# Transformers

ChatGPT

How do you greet an Italian lecturer?

To greet an Italian lecturer, you can use a polite and respectful greeting, especially if you're addressing them formally. Here are some appropriate ways to do so:

### In Italian:

- **"Buongiorno, professore"** (Good morning, professor) – if the lecturer is male.
- **"Buongiorno, professoressa"** (Good morning, professor) – if the lecturer is female.
- **"Buonasera, professore"** (Good evening, professor) – if it's later in the day and the lecturer is male.
- **"Buonasera, professoressa"** (Good evening, professor) – if it's later in the day and the lecturer is female.

# Transformers

- Let’s focus on language models
- Other relevant domains (vision transformer (ViT), sound and speech-transformers)

Key concepts:

- Backpropagation
- Feedforward vs. recurrent nets
- From LSTM to RNNs
- Seq2seq models
- Word embeddings, Attention (different types), and Positional encoding
- Encoder-decoder transformers
- Decoding-only transformers
- Fine-tuning (e.g., from GPT to chatGPT)
- Many many many implementations, performance tricks, etc

- …hey, is this a history class? Boring!
- Let’s have some fun instead

# Transformers

Challenges:

- This topic is a moving target
- Many possible implementations
- Performance and resources to train these models
- Many domains of application

Our focus?

- Let’s stay simple and learn the core elements that make this architecture important
- Intuitions + practical approach

# Transformers

Outline:

- Brief overview
- Let’s build a mini GPT from scratch (decoder-only transformer)
- Encoder-decoder transformers (intuition)
- Fine-tuning? From GPT to ChatGPT
- Examples

Advice

- It is crucial that you go through the architecture by yourselves
- Let’s take it one step at a time
- Don’t worry about performance and optimization for now. Let’s just make sure that the main steps of the architecture are clear
- Listen carefully in class, study the steps by yourselves, and run the code
- Remember that some architectural choices are important, others are just details

# Transformers overview

## Attention Is All You Need

**Ashish Vaswani*** Google Brain avaswani@google.com

**Noam Shazeer*** Google Brain noam@google.com

**Niki Parmar*** Google Research nikip@google.com

**Jakob Uszkoreit*** Google Research usz@google.com

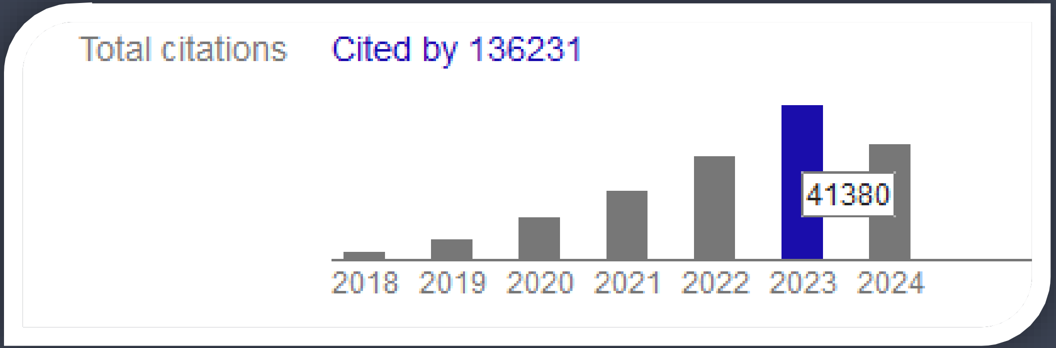
**Llion Jones*** Google Research llion@google.com

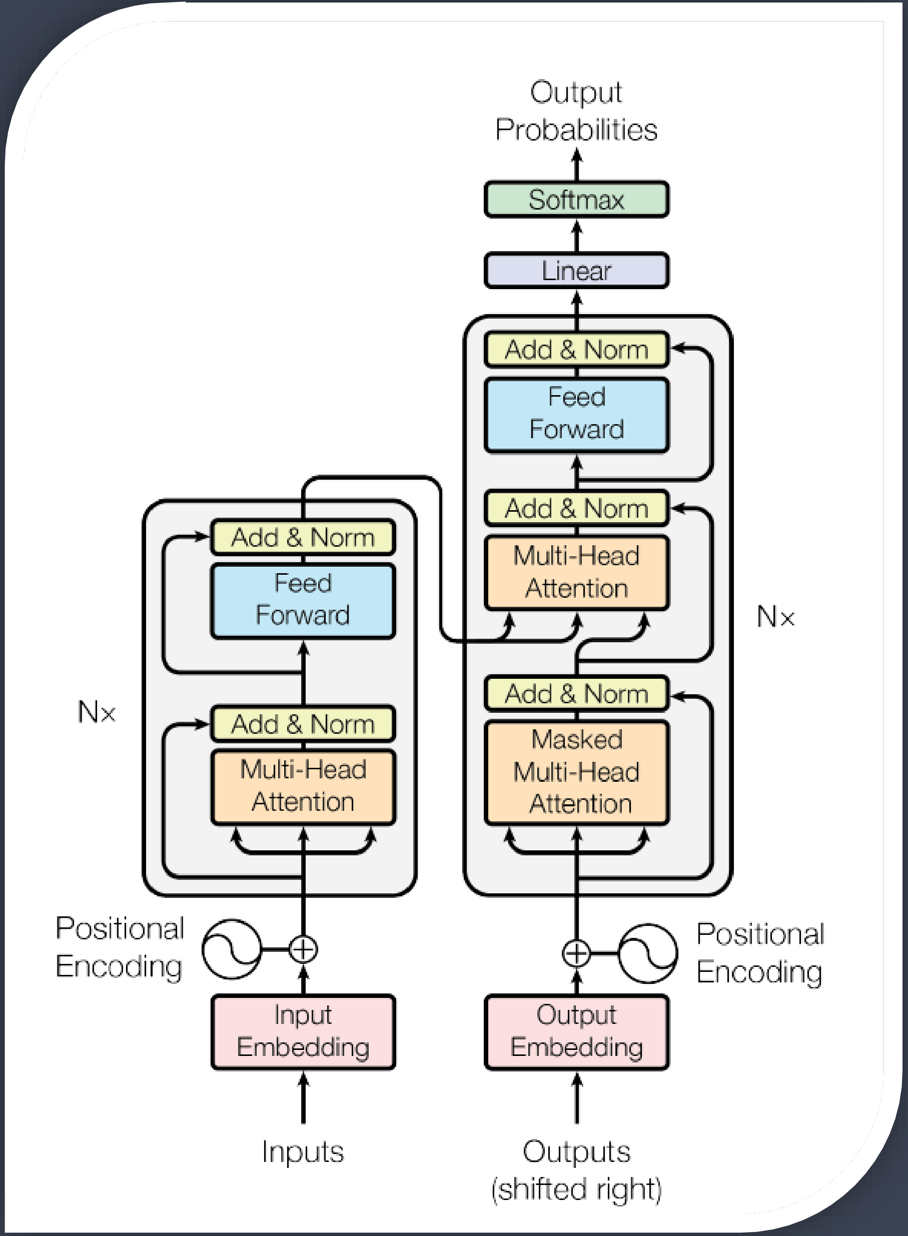
**Aidan N. Gomez*** † University of Toronto aidan@cs.toronto.edu

**Łukasz Kaiser*** Google Brain lukaszkaiser@google.com

**Illia Polosukhin*** ‡ illia.polosukhin@gmail.com

A breakthrough




https://scholar.google.com/citations?view_op=view_citation&hl=en&user=oR9sCGYAAAAJ&citation_for_view=oR9sCGYAAAAJ:zYLM7Y9cAGgC
https://arxiv.org/abs/1706.03762

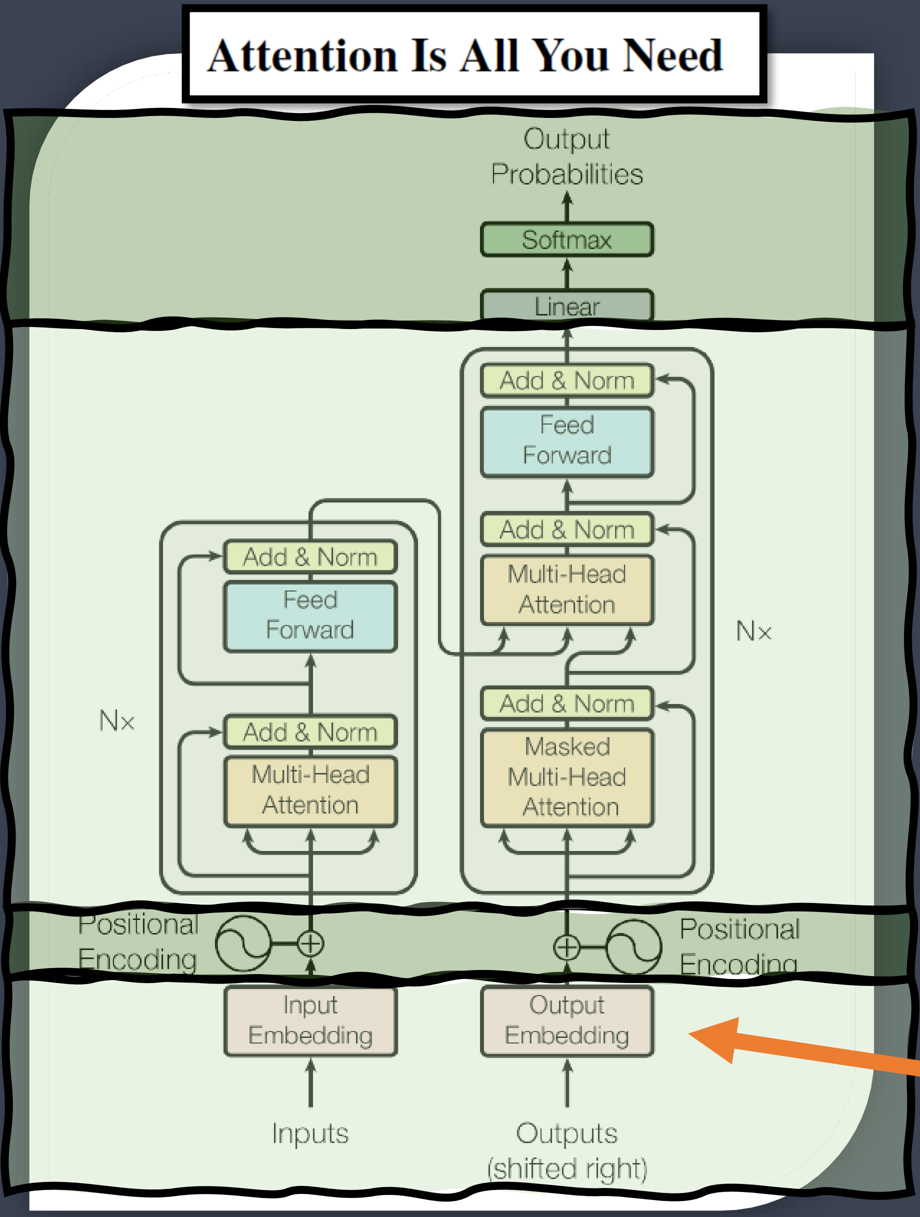
# Transformers

- Applications: Text translation, text summarization, sentiment analysis, generating text, and much more (e.g., text to images/music/sounds)
- Some breakthrough transformer models include BERT (applied for Google search in 2020) and GPT-3. There are also great open-source LLMs, like Mistral.

- Transformers are a type of neural network
- They were formulated to translate text from one language and another
- For example, translating English to Italian

Key concepts

- Self-Attention Mechanism: weigh the importance of different words in a sentence, allowing them to capture long-range dependencies.
- Applications:




https://arxiv.org/abs/1706.03762

English: That looks like a very complicated model!

Italian: Sembra un modello complicatissimo!

Model input:

- Input text: Full English sentence text
- Output text: Italian sentence text context (e.g., “Sembra un modello”)

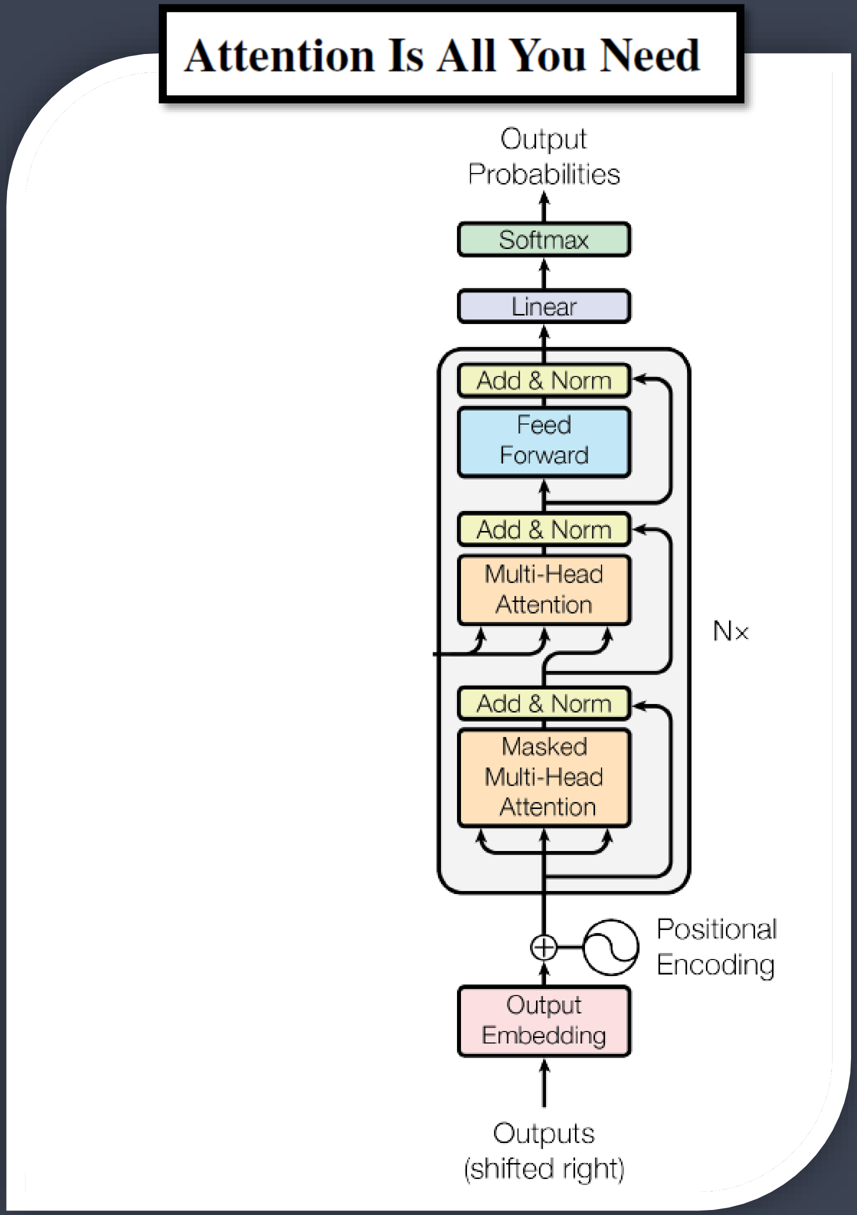
Model output:

- Output word: “complicatissimo”

For both model inputs:

- Text to embedding (to a vector of numerical values)
- Adding information on the position of the word in the sentence
- Sequence of Attention (different types) + Feed forward layers
- Fully connected layer (one output per token in vocabulary)

This could be the word embedding network, but in reverse (e.g., original GPT paper)




https://arxiv.org/abs/1706.03762

What if we only wanted text generation?
i.e., Generative Pretrained Transformer (GPT)!

Model input: text prompt
Model output: text continuation

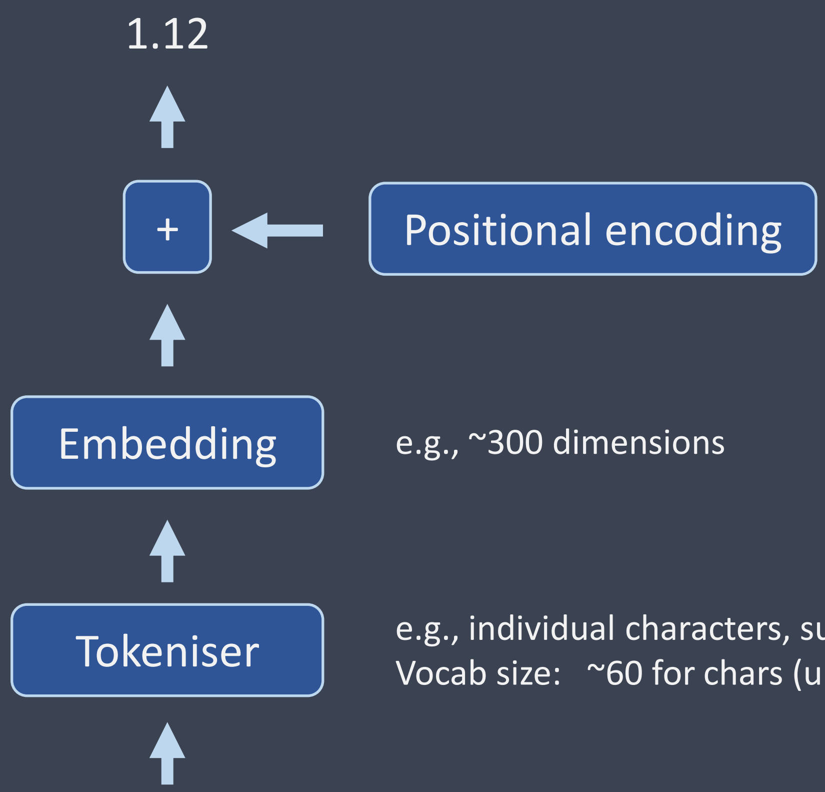
That's a Decoding-only transformer

# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding
- Key-step 2: Masked self-attention
- Key-step 3: Add residual connection
- Key-step 4: Next-word generation (Fully-connected layer + softmax)

# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding




Numerical output

“Sam hit the ball” vs. “The ball hit Sam” are very different. Knowing position is crucial!

