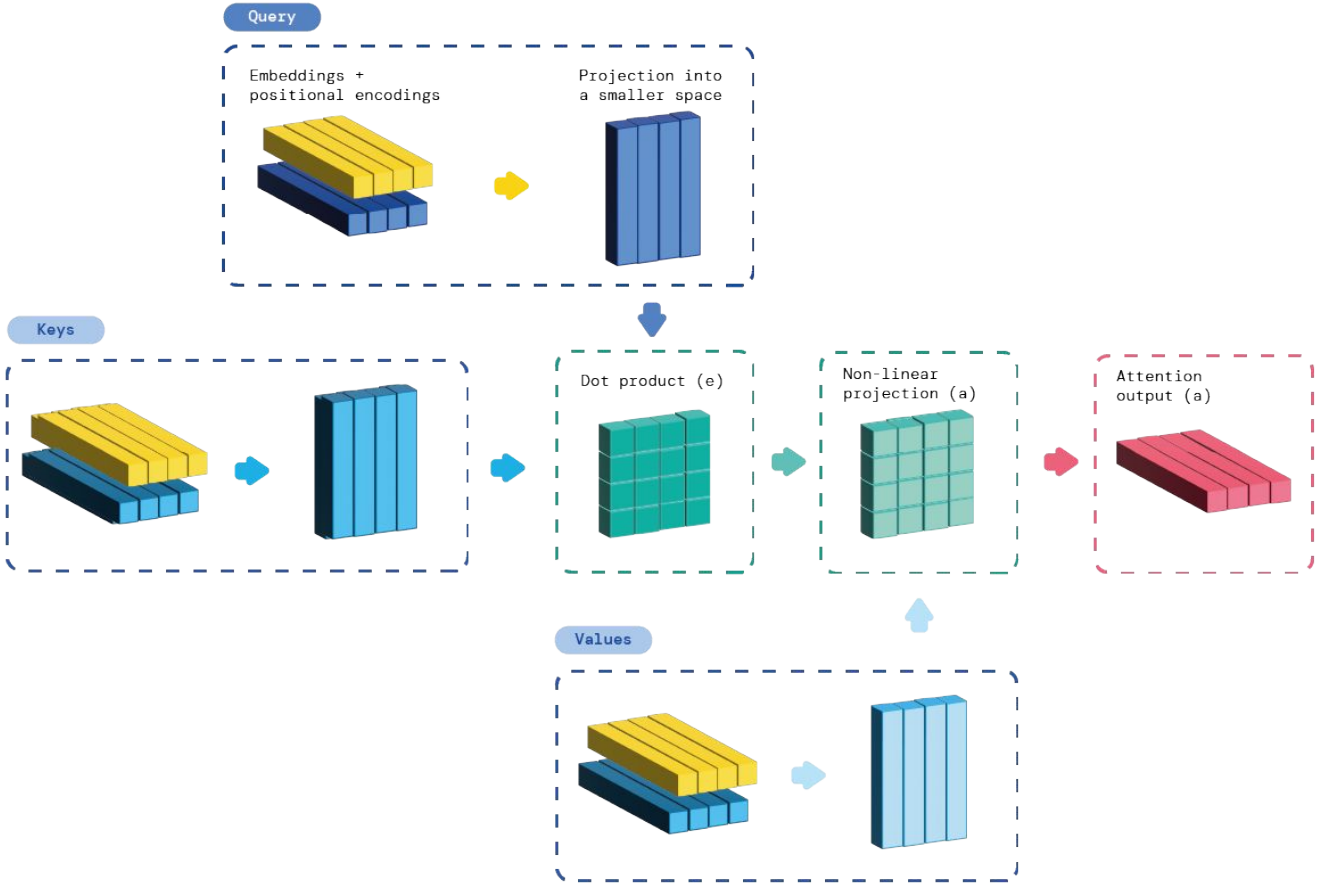
Attention mechanism



Attention mechanism



Attention mechanism

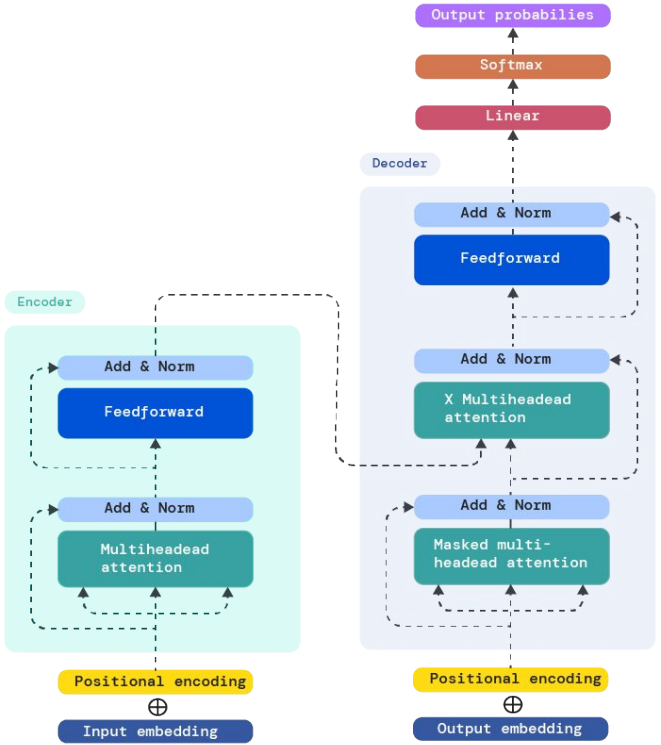


Transformers recap

Key features of an encoder-decoder transformers:

- **Encoder** branch with multiple layers containing:
  - Positional encodings.
  - Multi-headed self-attention.
  - Residual connections.
  - Normalization.
  - Feedforward.

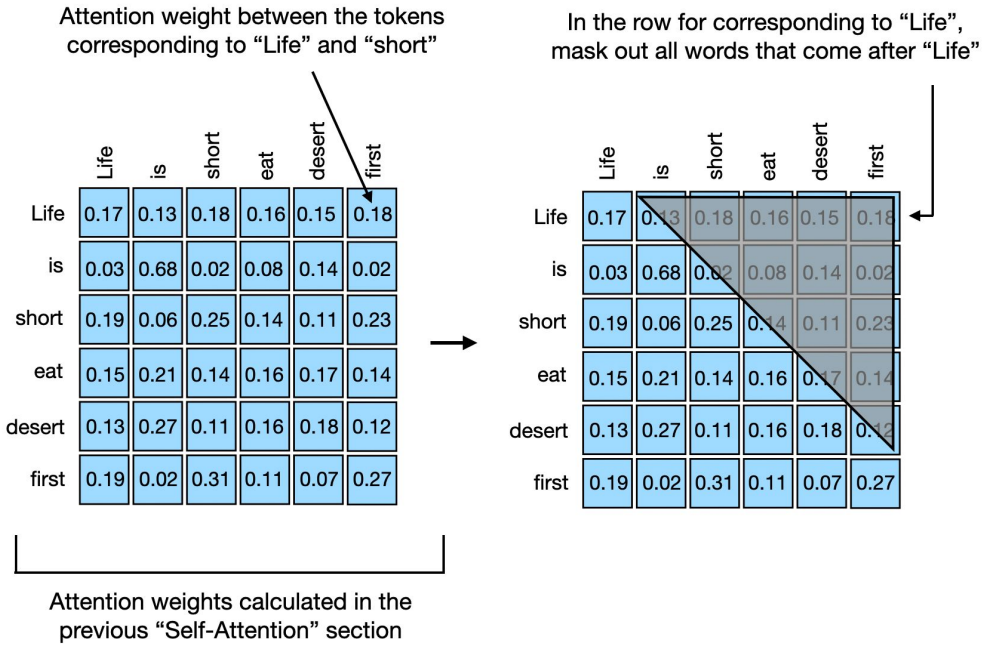
- **Decoder** branch with multiple layers containing:
  - Positional encodings.
  - Multi-headed cross-attention.
  - Residual connections.
  - Normalization.
  - Feedforward.



Language is causal

Causal Attention in LLMs:

- **Preserves Text Order:** Ensures the model generates text word-by-word, mirroring how we naturally speak and write.
- **Ensures Realistic Training:** Prevents the model from peeking at future information during training, guaranteeing that it learns to predict based on context, not future knowledge.
- **Powers Autoregressive Generation**: Each predicted word becomes the input for the next, allowing LLMs to create coherent and contextually relevant text.



https://magazine.sebastianraschka.com/p/understanding-and-coding-self-attention

Training objective

Transformer Loss: Measuring Prediction Accuracy

- Compares the model's predicted next words to the actual next words in the training data.
- Calculates how wrong the model's predictions are, giving a numerical measure of the prediction error.
- This loss value is used to adjust the model's parameters, guiding it towards making more accurate predictions.

$$\text{Loss}_t = -\sum_{w \in V} y_t \log(\hat{y_t})$$