O'REILLY®

Hands-On Machine Learning with Scikit-Learn and PyTorch

Concepts, Tools, and Techniques to Build Intelligent Systems

Aurélien Géron

Online Bonus Content for Hands-On Machine Learning with Scikit-Learn and PyTorch

Aurélien Geron

O'REILLY®

Hands-On Machine Learning with Scikit-Learn and PyTorch
by Aurélien Geron

Published by O'Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*https://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nicole Butterfield
Develeopment Editor: Michele Cronin
Production Editor: Beth Kelly
Copyeditor: Sonia Saruba
Proofreader: Kim Cofer
Indexer: Potomac Indexing LLC
Cover Designer: Susan Brown
Cover Illustrator: José Marzan Jr.
Interior Designer: David Futato
Interior Illustrator: Kate Dullea

October 2025: First Edition

Revision History for the First Edition
2025-10-22: First Release

See *https://oreilly.com/catalog/errata.csp?isbn=9798341607989* for release details.

979-8-341-60798-9

[LSI]

APPENDIX D
Relative Positional Encoding

Transformers struggle with long sequences. As we saw in Chapter 15, they suffer from the quadratic attention problem: the computational and memory cost of processing a sequence increases quadratically with the sequence length, so transformers are painfully slow and memory hungry for long input sequences. Luckily, this problem can be alleviated using the techniques discussed in Chapters 16 and 17: in fact, many modern transformer architectures can handle extremely long sequences of thousands, or even hundreds of thousands of tokens (or more).

However, speed and memory usage aren't the only problems: transformers are not very accurate for long input sequences either. This might be a bit surprising: you may expect a transformer to be *more* accurate when given extra information; but that's not what happens, at least for very long sequences. There are a couple reasons for this:

- Firstly, the training set naturally contains more tokens at small positions than at large positions, since all sequences contain the former, but only long sequences contain the latter. Therefore, when using positional encodings based on the absolute position of each token, the model just doesn't get as much training on tokens located at large positions. Moreover, at inference time, it cannot easily extrapolate to longer sequences than the longest one in the training set.
- Secondly, it's easier to find relevant information in a short document than in a long one. Admittedly, this is true for humans as well, but we humans have a very useful *locality bias*: we assume that nearby words are generally more relevant than distant words; in practice, this assumption is usually correct. However, the transformer has no such inductive bias (i.e., a bias baked into its architecture); it has to learn this fact during training. If it doesn't learn perfectly, then it will be distracted by far away tokens when processing each sentence: the larger the sequence, the more unrelated tokens will distract the transformer.

The *positional encoding* (PE) techniques discussed in this appendix address both issues: they are *relative* PEs, meaning that they only care about the relative distances between tokens rather than their absolute positions. For example, the relative distance between the query token at position $i = 10$ and the key token at position $j = 6$ is $r = i - j = -4$. Since relative PE techniques only focus on the relative distance, they don't care whether a token is located at position 10 or 10,000, or whether the sequence is short or long. This addresses the first issue: the model cares just as much about all tokens, and it's also easier to extrapolate to longer sequences than those seen during training.

Moreover, using relative PEs introduces the helpful locality bias, since the model will see much more nearby pairs than distant pairs during training. Indeed, in a sequence of length 100, there are 198 pairs of tokens with relative distance +1 or –1 (i.e., the pairs whose query token is just next to the key token), but only two pairs of tokens with relative distance +99 or –99. Moreover, only large sequences contain pairs of tokens that have a large relative distance.

In this appendix, we will discuss three of the main relative PE techniques used today:

- *Relative Position Encodings* (RPE)
- *Rotary Position Embeddings* (RoPE)
- *Attention with Linear Biases* (ALiBi)

All three methods get rid of the absolute PEs that the original Transformer architecture adds to the input embeddings before entering the first transformer block. Instead, RPE, RoPE, and ALiBi tweak the scaled dot-product attention equation to ensure that the output depends on the relative positions between token pairs. However, they all do so in different ways, as we will see. Let's start with RPE.

Relative Position Encodings (RPE)

RPE is the original relative PE method proposed in 2018 by Google researchers Peter Shaw et al.[1] In short, it adds trainable relative position embeddings to the key and value tokens when computing the scaled dot-product attention. Let's see exactly how.