Convert each word into numbers

e.g., ~300 dimensions

For simplicity, let’s use a word tokeniser

e.g., individual characters, subword particles, words
Vocab size: ~60 for chars (upper and lower cases); ~100,000 in GPT

**Input prompt:** “What is a transformer?”

# Decoder-only transformer

- <u>Key-step 1: Input prompt -> Word embedding + Positional encoding</u>

Let’s look only at the first word for now

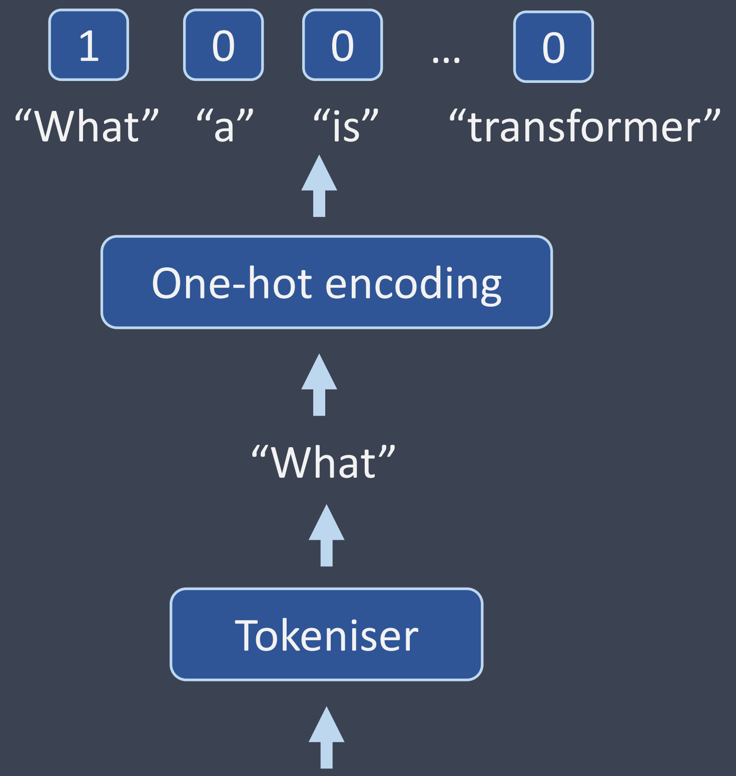
“What” “is” “a” “transformer?”

Tokeniser

**<u>Input prompt:</u>** “What is a transformer?”

# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding



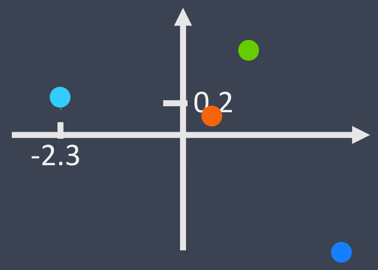

**Input prompt:** "What is a transformer?"

# Decoder-only transformer

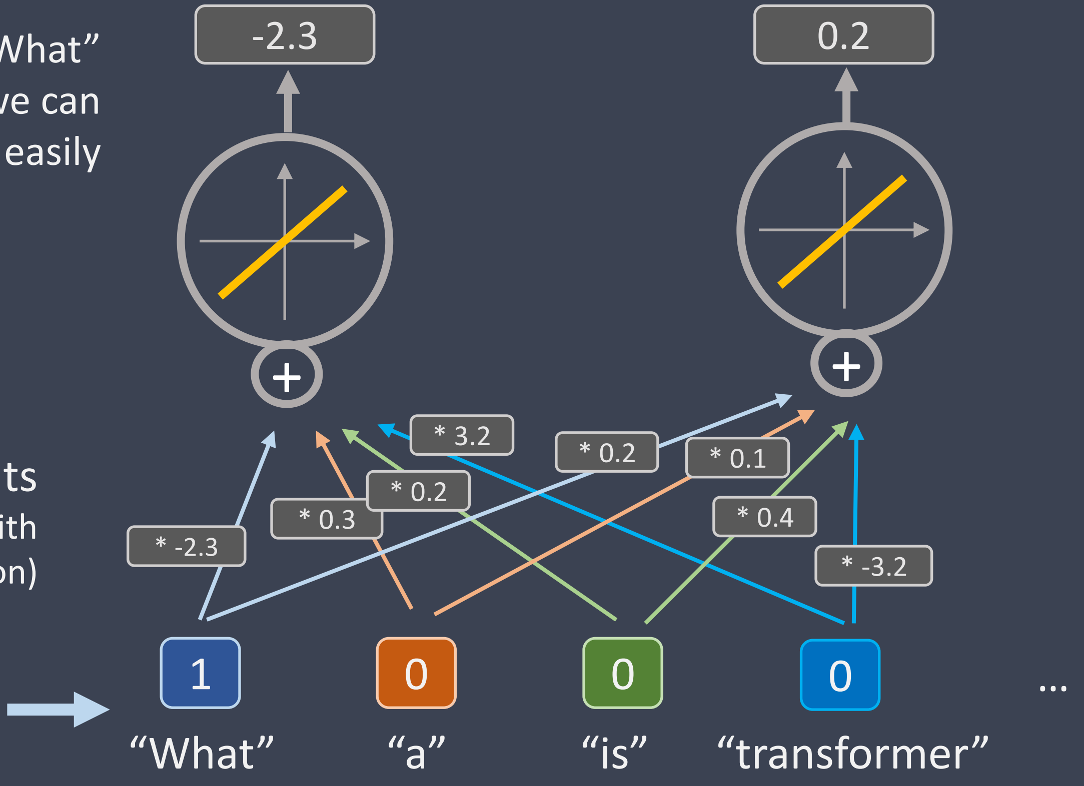
- Key-step 1: Input prompt -> Word embedding + Positional encoding

- The one hot encoding of the input is connected to many activation functions (two in the example). We could use 50, 100, 300 or more of these activation functions.

Embedding value for "What"
With 2 activation functions, we can plot the embedding easily




Weights
(trained or pretrained with backpropagation)




**Input token:** "What"

# Decoder-only transformer

- <u>Key-step 1: Input prompt -> Word embedding + Positional encoding</u>

**<u>Input token:</u>** “transformer” →

| 0 | 0 | 0 | … | 1 |
|---|---|---|---|---|
| “What” | “a” | “is” | | “transformer” |

**<u>Input token:</u>** “a” →

| 0 | 1 | 0 | … | 0 |
|---|---|---|---|---|
| “What” | “a” | “is” | | “transformer” |

**<u>Input token:</u>** “is” →

| 0 | 0 | 1 | … | 0 |
|---|---|---|---|---|
| “What” | “a” | “is” | | “transformer” |

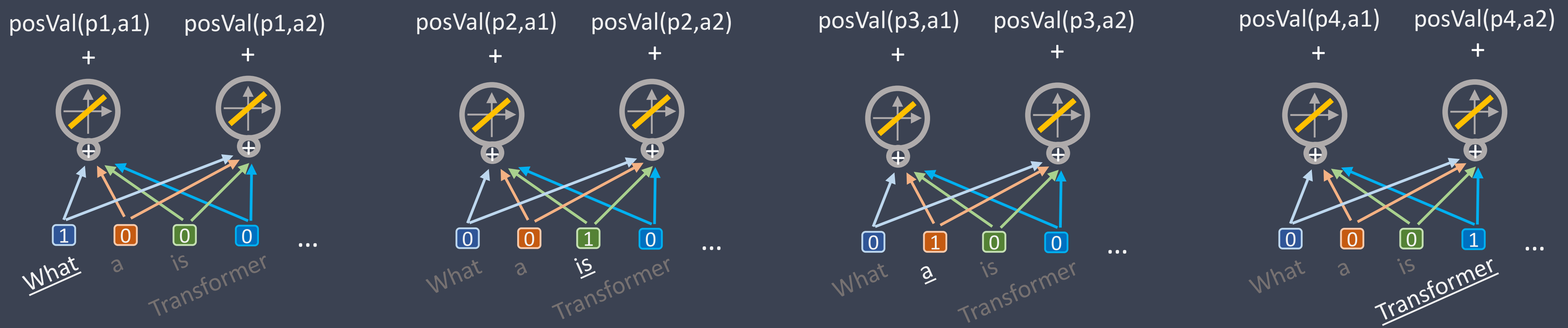
**<u>Input token:</u>** “What” →

| 1 | 0 | 0 | … | 0 |
|---|---|---|---|---|
| “What” | “a” | “is” | | “transformer” |

# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding

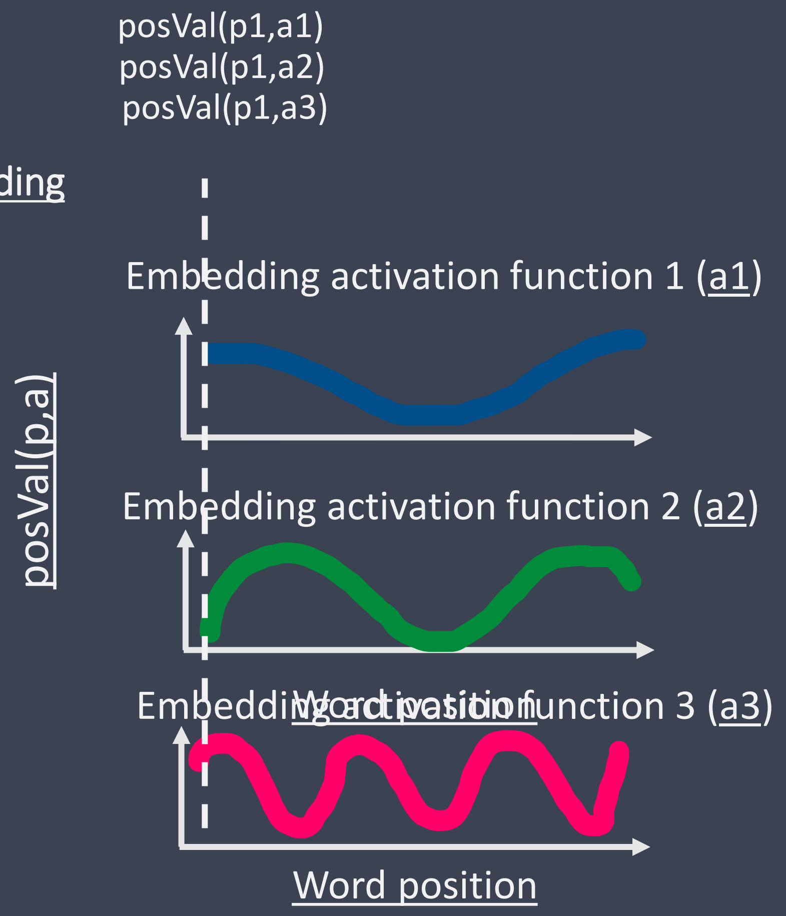
- Keep track of word order matters!
- “Sam hit the ball” vs. “The ball hit Sam” are very different
- Positional encoding
- posVal(p1,a1) stands for Position_value(position1, activation1)
- But what are these numbers?
- We could just give 1, 2, 3, 4, etc to indicate the position. However, that has many issues e.g., large numbers for large sequences might lead to scaling problems; same value for all embeddings, so less flexibility in learning; can’t capture periodicity of input; integers would be specific to input sequence, so they would not generalise to sequences of any length
- posVal can be a learnable table (i.e., tensor) of weights (see implementation that follows)

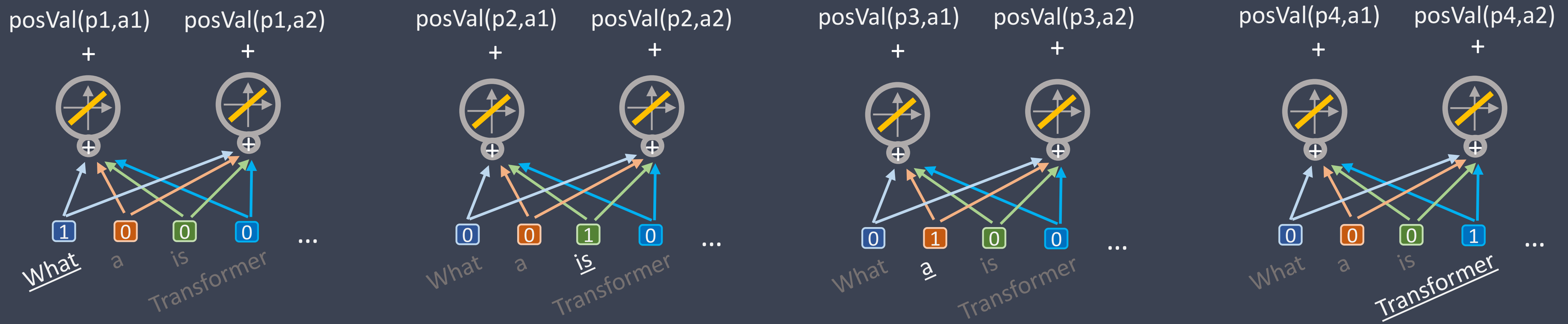


# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding

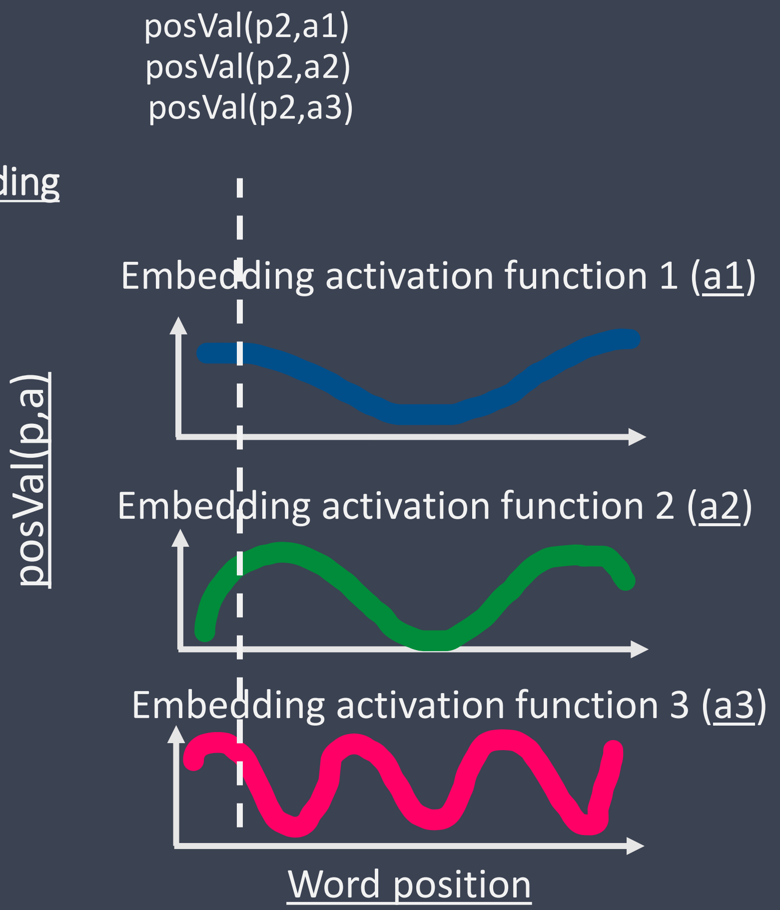
- A popular approach is to use these sin/cos functions that can capture regularities at different timescales
- Values on a single dimensions repeat, but the use of many embedding dimensions makes the position encoding unique in practice, as the same combination won't repeat within the limited length of a prompt
- Typically more than two activation functions i.e., embedding dimension



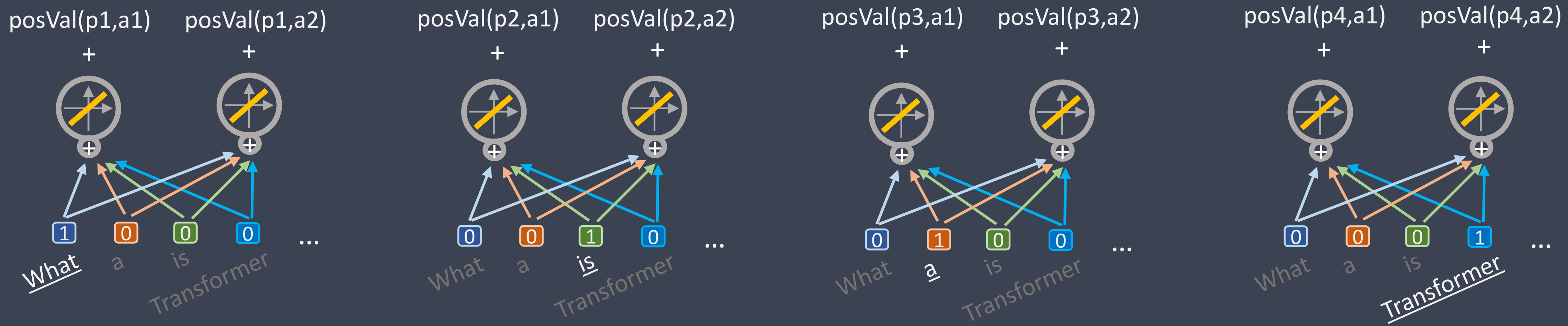



# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding

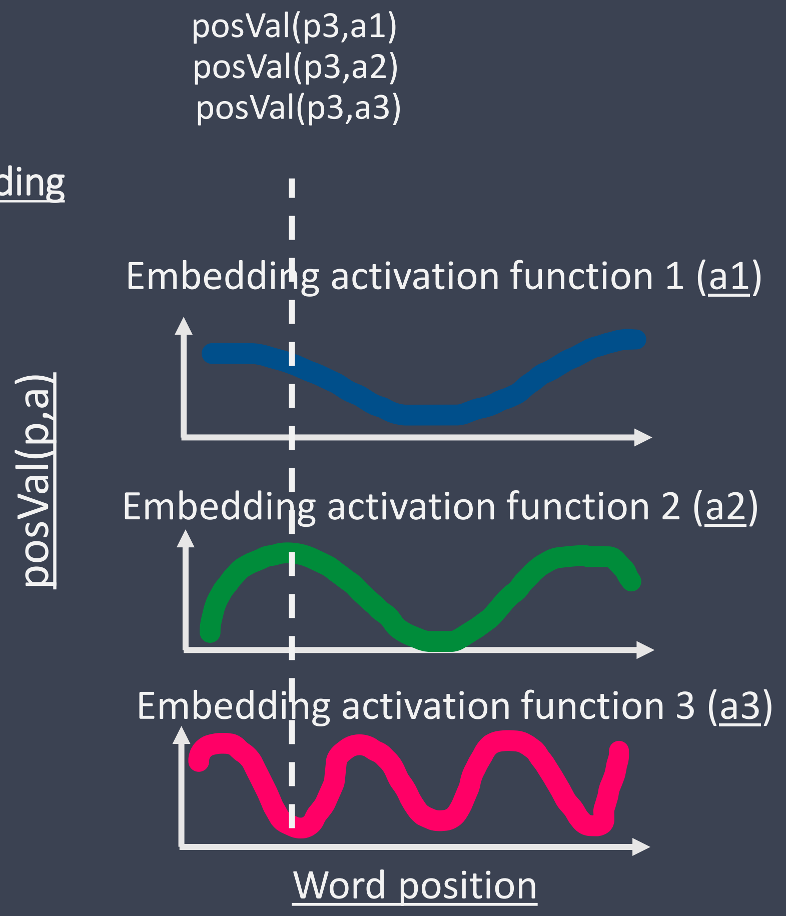



- A popular approach is to use these sin/cos functions that can capture regularities at different timescales
- Values on a single dimensions repeat, but the use of many embedding dimensions makes the position encoding unique in practice, as the same combination won't repeat within the limited length of a prompt
- Typically more than two activation functions i.e., embedding dimension

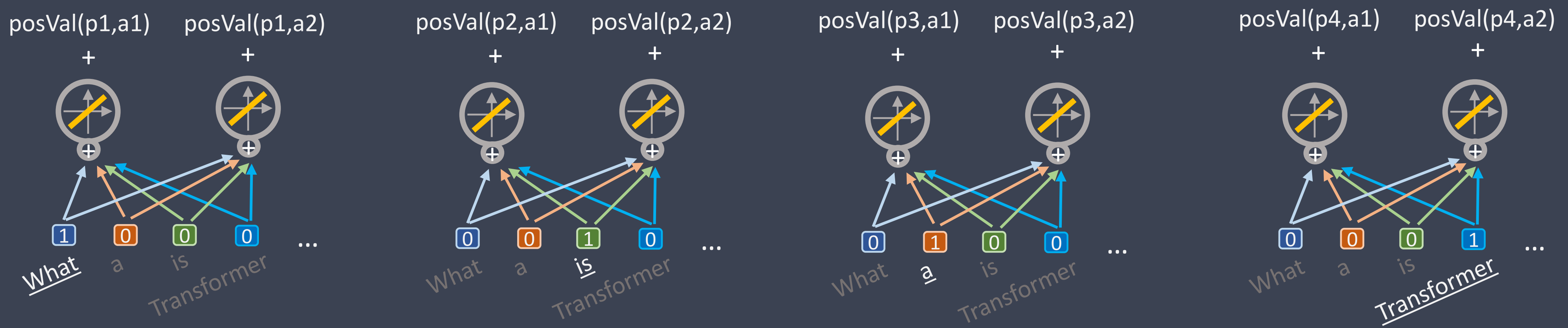


# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding



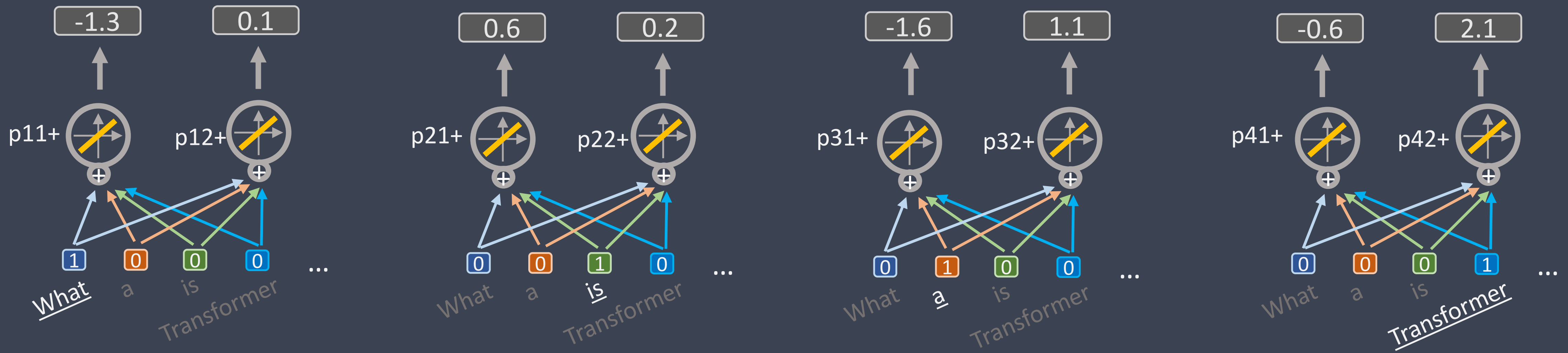

- A popular approach is to use these sin/cos functions that can capture regularities at different timescales
- Values on a single dimensions repeat, but the use of many embedding dimensions makes the position encoding unique in practice, as the same combination won't repeat within the limited length of a prompt
- Typically more than two activation functions i.e., embedding dimension
- This is a fixed mapping (rather than a learnable table of weights)



# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding

- The output of this first part consists of position-encoded embeddings
- Each word is associated with a vector of position-encoded values, with as many values as activation functions (i.e. same as embedding dimensions)
- Let's draw this in a compact way now
- How do we get those numbers? At init, they can be random values and then they get trained via backpropagation. Or one could start with a pre-trained embedding.

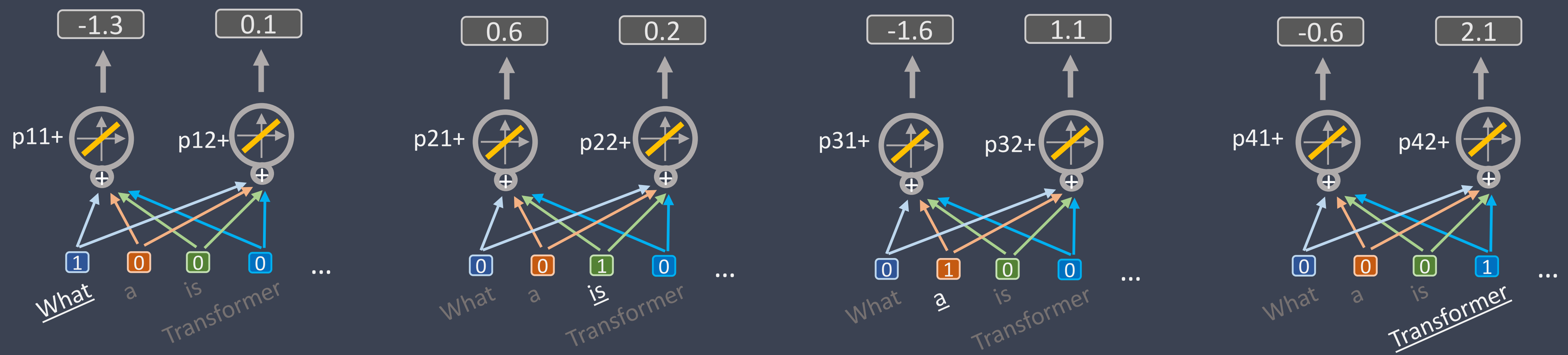


# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding

- Before moving to the self-attention mechanism, let’s look at the code for this part
- Check out full code on Blackboard (gpt_mini_text.py)

- Script based on nanoGPT (https://github.com/karpathy/nanoGPT/tree/master)
- Check out Andrej Karpathy’s Zero-to-hero tutorial-based course (https://karpathy.ai/zero-to-hero.html)



# Decoder-only transformer

- <u>Key-step 1: Input prompt -> Word embedding + Positional encoding</u>

- Before moving to the self-attention mechanism, let's look at the code for this part

```
# Load input file
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()
    # If we wanted to build a dummy model, we could shuffle the text for example
    # Or just use random weights (stop at random init)
    if toShuffleInput:
        text = ''.join(random.sample(text, len(text))) # shuffled text
```

Input: usually a **<u>large</u>** corpus of text

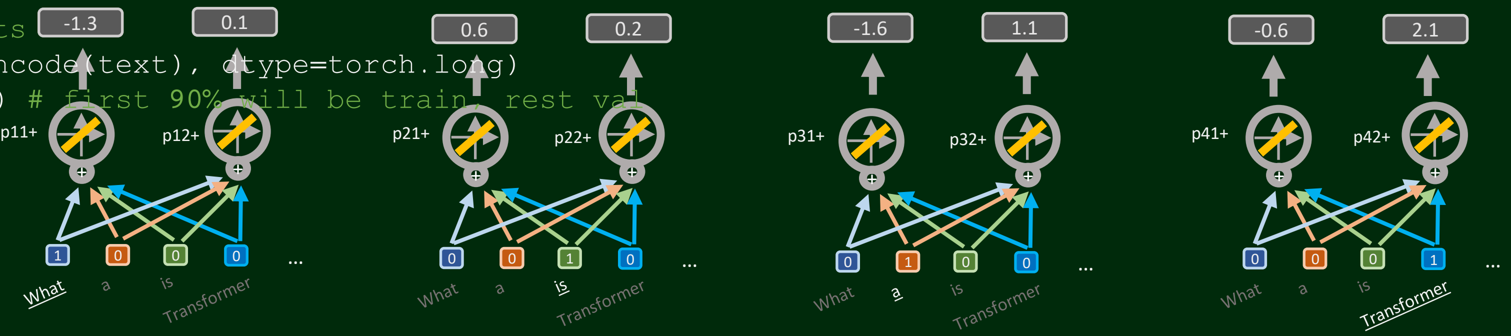
Here

- input.txt: Shakespeare corpus
- inputTheOldMan.txt: The old man and the sea book (Ernest Hemingway)
- You can try with your own input, of course. e.g., different language, style

```
# here are all the unique characters that occur in this text
chars = sorted(list(set(text)))
vocab_size = len(chars)
print(f"vocab_size: {vocab_size}")

# create a mapping from characters to integers
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s] # encoder: take a string, output a list of integers
decode = lambda l: ''.join([itos[i] for i in l]) # decoder: take a list of integers, output a string
```

For simplicity, this code uses characters as tokens, rather than words or subwords. This means that we have a much smaller vocabulary
chars is ['a', 'b', 'c', …, 'A', 'B', 'C', …, '\n', ' ', '.', …]
which is sufficient for our goal here
Trade-off between vocab_size and block_size (sequence length)

```
# Train and test splits
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9*len(data)) # first 90% will be train, rest val
train_data = data[:n]
val_data = data[n:]
```



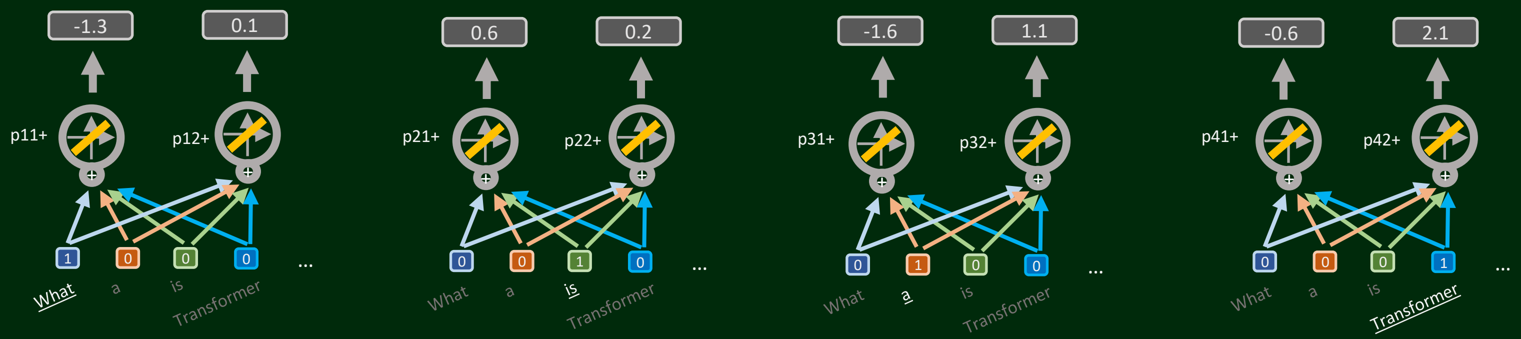
# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding

- Before moving to the self-attention mechanism, let's code this part

```
# For example
data = torch.tensor(encode("He was an old man"), dtype=torch.long)
print(data)                    # output: tensor([21, 44,  1, 62, 40, 58,  1, 40, 53,  1, 54, 51, 43,  1, 52, 40, 53])
                               # This is the kind of input that our model will receive

# data loading
def get_batch(split):
    # generate a small batch of data of inputs x and targets y
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y
```

- This function loads data (starting from a random index), returning x and y
- x is a random segment of the input text
- y is a shifted version of x (1 position shift)
- (This shifting approach is typical when dealing with time-series – recall our lecture on forecasting)



# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding

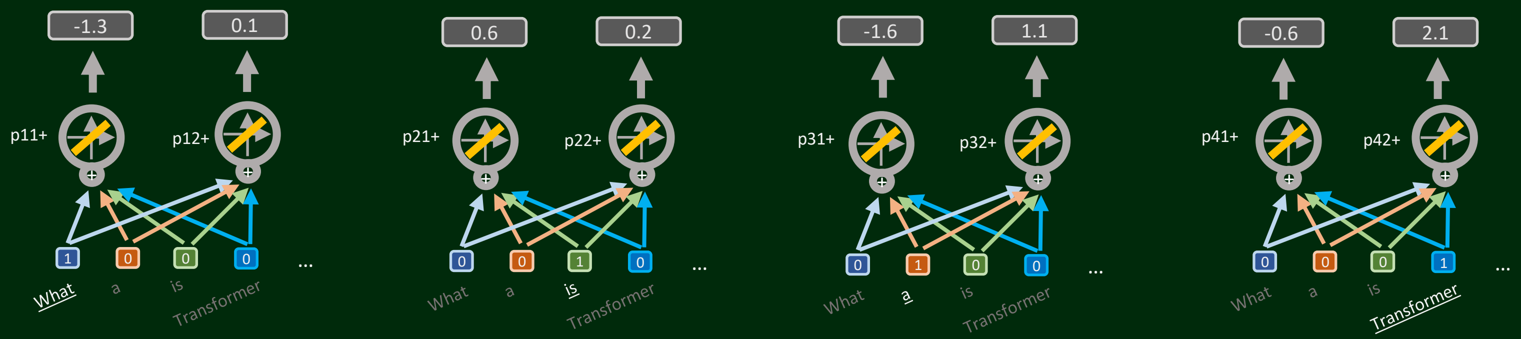
- Let’s build the language model class

- nn.module: base class for neural network modules in PyTorch.

```
class GPTLanguageModel(nn.Module):
    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)
        # better init, not covered in the original GPT video, but important
        self.apply(self._init_weights)

    def _init_weights(self, module):
        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
```

- Embedding tensor (table of weights)
- Positional tensor
- (The example below uses n_embd=2)

- Weights initialised randomly (normal distribution)
- Bias values initialised to zero



# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding

- Main script. Same as for other nnets (load data, backpropagation)

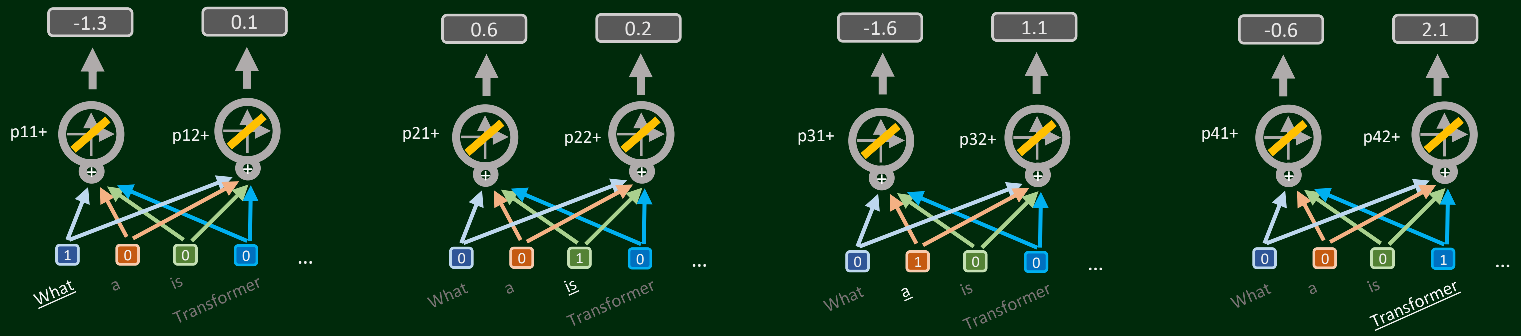
```
model = GPTLanguageModel() # create language model object
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')

# create a PyTorch optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

for iter in range(max_iters):
    # sample a batch of data
    xb, yb = get_batch('train')

    # evaluate the loss
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()
```