The authors reasoned that the exact relative distance matters less when two tokens are very far away from each other, so they decided to clamp the relative distances to a predefined range from $-r_{\text{max}}$ to $+r_{\text{max}}$ (typically $r_{\text{max}} = 128$). If the relative distance between two tokens is higher than r_{max} or lower than $-r_{\text{max}}$, then their relative distance

1 Peter Shaw et al., "Self-Attention with Relative Position Representations", arXiv preprint arXiv:1803.02155 (2018).

r is clamped to r_{max} or $-r_{max}$, respectively. Figure D-1 shows the relative distances between each pair of query and key tokens, clamped using a tiny $r_{max} = 4$.

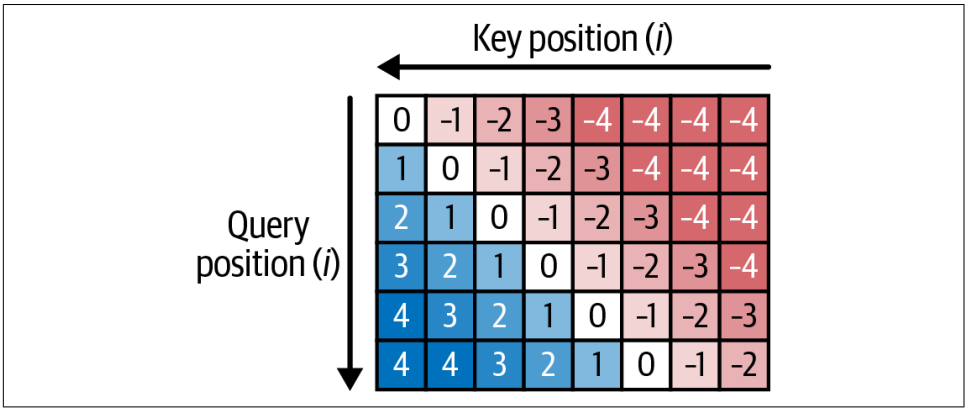

Figure D-1. Relative distance between each pair of query and key tokens, clamped to $\pm r_{max}$ (e.g., ±4)

With that in mind, RPE adds two trainable embedding matrices $\mathbf{A}^K$ and $\mathbf{A}^V$ to each MHA layer. In both matrices, each row is an embedding that represents a possible relative distance r ranging from $-r_{max}$ to $+r_{max}$, so there are $2r_{max} - 1$ embeddings in each matrix (e.g., the first row represents the relative distance $-r_{max}$, the second represents $-r_{max} + 1$, and so on). The dimensionality of each embedding in $\mathbf{A}^K$ is d_k, which is the key's dimensionality, and the dimensionality of each embedding in $\mathbf{A}^V$ is d_v, which is the value's dimensionality (in most transformer architectures, $d_k = d_v$). Now let's see how these embeddings are actually used.

Firstly, when computing the attention score $e_{i,j}$ for the i^{th} query token $\mathbf{q}_i$ and the j^{th} key token $\mathbf{k}_j$, the embedding for the relative position $r = i - j$ from matrix $\mathbf{A}^K$, denoted $\mathbf{a}^K_{i-j}$, is added to the key token $\mathbf{k}_j$. In other words, instead of computing the usual scaled dot product $e_{i,j} = \mathbf{q}_i \cdot \mathbf{k}_j / \sqrt{d_k}$, RPE computes $e_{i,j} = \mathbf{q}_i \cdot (\mathbf{k}_j + \mathbf{a}_{i-j}) / \sqrt{d_k}$. Once we have computed the attention score for all query/token pairs, we can compute the attention weights $\alpha_{i,j}$ normally using the softmax function.

Secondly, when computing the output $\mathbf{o}_i$ for the i^{th} query token, instead of calculating the usual weighted sum of value tokens, $\mathbf{o}_i = \sum_{j=1}^{L_k} \alpha_{i,j}\mathbf{v}_j$, we add a relative position embedding $\mathbf{a}^V_{i-j}$ to each value vector: $\mathbf{o}_i = \sum_{j=1}^{L_k} \alpha_{i,j}\left(\mathbf{v}_j + \mathbf{a}^V_{i-j}\right)$. Note that this embedding comes from the second matrix $\mathbf{A}^V$.

The RPE authors showed that this method significantly improves model accuracy, especially for long sequences. However, they remarked that the second embedding (the one from matrix $\mathbf{A}^V$ that is added to $\mathbf{v}_j$) didn't seem to help, so later relative PE techniques all dropped this idea.

In January 2019, other Google researchers released the Transformer-XL architecture,[2] which uses a variant of RPE that learns the content-to-content relationships (i.e., the usual $\mathbf{QK}^\intercal$ term), as well as content-to-position, position-to-content, and position-to-position relationships, all separately. This architecture got impressive results on many benchmarks, and it particularly shines on long sequences. However, it's more complex and requires more parameters, so researchers looked for simpler solutions.