The main code

- Initialise GPTLanguageModel object
- Assign computation to cpu or gpu
- Assign optimizer

For each iteration of the training

- Get batch of data (xb and yb). Remember, yb is a shifted version of xb
- Calculate model logits and loss (see next slide)
- Clear previous gradients
- Calculate gradients (backpropagation)
- Parameter update



# Decoder-only transformer

- Key-step 1: Input prompt -> Word embedding + Positional encoding

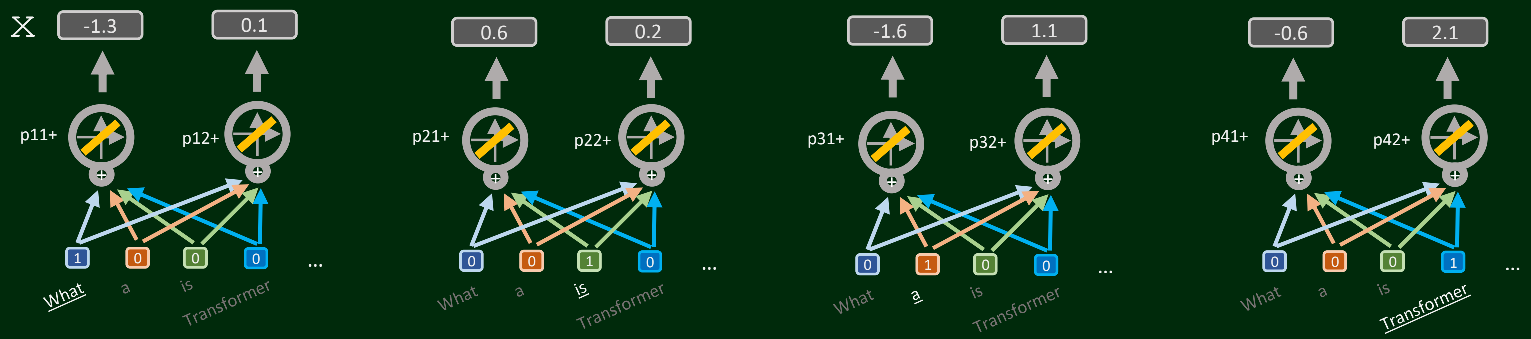
- Let's build the model in the figure below

```
logits, loss = model(xb, yb)
# This line of code (from the previous slide) calls the following function

class GPTLanguageModel(nn.Module):
   def forward(self, idx, targets=None):
        B, T = idx.shape # idx and targets (xb and yb) are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        B, T, C = logits.shape
        logits = logits.view(B*T, C)
        targets = targets.view(B*T)
        loss = F.cross_entropy(logits, targets)

        return logits, loss
```

B: batch; T: time/number of tokens/block-size C: embedding dims

Building our decoder transformer network

- Build embedding table (we had defined it as *nn.Embedding(vocab_size, n_embd)*
- Build positional encoding table *nn.Embedding(block_size, n_embd)*
- *Here, learnable weights. These could also be fixed (e.g., sin/cos functions)*
- Sum token and position embedding
- What's next? Attention

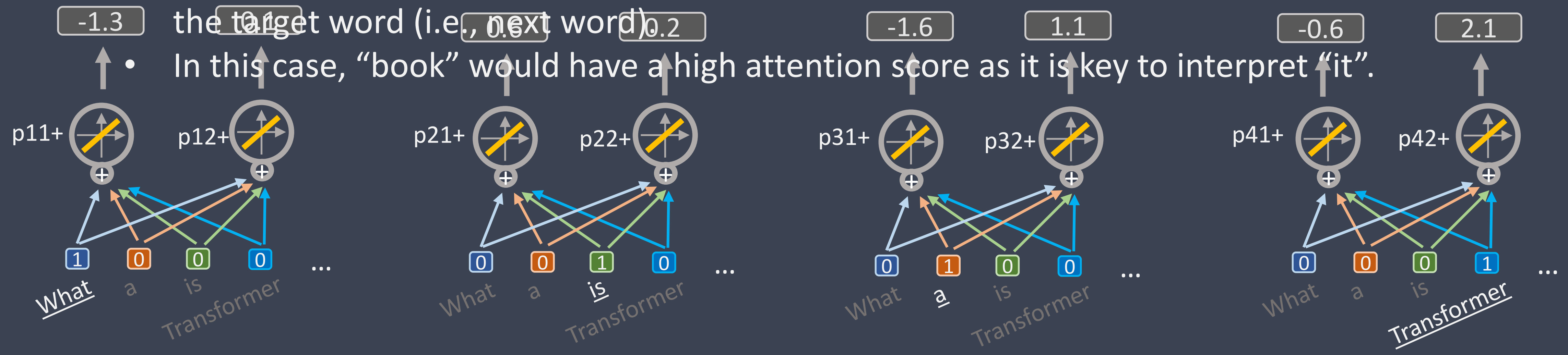


# Decoder-only transformer

Key-step 2: Masked self-attention

- So, we have an encoding of each word that accounts for their position.
- But we also need to account for the relationship between tokens, as the next token depends on the preceding context
- For example: The book was on the table, and it was very interesting.

- The mechanism building these associations is called **Self-attention**
- For example, when "it" is our target word, the context would be "The book was on the table, and". Self-attention tells us which of those previous words are most important for predicting the target word (i.e., next word).
- In this case, "book" would have a high attention score as it is key to interpret "it".



# Decoder-only transformer

Key-step 2: Masked self-attention

The book was on the table, and it was very interesting.

# Decoder-only transformer

Key-step 2: Masked self-attention

The book was on the table, and it was very interesting.

# Decoder-only transformer

Key-step 2: Masked self-attention

The book was on the table, and it was very interesting.

# Decoder-only transformer

Key-step 2: Masked self-attention

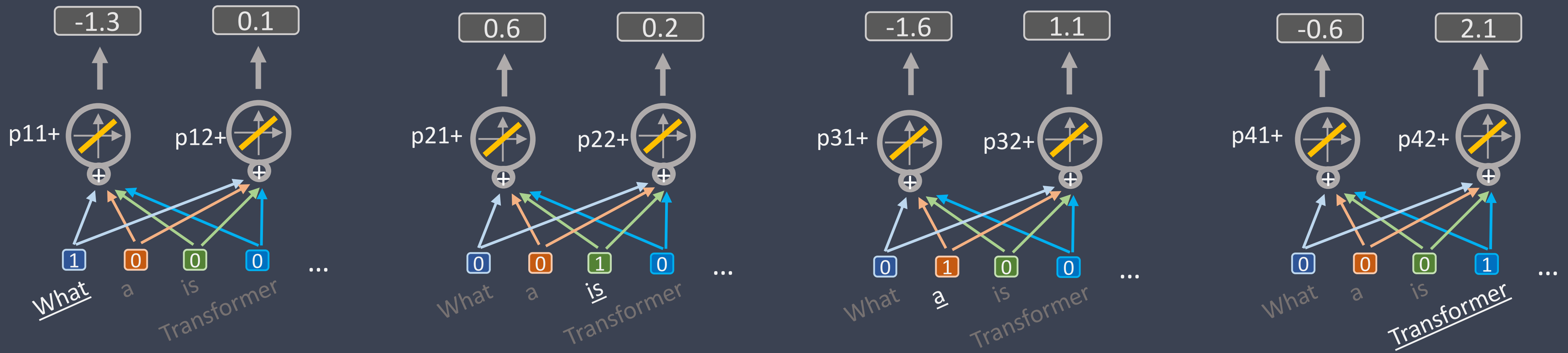
The book was on the table, and it was very interesting.

- GPT uses Masked Self-Attention, where each word can only be informed by the preceding context (and not future words)
- This is useful as it enables generation of new content (the G in GPT) (check out “autoregressive methods”)
- Note that this is compatible with human experience (e.g., speech; but note that reading is different)
- Note that this not the case for all text-based transformers (e.g., BERT)

# Decoder-only transformer

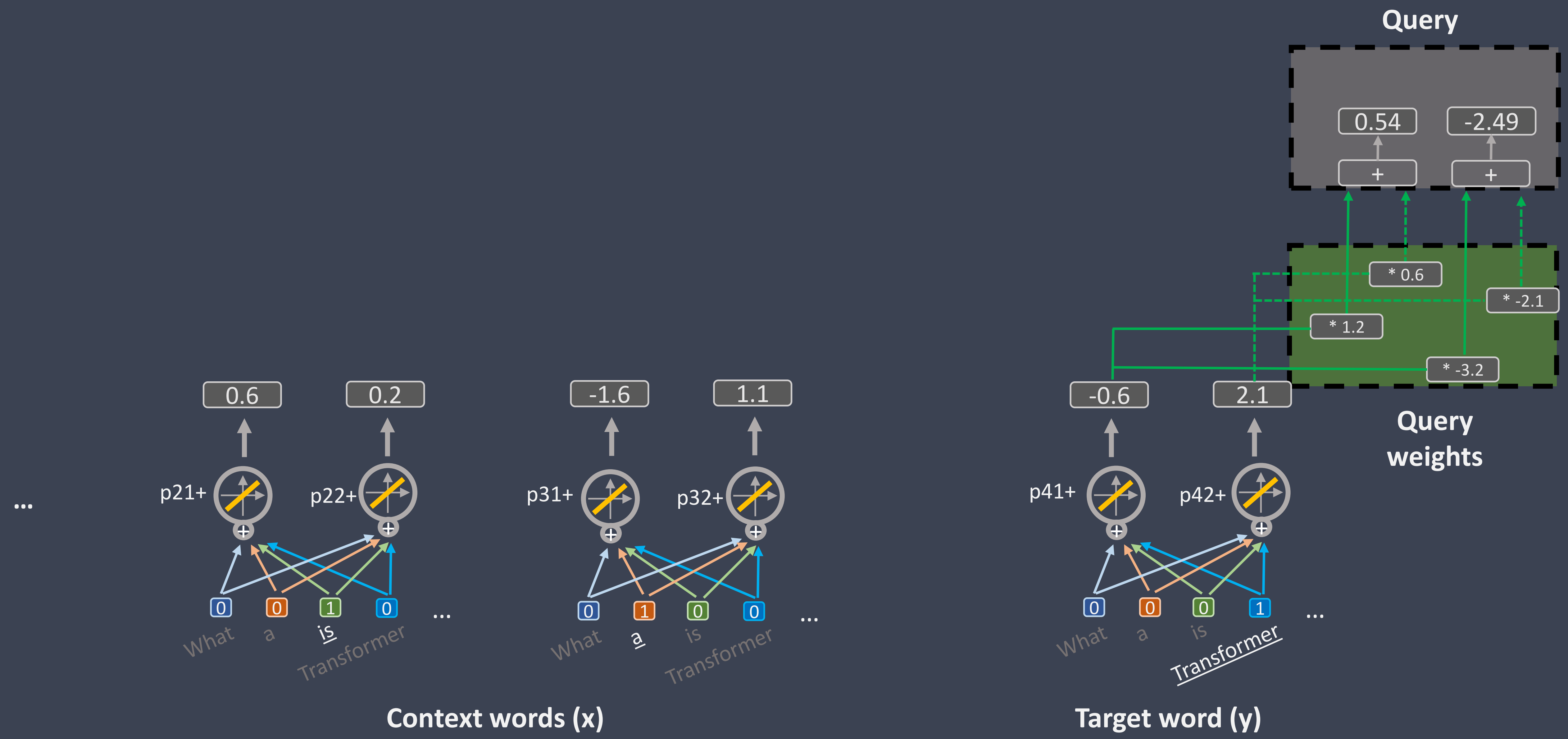
Key-step 2: Masked self-attention

- So, back to the model we built so far
- We want to derive numerical values describing
  - The isolated word embedding
  - That also accounts for its position (position encoding)
  - And it is informed by the preceding context (masked self-attention)



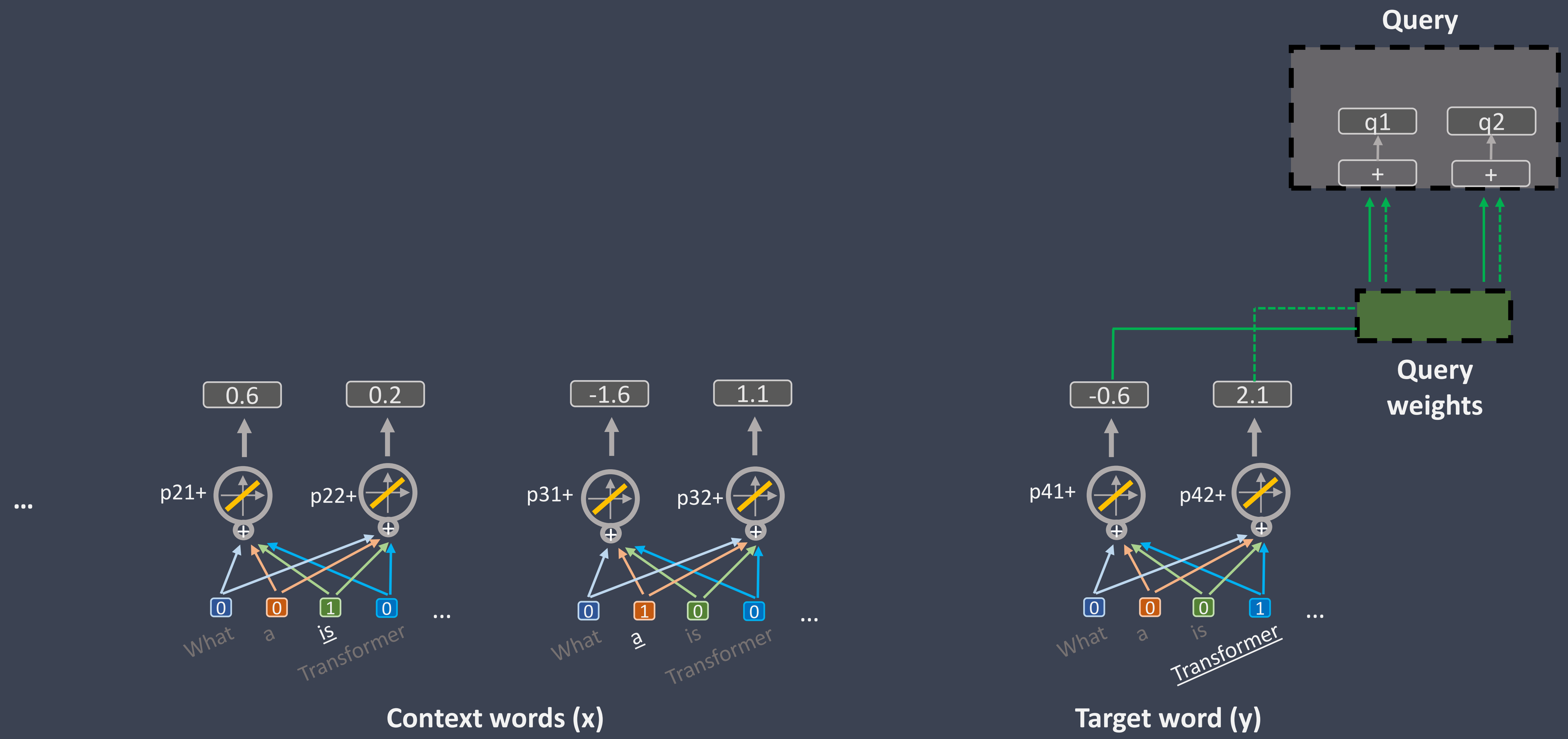
# Decoder-only transformer

Key-step 2: Masked self-attention



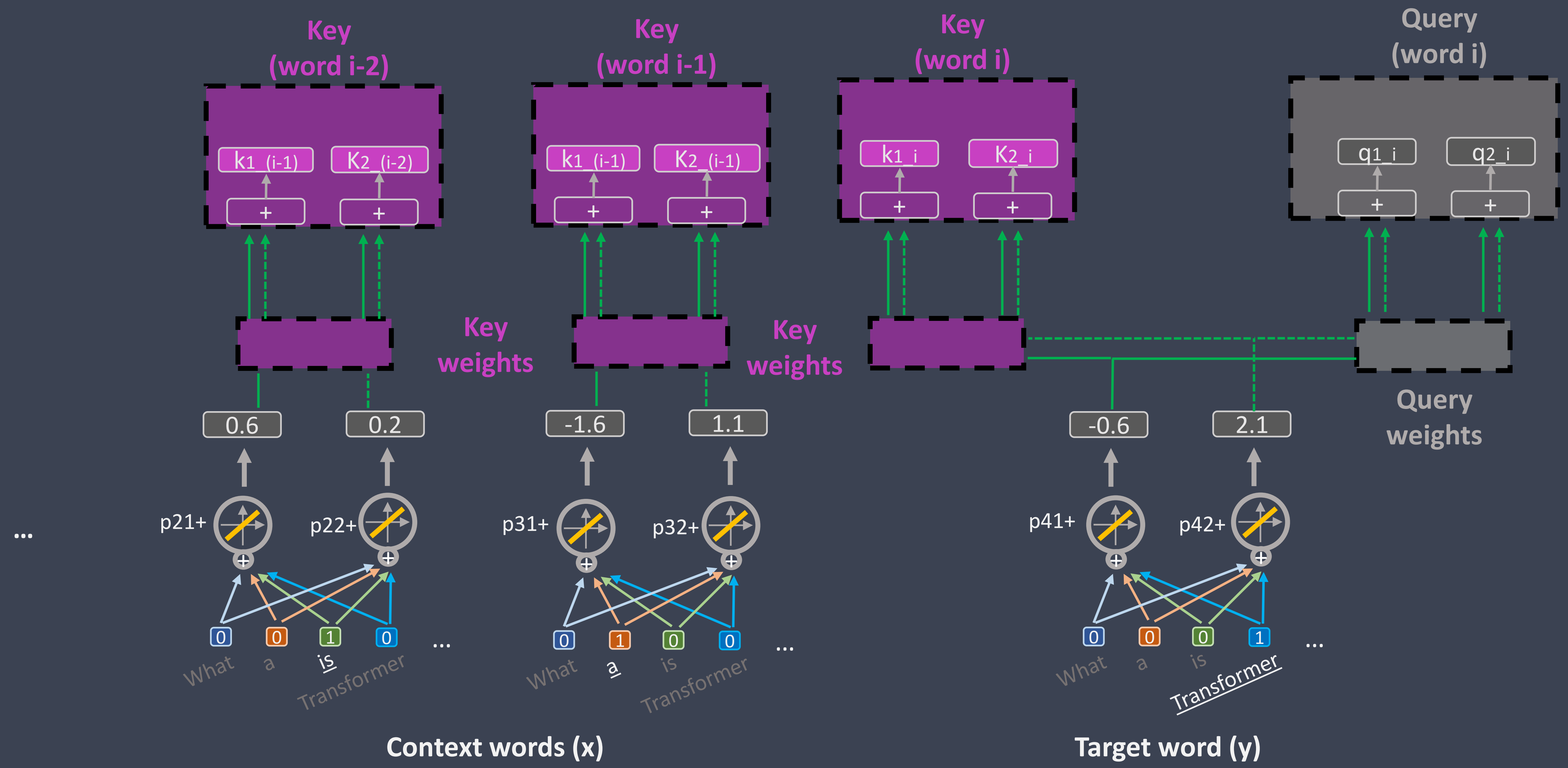
# Decoder-only transformer

Key-step 2: Masked self-attention



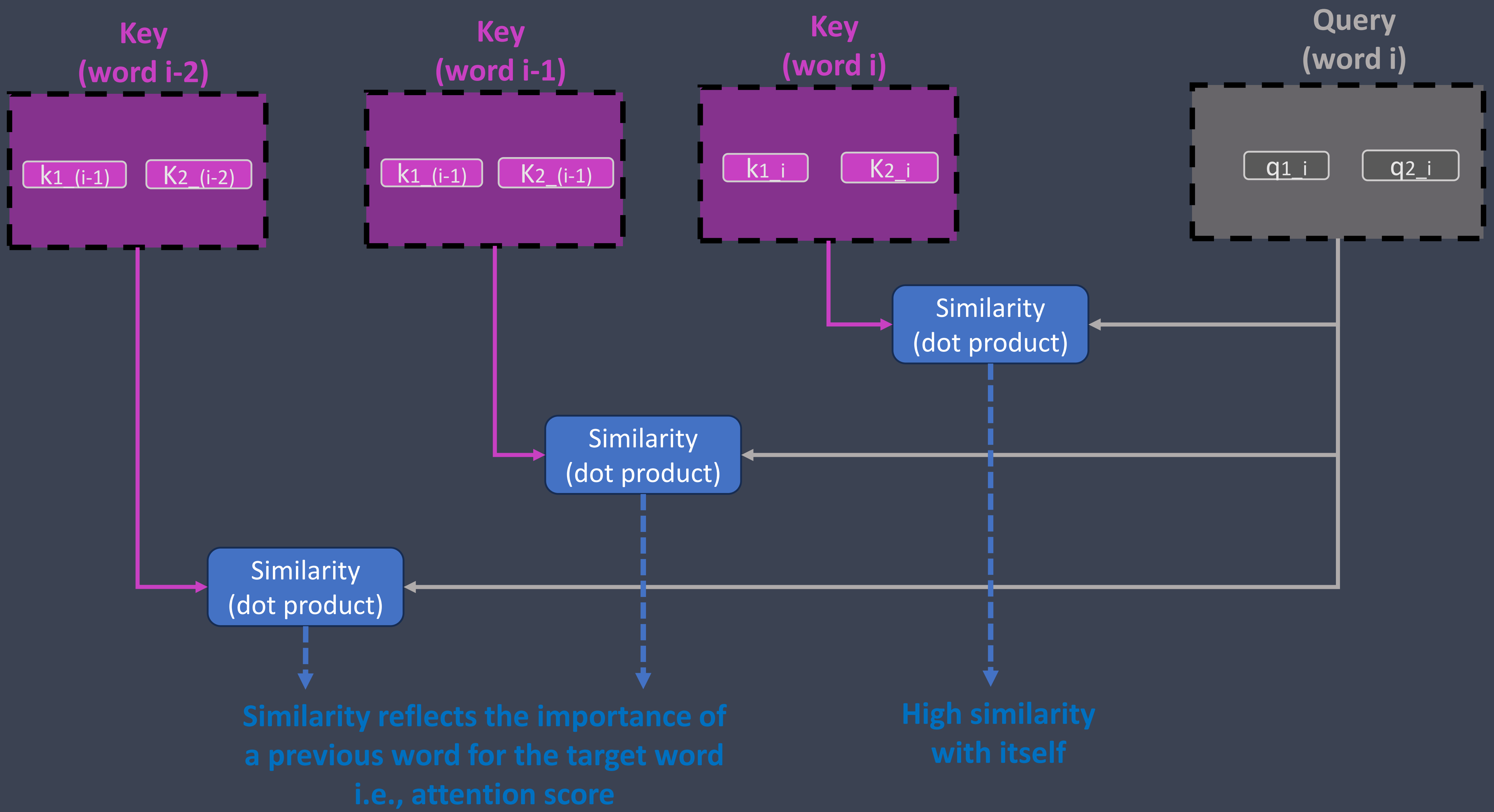
# Decoder-only transformer

Key-step 2: Masked self-attention



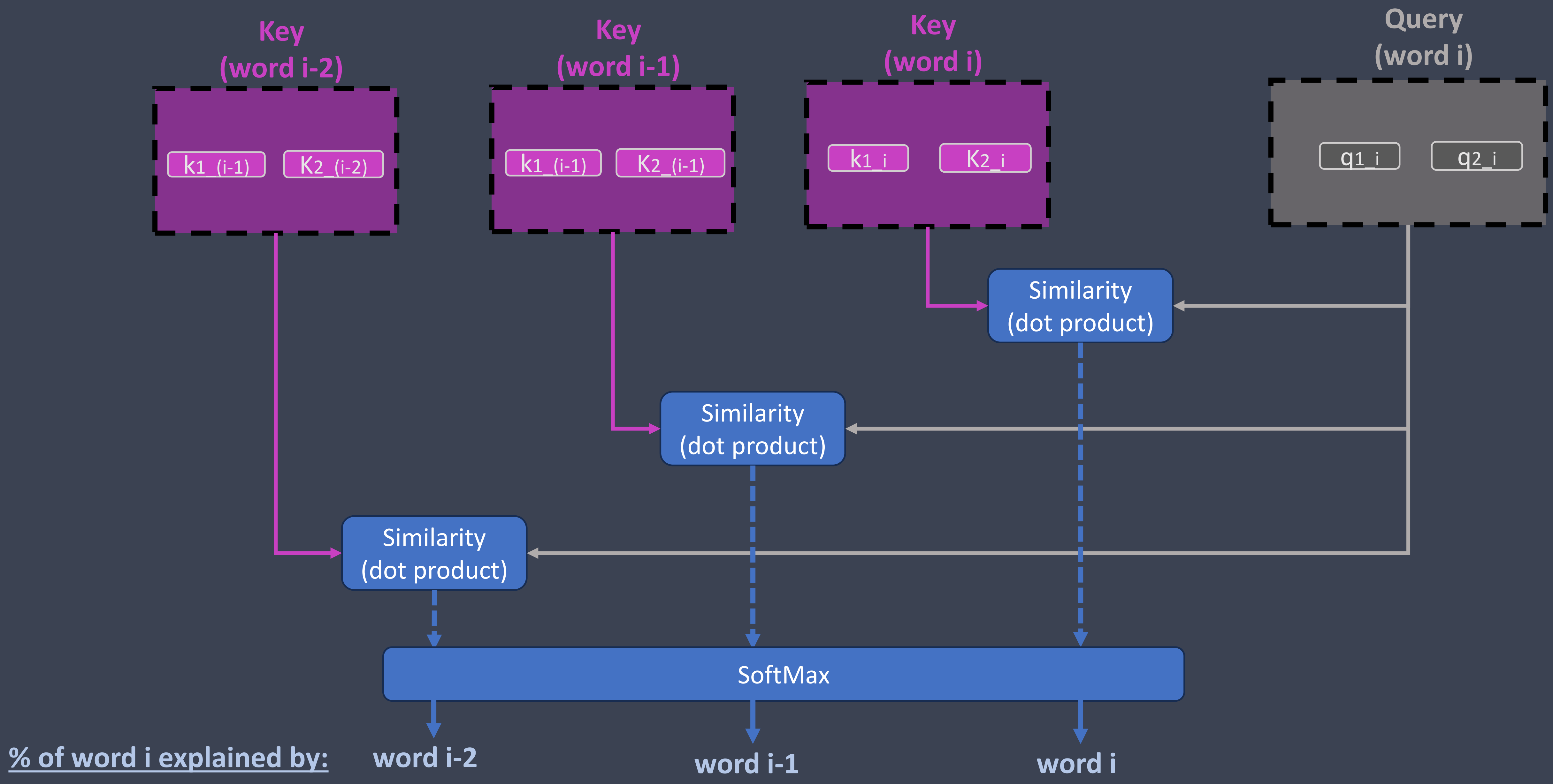
# Decoder-only transformer

Key-step 2: Masked self-attention



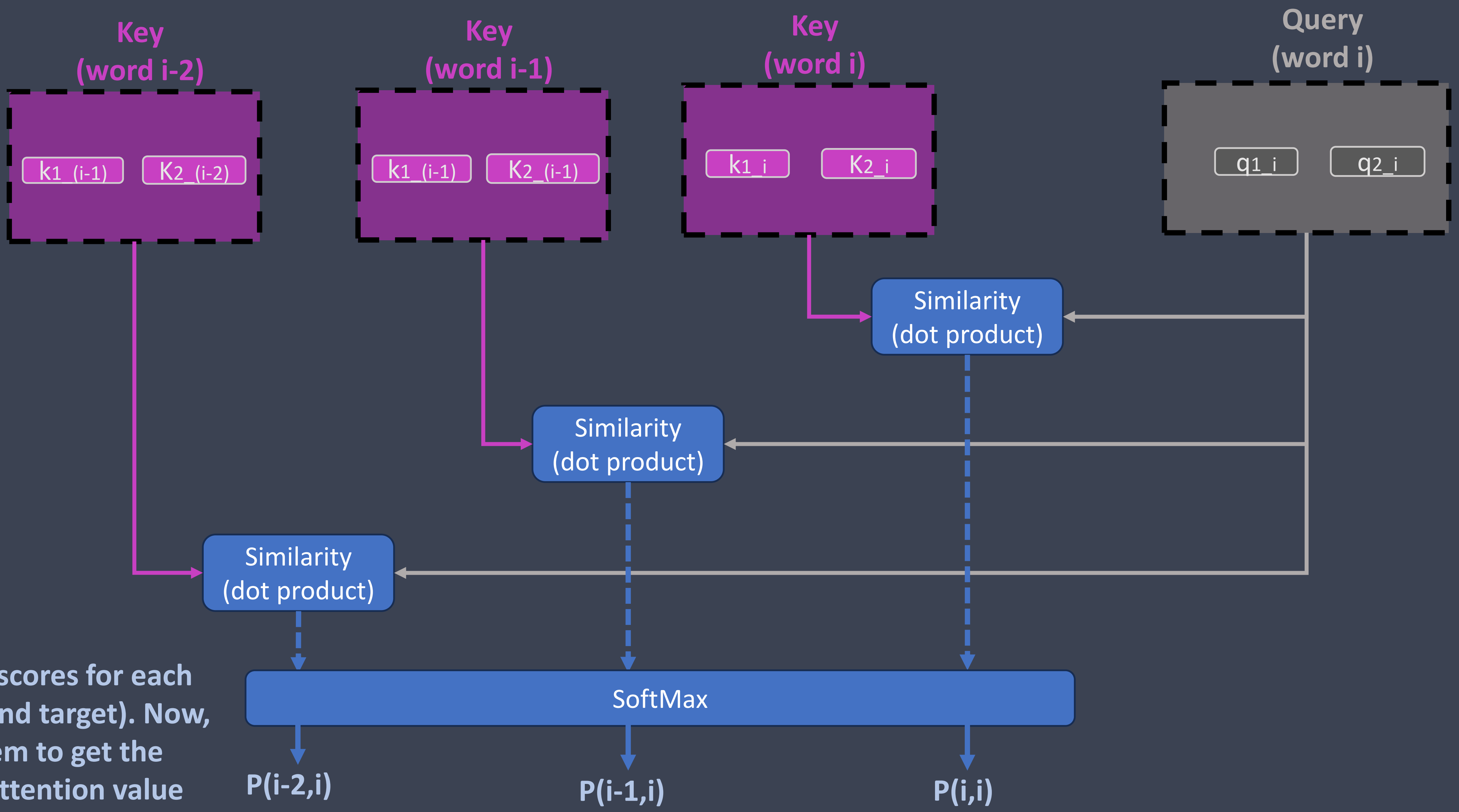
# Decoder-only transformer

Key-step 2: Masked self-attention



# Decoder-only transformer

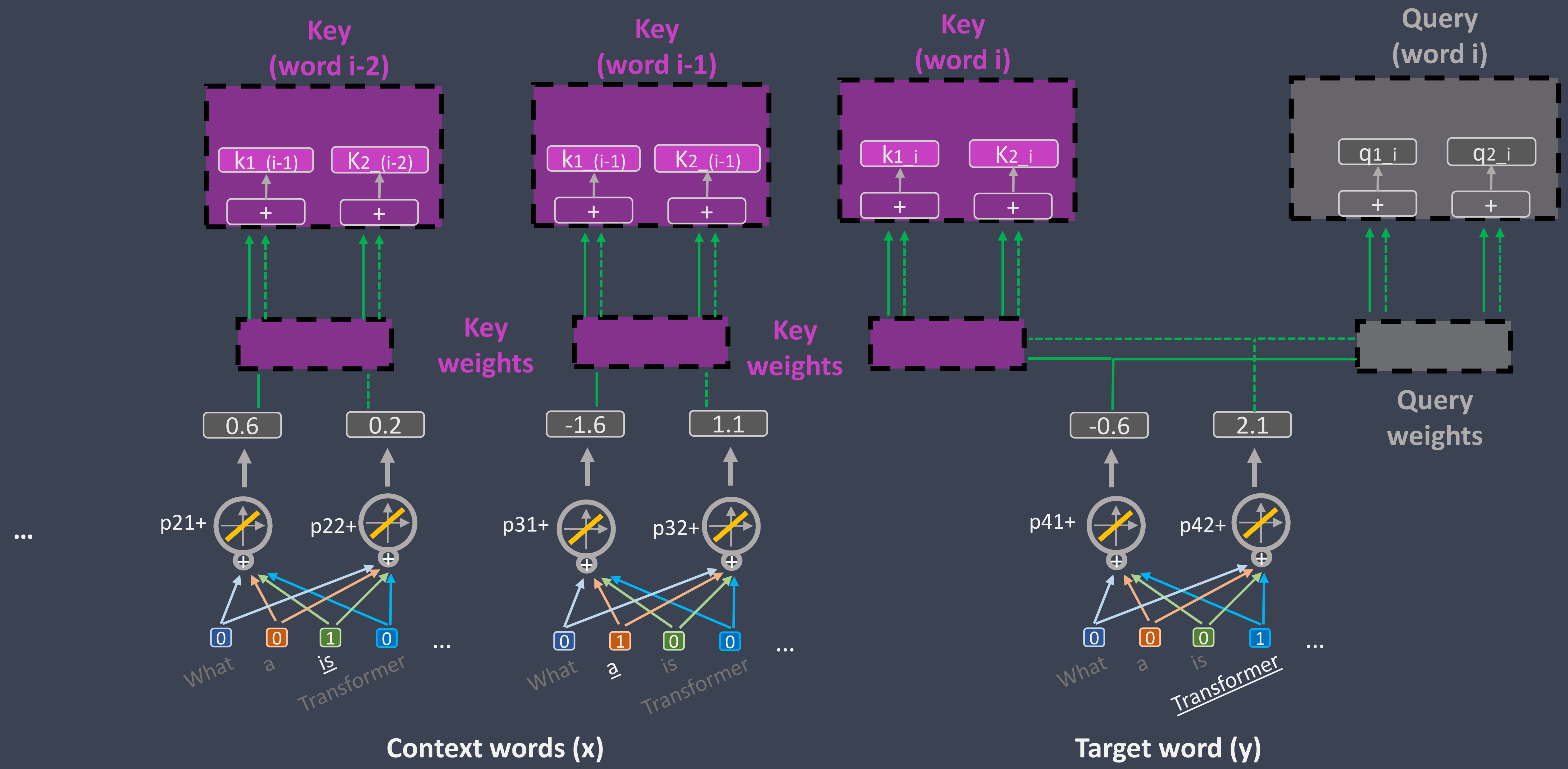
Key-step 2: Masked self-attention




**Self-attention scores for each word (context and target). Now, combine them to get the masked self-attention value**

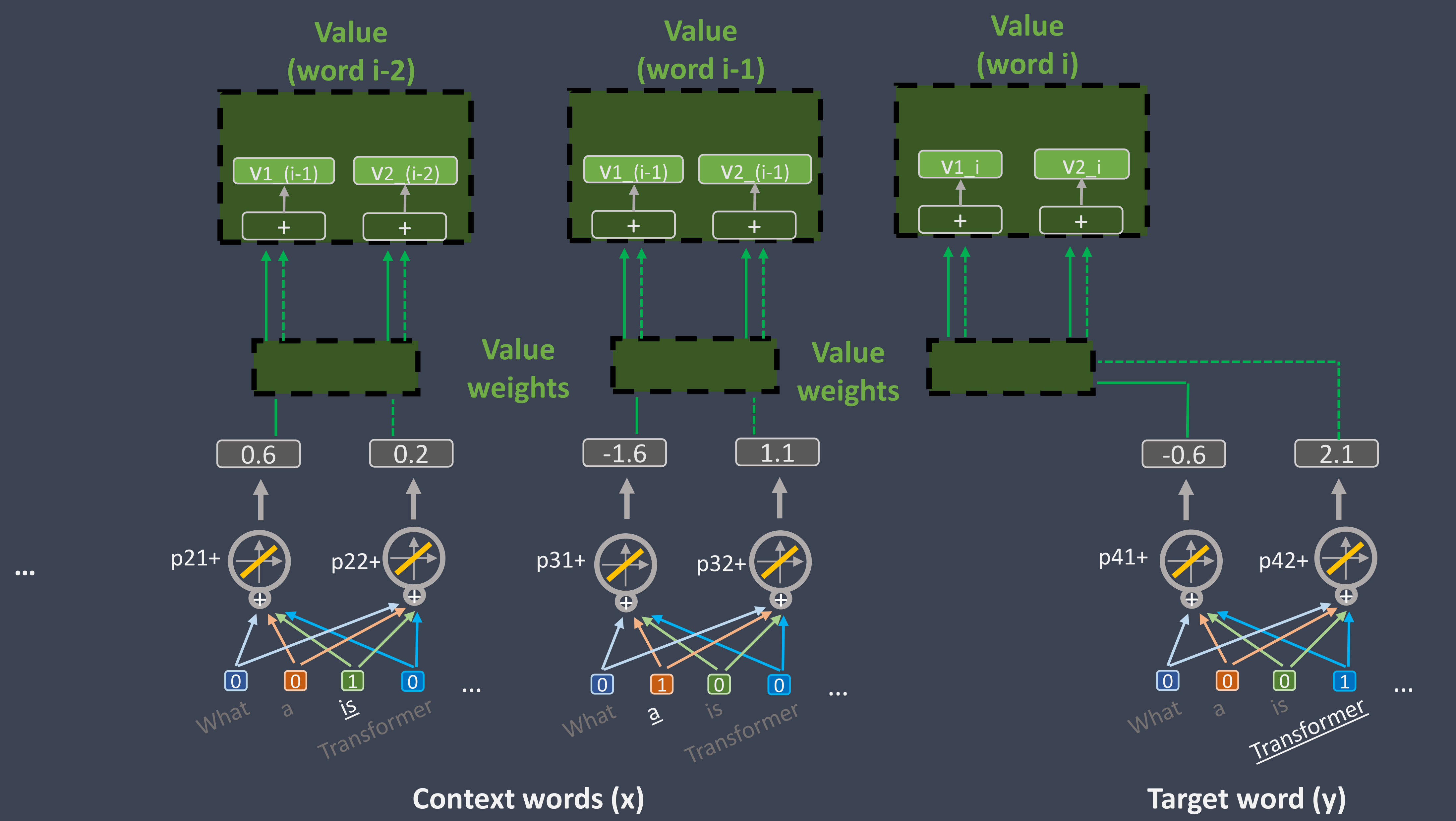
# Decoder-only transformer

Key-step 2: Masked self-attention



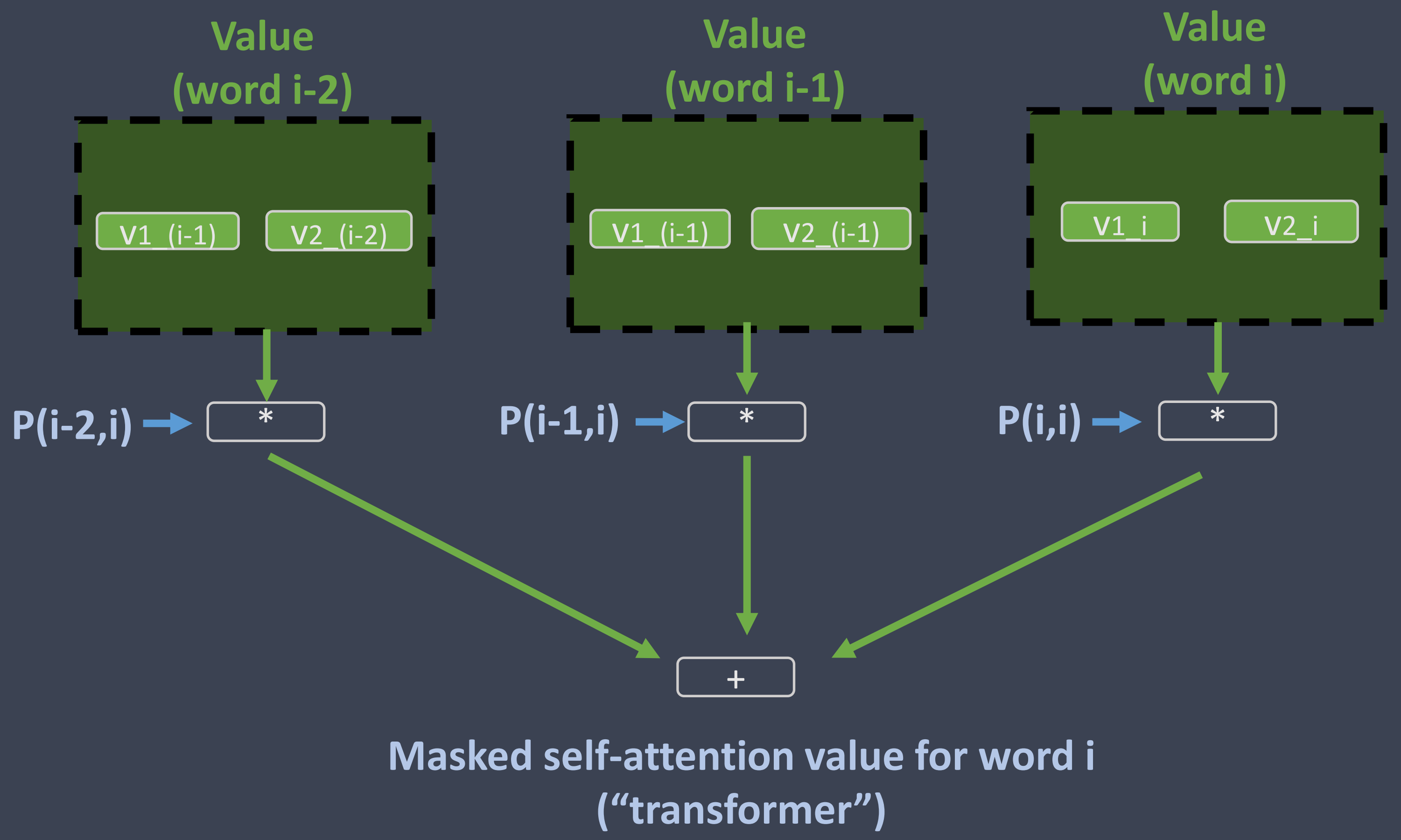
# Decoder-only transformer

Key-step 2: Masked self-attention



# Decoder-only transformer

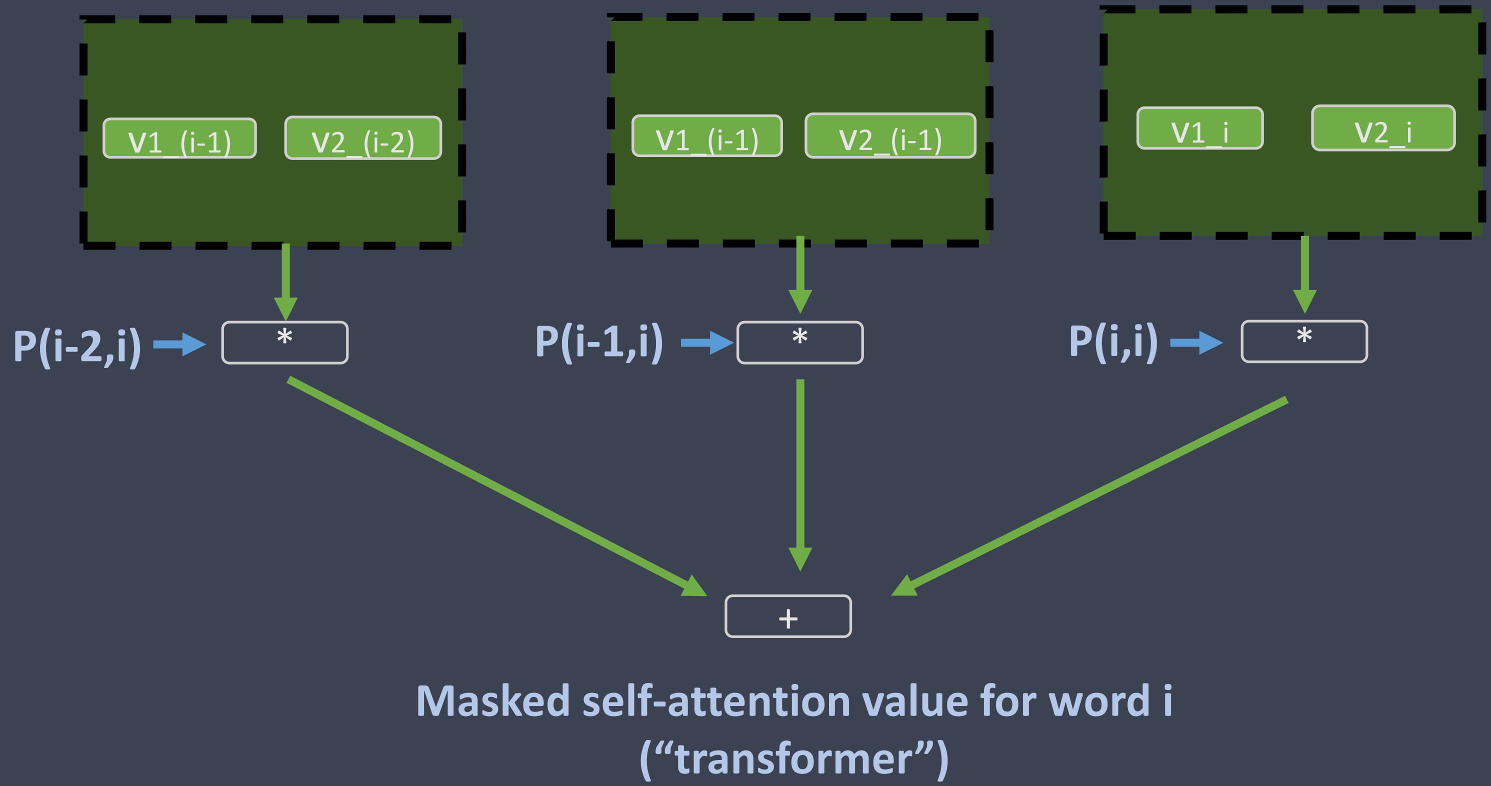
Key-step 2: Masked self-attention




Now.. repeat for each word!
Important: we reuse the same set of query weights for all words. In the same way, key and value weights are also re-used across words. This reduces the number of parameters + allows us to handle prompts that have different lengths because we can keep reusing the weights for as many words as we have in the prompt

# Decoder-only transformer

Key-step 2: Masked self-attention




Value weights (as opposed to using word+position embedding directly) gives additional flexibility to the architecture, allowing to reweight the initial embeddings and to reduce dimensionality.

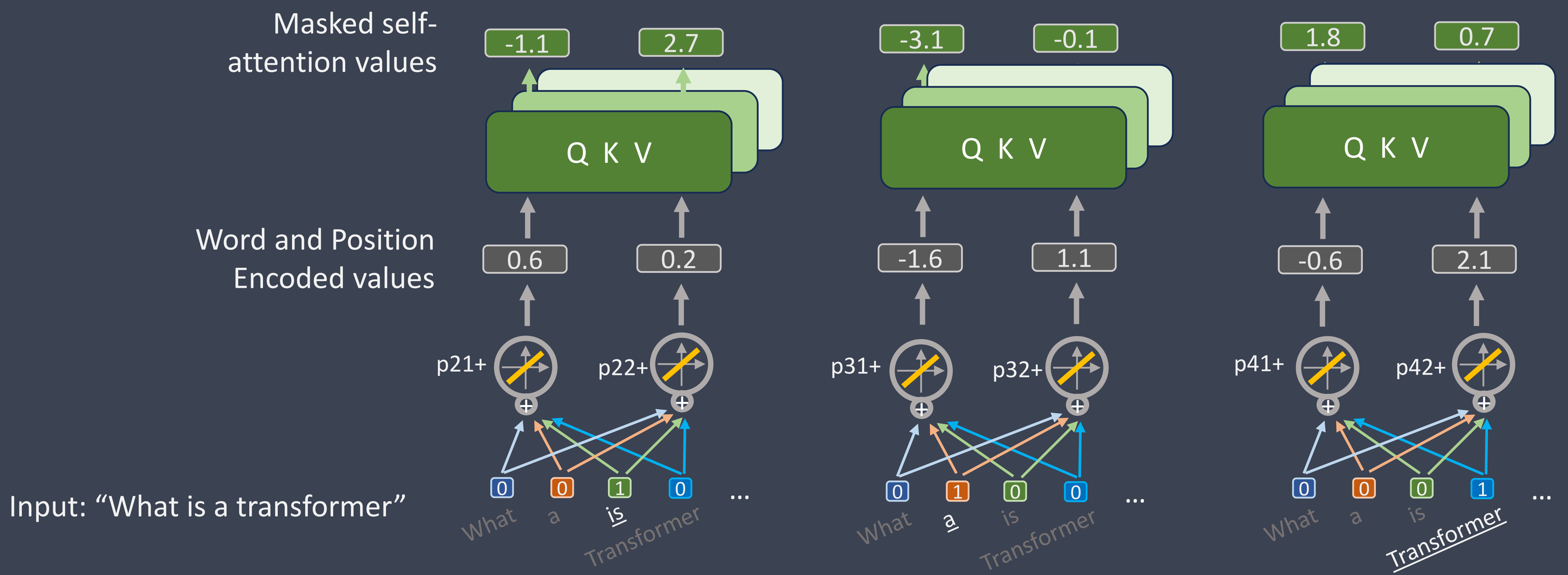
Now.. repeat for each word!

Important: we reuse the same set of query weights for all words. In the same way, key and value weights are also re-used across words. This reduces the number of parameters + allows us to handle prompts that have different lengths because we can keep reusing the weights for as many words as we have in the prompt

# Decoder-only transformer

Key-step 2: Masked self-attention

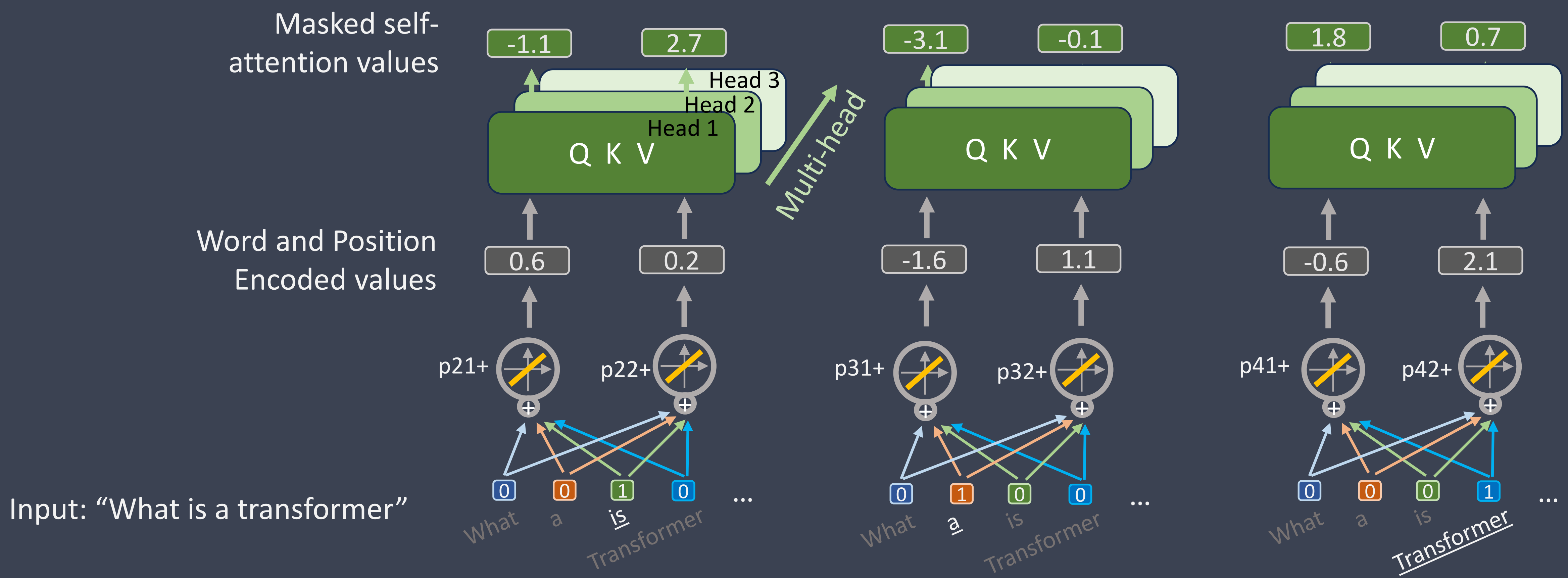
- We can have many sets of Query-Keys-Value to capture different types of relationships (e.g., different time-scales) (12 sets in the original GPT)



# Decoder-only transformer

Key-step 2: Masked self-attention

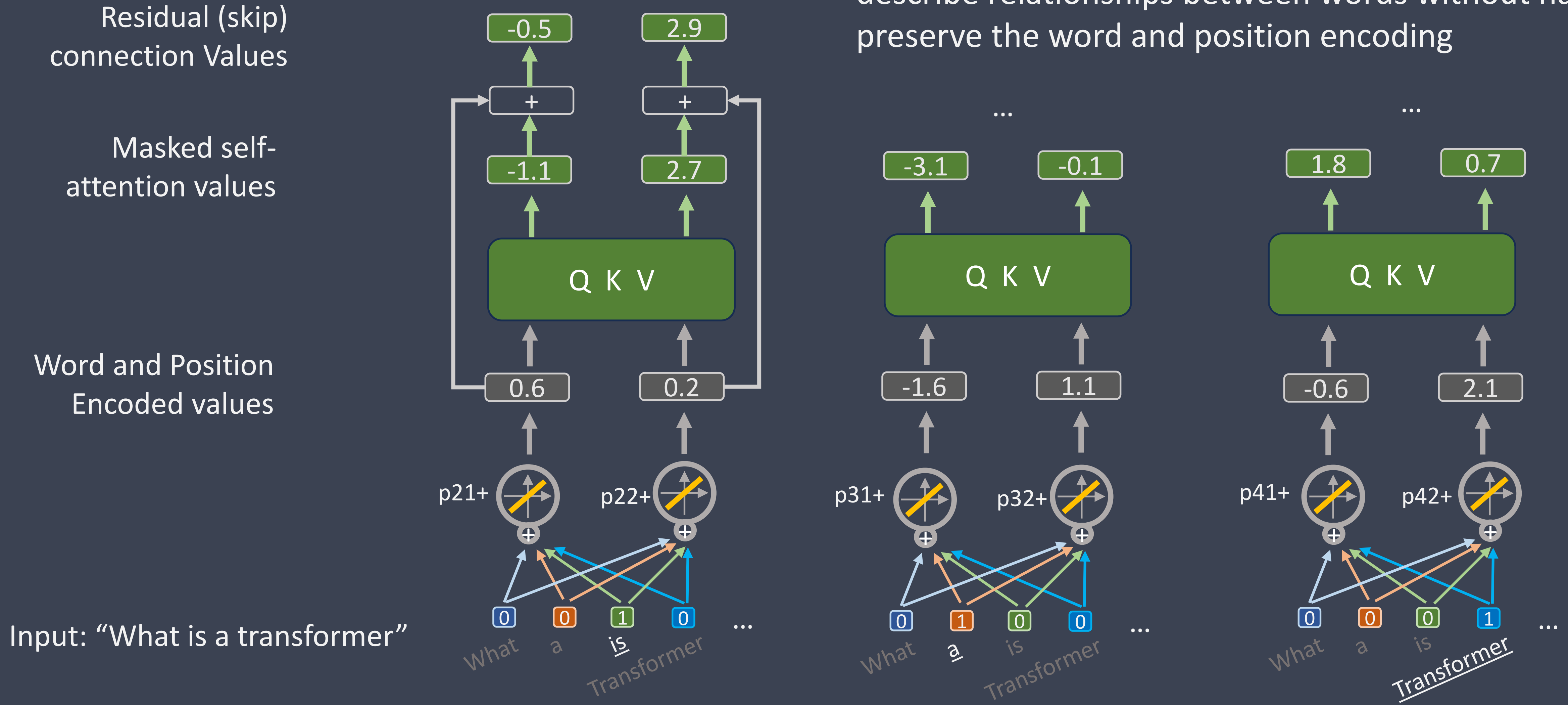
- We can have many sets of Query-Keys-Value to capture different types of relationships (e.g., different time-scales) (12 sets in the original GPT)



# Decoder-only transformer

Key-step 3: Add residual connection (skip connection)

Key advantage: The masked self-attention values can describe relationships between words without having to preserve the word and position encoding



## Key-step 4: Next-word generation

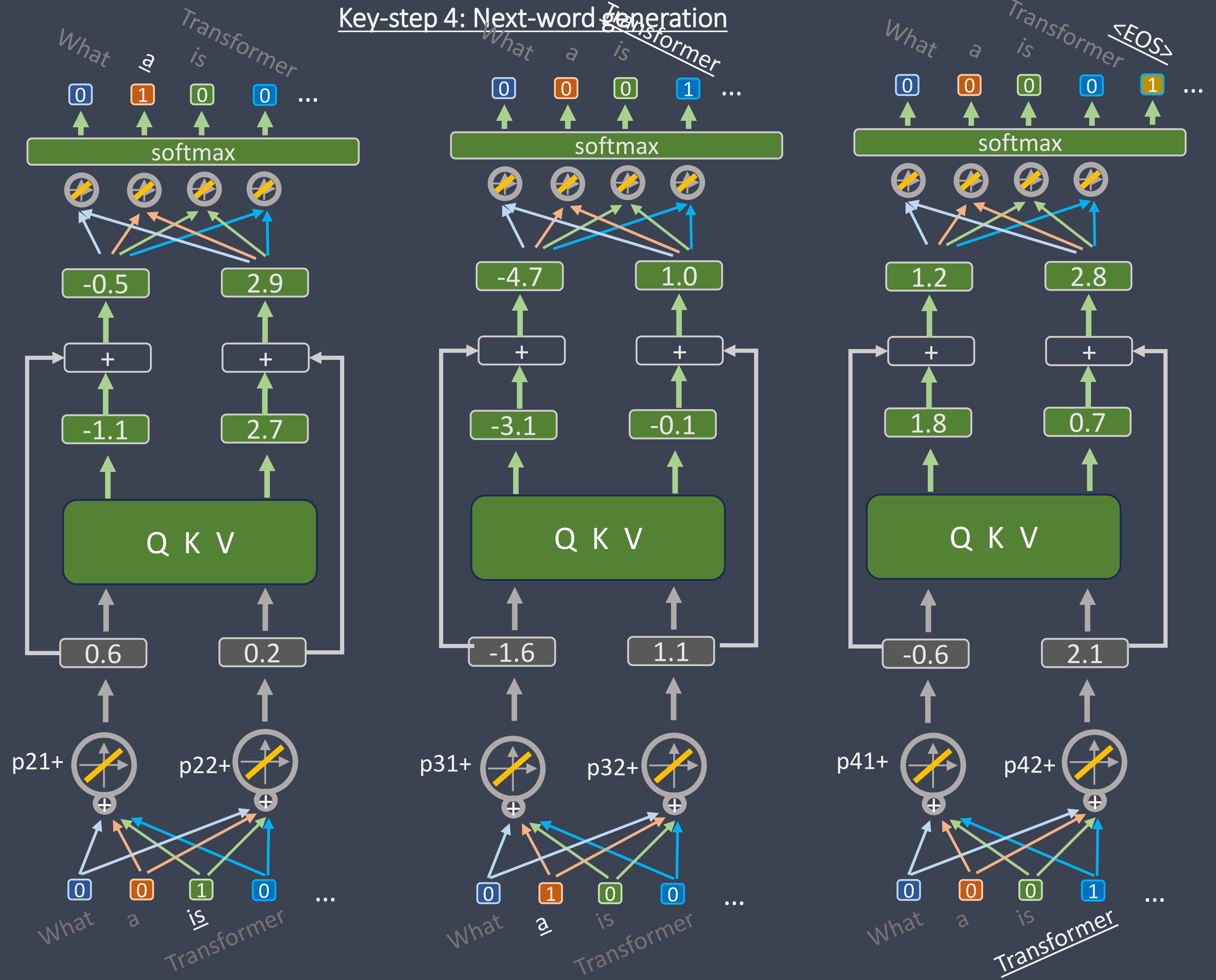
Fully-connected layer

Residual Connection Values

Masked self-attention values

Word and Position Encoded values

Input: “What is a transformer”



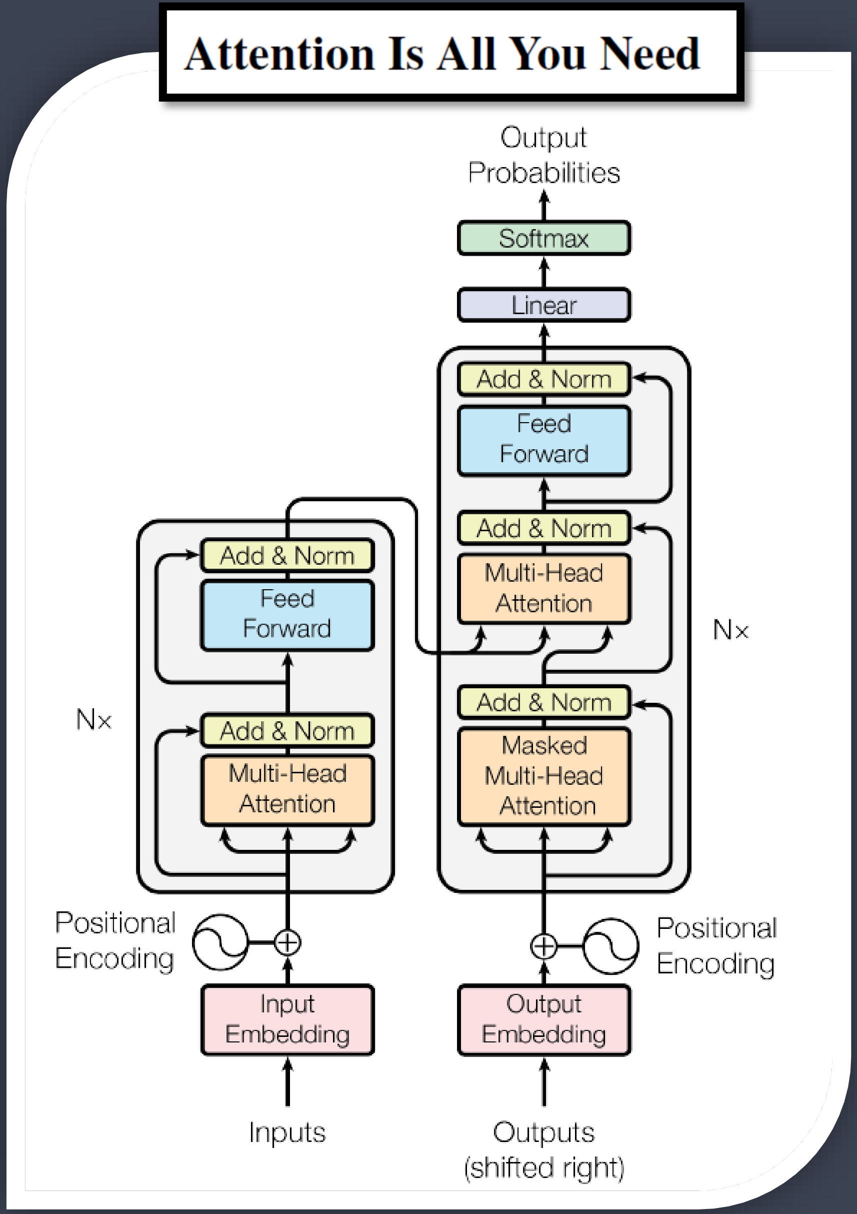
# So, attention! Let’s recap

- For each target word, that target word will relate to previous words to a different degree
- If all previous words were equally relevant, then we would have this situation:
- Block_size = 4
- Tokens corresponding to words
- Target word is “$w_0$”
- Attention scores for future words are set to zero, as we don’t want them to be seen when generating the target word. That’s why we call this “masked” self-attention
- Attention scores for $w_{-4}$, $w_{-3}$, $w_{-2}$, $w_{-1}$, $w_0$ : [0.20, 0.20, 0.20, 0.20, 0.20] (these are the P from the previous slide)
- That means that the four words will be weighted equally when predicting/generating $w_0$
- Indeed, attention should vary across different words. So, we figure this out with the Query-Key weights and their output
    - Query ~ based on the current word, what am I looking for?
    - Key ~ the potential matches to the query. One for each token in the prompt
    - Attention: Dot product (i.e., distance) between Q and K
    - Values (V) representing the content gets weighted by the attention score to pass the information forward
    - A set of Q, K, and V are called a head.
    - The word embedding + position embedding tells us “what we have”, a representation for a word+position. But, for a given dimension of our head, we might want to get different things from a given word+position. So, the values give us thig flexibility (many different aspects of interest in a word+position; values allow us to tease them apart, use them independently, and the reweight them at the end).

# So, attention! Let's recap

- ConvNet: specific layout that includes space
- Here, the notion of space (i.e., position) must be explicitly added to the input (position encoding)
- Parallel processing greatly facilitated by the following observations:
  - Each token is processed independently within a give layer i.e., without needing to wait for the results of previous tokens (for example, RNN process data sequentially, as the hidden state is passed forward through time)
  - Attention only within batch i.e., The elements across the batch dimension (independent examples) do not interact. This enables parallel processing.

# Attention? What kind of attention?




- Why is it called "self-attention"?
- Q, K, and V refer to the same source. In the original transformer paper we had different kinds of attention

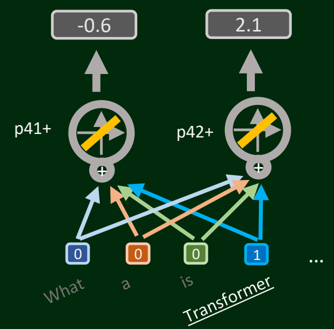
- The original transformer paper (i.e., for language translation) used self-attention in the encoder (looking at the whole block), masked self-attention in the decoder (looking only at previous tokens), and encoder-decoder attention (Q from decoder, K and V from encoder).
- We won't go into details, but this is just to show you that attention is a general approach that can take on many forms.

# Decoder-only transformer

Key-steps 2-4

- This is the code we saw before. Let's unmask the remaining lines

```
logits, loss = model(xb, yb)
# This line of code (from the previous slide) calls the following function

class GPTLanguageModel(nn.Module):
   def forward(self, idx, targets=None):
        B, T = idx.shape # idx and targets (xb and yb) are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        B, T, C = logits.shape
        logits = logits.view(B*T, C)
        targets = targets.view(B*T)
        loss = F.cross_entropy(logits, targets)

        return logits, loss
```



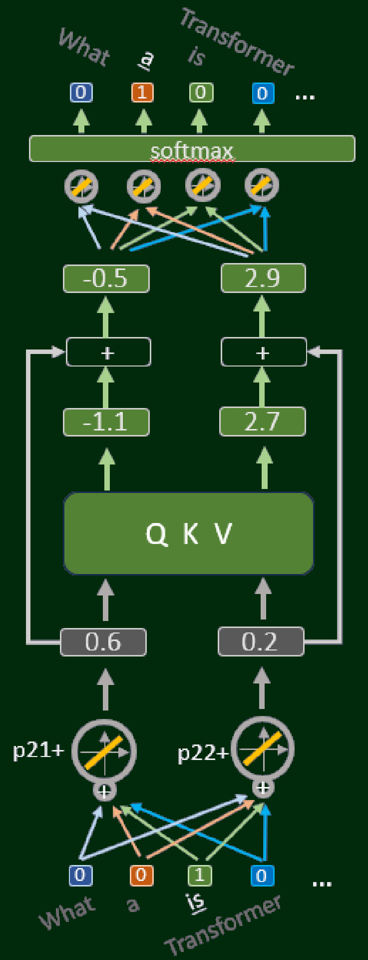
# Decoder-only transformer

Key-steps 2-4

- This is the code we saw before. Let’s unmask at the remaining lines

```
logits, loss = model(xb, yb)
# This line of code (from the previous slide) calls the following function

class GPTLanguageModel(nn.Module):
    def forward(self, idx, targets=None):
        B, T = idx.shape # idx and targets (xb and yb) are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        B, T, C = logits.shape
        logits = logits.view(B*T, C)
        targets = targets.view(B*T)
        loss = F.cross_entropy(logits, targets)

        return logits, loss

    def __init__(self):
        [...]
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)

        [...]
```

```
nn.Sequential(
    Block(n_embd, n_head=n_head),
    Block(n_embd, n_head=n_head),
    …
)
```

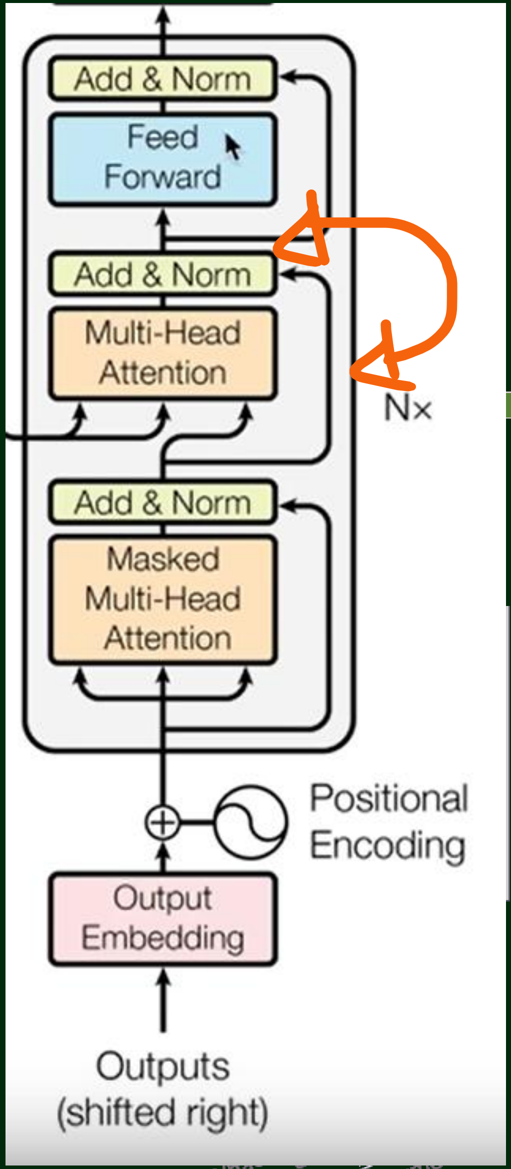


# Decoder-only transformer

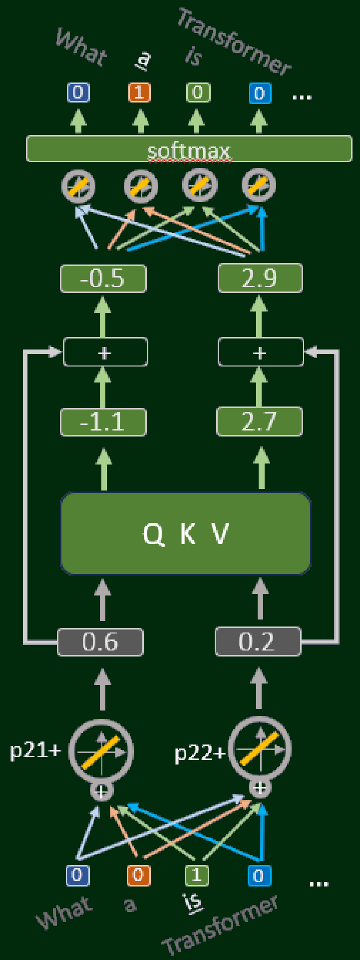
Key-steps 2-4

- Let's look inside the block

```
class Block(nn.Module):
    """ Transformer block: communication followed by computation """
    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size) # masked self-attention - see next slide
                                            # this is where we get tokens to communicate with each other
        self.ffwd = FeedFoward(n_embd)  # computation (on all tokens independently – non-linearity here)
        self.ln1 = nn.LayerNorm(n_embd) # different from original transformer paper, where norm was
        self.ln2 = nn.LayerNorm(n_embd) #   after attention and feedforward. This way is more common now

    def forward(self, x):
        x = x + self.sa(self.ln1(x))    # sa (self-attention): communication + skip connection
        x = x + self.ffwd(self.ln2(x))  # ffwd: computation + skip connection
        return x

class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd), # trick from original transformer paper
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        return self.net(x)
```



```python
class Head(nn.Module):  """ one head of self-attention """
    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)   # set up linear combination
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size))) # we can call self.tril (lower triang)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        # input of size (batch, time-step, channels)
        # output of size (batch, time-step, head size)
        B,T,C = x.shape
        k = self.key(x)   # (B,T,hs)   Calculate K and Q
        q = self.query(x) # (B,T,hs)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, hs) @ (B, hs, T) -> (B, T, T)
                                                         # Q * K, rescaled
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T). Masking future tokens
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,hs)
        out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        return out

class MultiHeadAttention(nn.Module):   """ multiple heads of self-attention in parallel """
    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(head_size * num_heads, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1) # concat all outputs of the multiple heads
        out = self.dropout(self.proj(out)) # apply the projection (linear transf of output)
        return out
```



# Decoder-only transformer

Training data: The old man and the sea

```
# hyperparameters
batch_size = 8 # how many independent sequences will we process in parallel?
block_size = 64 # what is the maximum context length for predictions?
max_iters = 5000
eval_interval = 100
learning_rate = 3e-4
device = 'cpu'
eval_iters = 200
n_embd = 150
n_head = 6
n_layer = 6
dropout = 0.2
```

```
vocab_size: 66
1.658766 M parameters
step 0: train loss 4.1353, val loss 4.1389
step 100: train loss 2.4421, val loss 2.4553
```

```
step 4000: train loss 1.3488, val loss 1.5371
step 4100: train loss 1.3510, val loss 1.5428
step 4200: train loss 1.3433, val loss 1.5435
step 4300: train loss 1.3258, val loss 1.5116
step 4400: train loss 1.3345, val loss 1.5276
step 4500: train loss 1.3206, val loss 1.5285
step 4600: train loss 1.3060, val loss 1.5210
step 4700: train loss 1.3007, val loss 1.5020
step 4800: train loss 1.2934, val loss 1.5107
step 4900: train loss 1.2925, val loss 1.5073
step 4999: train loss 1.2794, val loss 1.4861
```

```
train loss 1.2042, val loss 1.4441
train loss 1.1984, val loss 1.4640
```

Overfitting (train loss a good bit lower than val loss) -> reasonable, as our dataset is very small

Overfitting gets worse if you keep going

```python
# With a prompt
context = torch.stack([torch.tensor(encode("He was an old "),
dtype=torch.long)])print(decode(m.generate(context, max_new_tokens=500)[0].tolist()))
```

Train loss = 4.1
Val loss = 2.7

```
He was an old stkiseoande is osrojodllzano pitt f tWriCateUr-he t o ShonecatQ nm anSen tabouGd and ewifiner
Me toCirhs Ine loan
neyell gh fto? inkud m omt
thitheibusf.]0d ghert1hell, ? thtwh a0kie d['cothgFhhihd.GBfAowo1or.
la ndha y af oyld tsflmat
```

Train loss = 1.3
Val loss = 1.5

```
He was an old are a nesternaps with the shark's dorrous and bream, he thought.
He thought. Myst come worken, the old man said. In then first when I will
hit have up to see it good
about?
He fellbing his bill, he thought. And knew he course I could not like as get
the watchanging.
```

# A lot of progress in only a decade

- Word embeddings (2013): word2vec https://arxiv.org/abs/1301.3781
- Contextualised word embeddings (2018): ELMo https://arxiv.org/abs/1802.05365
  (based on bidirectional LSTM)
- Attention is all you need (2018)! The transformer https://arxiv.org/abs/1706.03762
- GPT: 117M parameters
- GPT-2 (2018). https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf
  Pretraining data ~40GB. 1.5B parameters
- GPT-3 (2020). https://arxiv.org/pdf/2005.14165.pdf
  175B parameters
- InstructGPT (2021). Humans in the loop to demonstrate the desired model behaviour (e.g., telling the truth!)
- ChatGPT (2022). Similar approach to instructGPT, but with a dialogue dataset from human AI trainers
- GPT-4 (2023). Unknown architecture.
  Possibly ~1 trillion parameters (rumours)

The number of parameters is impressive. Even more impressive is that with this architecture and lots of data we can actually train those massive neural networks!

# Fine-tuning

- Most people use pre-trained models.

- Many **fine-tune** pre-trained models

- Some train their own models

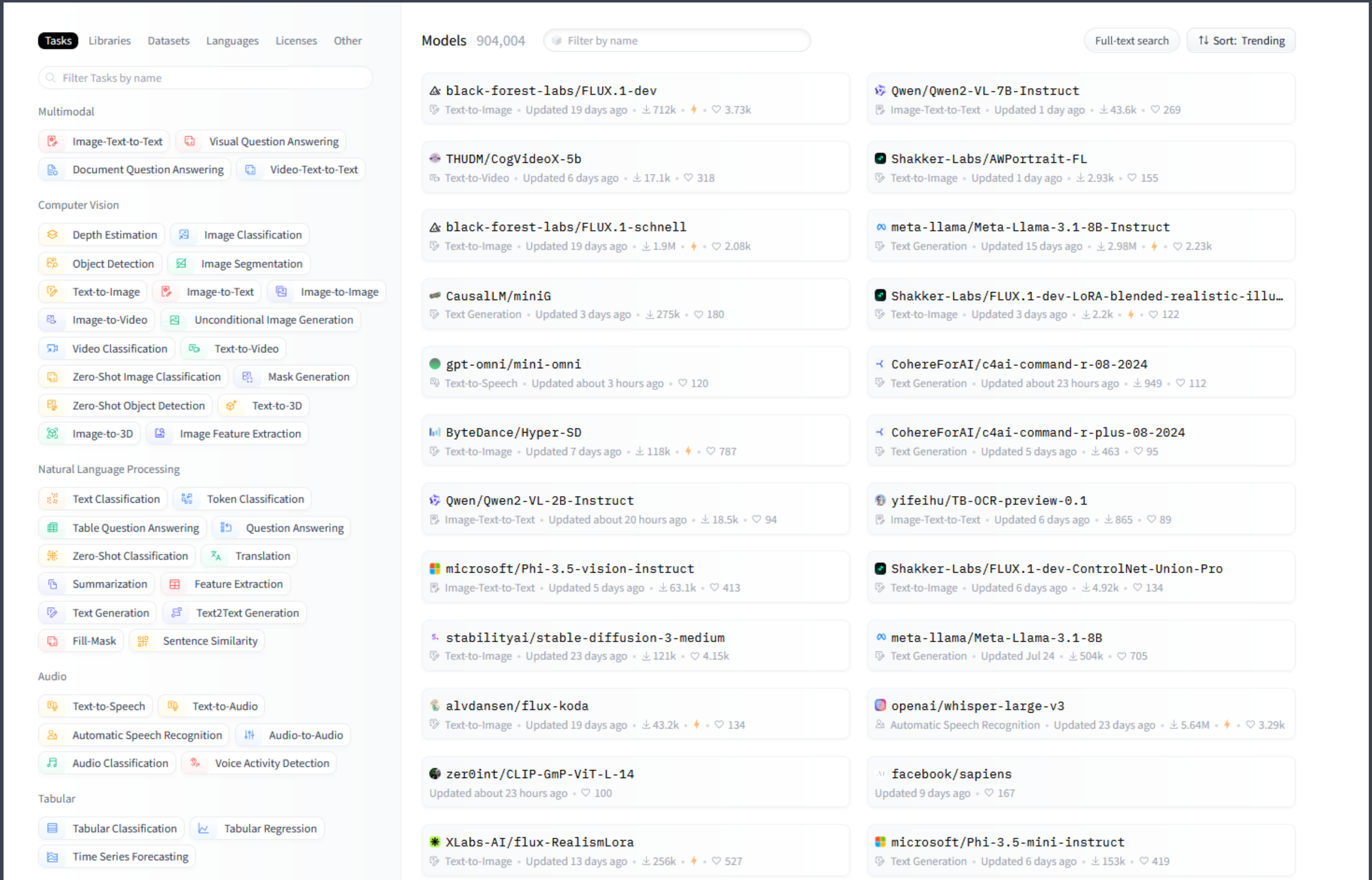
- What does it mean to "fine-tune" a model?
- Technique for customizing and optimising model performance for specific use cases
- Many ways to do that. E.g., Mistral AI provides a fine-tuning API
  https://docs.mistral.ai/guides/finetuning/
  With open source fine-tuning code here: https://github.com/mistralai/mistral-finetune/
- See the following guides
  https://huggingface.co/docs/transformers/en/training
  https://medium.com/@lokaregns/fine-tuning-transformers-with-custom-dataset-classification-task-f261579ae068

# Fine-tuning

- There exist very different approaches
- The key part is to gather a dataset. This dataset might be organised into
  - system prompts: instructions that shape the model response (e.g., “be polite”)
  - user prompts: user input
- We then use this dataset to re-train the model weights
  - Option 1: Replace final layer altogether (if task is different from original one)
  - Option 2: Update model parameters (some parameters can be frozen). This is when the task is similar to the original one
  - Check out existing models and fine-tuned models https://huggingface.co/models

  - Data is crucial. Different types/steps: Narrower dataset. Or supervised dataset (input and expected output e.g., chatbot). Learning from human feedback

# Fine-tuning


Tasks
Libraries
Datasets
Languages
Licenses
Other
Filter Tasks by name
Multimodal
Image-Text-to-Text
Visual Question Answering
Document Question Answering
Video-Text-to-Text
Computer Vision
Depth Estimation
Image Classification
Object Detection
Image Segmentation
Text-to-Image
Image-to-Text
Image-to-Image
Image-to-Video
Unconditional Image Generation
Video Classification
Text-to-Video
Zero-Shot Image Classification
Mask Generation
Zero-Shot Object Detection
Text-to-3D
Image-to-3D
Image Feature Extraction
Natural Language Processing
Text Classification
Token Classification
Table Question Answering
Question Answering
Zero-Shot Classification
Translation
Summarization
Feature Extraction
Text Generation
Text2Text Generation
Fill-Mask
Sentence Similarity
Audio
Text-to-Speech
Text-to-Audio
Automatic Speech Recognition
Audio-to-Audio
Audio Classification
Voice Activity Detection
Tabular
Tabular Classification
Tabular Regression
Time Series Forecasting
Models 904,004
Filter by name
Full-text search
Sort: Trending
black-forest-labs/FLUX.1-dev
Text-to-Image • Updated 19 days ago • 712k • 3.73k
Qwen/Qwen2-VL-7B-Instruct
Image-Text-to-Text • Updated 1 day ago • 43.6k • 269
THUDM/CogVideoX-5b
Text-to-Video • Updated 6 days ago • 17.1k • 318
Shakker-Labs/AWPortrait-FL
Text-to-Image • Updated 1 day ago • 2.93k • 155
black-forest-labs/FLUX.1-schnell
Text-to-Image • Updated 19 days ago • 1.9M • 2.08k
meta-llama/Meta-Llama-3.1-8B-Instruct
Text Generation • Updated 15 days ago • 2.98M • 2.23k
CausalLM/miniG
Text Generation • Updated 3 days ago • 275k • 180
Shakker-Labs/FLUX.1-dev-LoRA-blended-realistic-illu…
Text-to-Image • Updated 3 days ago • 2.2k • 122
gpt-omni/mini-omni
Text-to-Speech • Updated about 3 hours ago • 120
CohereForAI/c4ai-command-r-08-2024
Text Generation • Updated about 23 hours ago • 949 • 112
ByteDance/Hyper-SD
Text-to-Image • Updated 7 days ago • 118k • 787
CohereForAI/c4ai-command-r-plus-08-2024
Text Generation • Updated 5 days ago • 463 • 95
Qwen/Qwen2-VL-2B-Instruct
Image-Text-to-Text • Updated about 20 hours ago • 18.5k • 94
yifeihu/TB-OCR-preview-0.1
Image-Text-to-Text • Updated 6 days ago • 865 • 89
microsoft/Phi-3.5-vision-instruct
Image-Text-to-Text • Updated 5 days ago • 63.1k • 413
Shakker-Labs/FLUX.1-dev-ControlNet-Union-Pro
Text-to-Image • Updated 6 days ago • 4.92k • 134
stabilityai/stable-diffusion-3-medium
Text-to-Image • Updated 23 days ago • 121k • 4.15k
meta-llama/Meta-Llama-3.1-8B
Text Generation • Updated Jul 24 • 504k • 705
alvdansen/flux-koda
Text-to-Image • Updated 19 days ago • 43.2k • 134
openai/whisper-large-v3
Automatic Speech Recognition • Updated 23 days ago • 5.64M • 3.29k
zer0int/CLIP-GmP-ViT-L-14
Updated about 23 hours ago • 100
facebook/sapiens
Updated 9 days ago • 167
XLabs-AI/flux-RealismLora
Text-to-Image • Updated 13 days ago • 256k • 527
microsoft/Phi-3.5-mini-instruct
Text Generation • Updated 6 days ago • 153k • 419

# Fine-tuning

MISTRAL AI_

## Prepare the dataset

Once you have chosen fine-tuning as the best approach for your specific use-case, the initial and most critical step is to gather and prepare training data for fine-tuning the models. Here are six specific use cases that you might find helpful:

### Use cases

- **Use case 1: specific tone**
- **Use case 2: specific format**
- **Use case 3: specific style**
- **Use case 4: coding**
- **Use case 5: domain-specific augmentation in RAG**
- **Use case 6: knowledge transfer**
- **Use case 7: agents for function calling**

# Fine-tuning

MISTRAL AI_

Setting up a fine-tuning job with Mistral API

python typescript curl

```python
# create a fine-tuning job
created_jobs = client.fine_tuning.jobs.create(
    model="open-mistral-7b",
    training_files=[{"file_id": ultrachat_chunk_train.id, "weight": 1}],
    validation_files=[ultrachat_chunk_eval.id],
    hyperparameters={
        "training_steps": 10,
        "learning_rate":0.0001
    },
    auto_start=False
)

# start a fine-tuning job
client.fine_tuning.jobs.start(job_id = created_jobs.id)

created_jobs
```

# Challenges and limitations

Or we can do this “manually”, which can look more familiar to us, now that we have looked at the code for GPT. Below is a snippet of code that makes the final layer trainable while freezing the other weights

```python
from transformers import AutoModelForCausalLM, Trainer, TrainingArguments

# Load the model
model = AutoModelForCausalLM.from_pretrained("mistralai/Mistral-7B-v0.1")

# Freeze some layers
for param in model.base_model.parameters():
    param.requires_grad = False

# Fine-tune only the top layers
for param in model.lm_head.parameters():
    param.requires_grad = True
```

```python
# Define training arguments
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=4,
    per_device_eval_batch_size=4,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir="./logs",
)

# Create Trainer instance
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
)

# Start training
trainer.train()
```