In October 2019, yet another team of Google researchers released the T5 architecture (introduced in Chapter 15), which greatly simplified the RPE method: instead of introducing two embeddings in the attention equation, their method adds a single learnable bias (a scalar) to each attention score, depending on the relative position. This *bias-based RPE* method computes the attention scores like this: $e_{i,j} = \mathbf{q}_i \cdot \mathbf{k}_j / \sqrt{d_k} + b_{i-j}$, where b_{i-j} is the learnable bias for the relative position $r = i - j$. That's a nice and simple solution, it adds much fewer parameters, and it runs significantly faster than the original RPE method (often called *full-embedding RPE* or just *full RPE*), while even performing better.

Let's implement bias-based RPE in PyTorch. For example, we can tweak the `MultiheadAttention` class we defined in Chapter 15. The first step is to add a parameter for the biases in the constructor:

```
class MultiheadAttentionWithBiasRPE(nn.Module):
    def __init__(self, embed_dim, num_heads, dropout=0.1, r_max=128):
        [...]  # the rest of the constructor is unchanged
        self.biases = nn.Parameter(torch.zeros(num_heads, 2 * r_max - 1))
    [...]  # the rest of the class is unchanged for now
```

Notice the new argument `r_max`, which defines the maximum relative distance (any larger distance will be clamped). There are $2r_{max} - 1$ possible relative distances. Notice that our `biases` parameter contains one set of biases for each attention head: indeed, bias-based RPE does not usually share biases across attention heads or MHA layers. All the biases are initialized to zero.

Next, let's create a method that will take L_q and L_k (the length of the query and key, respectively), and it will create a tensor of shape $[h, L_q, L_k]$ containing the appropriate bias for each pair of query and key token positions, depending on their relative distance. The output tensor (which I will call `b`) will contain many repetitions, since all pairs of tokens with the same relative distance have the same bias. For example, `b[0, 4, 10]` is equal to `b[0, 5, 11]` and `b[0, 6, 12]`, and so on. However, `b[0, 4, 10]` is *not* equal to `b[2, 4, 10]`, since these biases are for two different attention heads.

2 Zihang Dai et al., "Transformer-XL: Attentive Language Models Beyond a Fixed-Length Context", arXiv preprint arXiv:1901.02860 (2019).

```
    def gather_biases(self, Lq, Lk):
        h, n_biases = self.biases.shape  # [h, 2 * r_max - 1]
        r_max = (n_biases + 1) // 2
        pos_q = torch.arange(Lq, device=self.biases.device)  # [0, ..., Lq - 1]
        pos_k = torch.arange(Lk, device=self.biases.device)  # [0, ..., Lk - 1]
        rel_pos = pos_q[:, None] - pos_k[None, :]  # [Lq, Lk] (contains i - j)
        rel_pos = rel_pos.clamp(-r_max + 1, r_max - 1) + r_max - 1  # [Lq, Lk]
        return self.biases[:, rel_pos]  # [h, Lq, Lk]
```

This method creates two tensors `pos_q` and `pos_k` containing all the query and key positions, respectively. It then uses broadcasting (`pos_q[:, None] - pos_k[None, :]`) to compute an $L_q \times L_k$ matrix containing the relative distance for each pair of query and key positions. We then clamp these relative distances to the range $-r_{max}$ to $+r_{max}$ (so at this stage `rel_pos` looks like Figure D-1), and we add $r_{max} - 1$ because the rows in the bias table are indexed from 0 to $2r_{max} - 1$ rather than from $-r_{max}$ to r_{max}. Lastly, we fetch all the biases from the `biases` parameter, for each pair of query and key position. This kind of complex indexing and broadcasting is often the most confusing part when looking at relative PE code.

Finally, we're ready to tweak the `forward()` method to add the gathered biases to the attention scores:

```
    def forward(self, query, key, value):
        [...]  # the rest is unchanged
        scores = q @ k.transpose(2, 3) / self.d**0.5  # (B, h, Lq, Lk)
        b = self.gather_biases(query.size(-2), key.size(-2))  # (h, Lq, Lk)
        scores = scores + b
        [...]  # the rest is unchanged
```

And that's all, our modified multi-head attention class is ready to be used!

FlashAttention (introduced in Chapter 17) does not currently support passing arbitrary attention biases, so it cannot be used along with RPE.

Now let's turn to the next relative PE technique: RoPE.

Rotary Position Embeddings (RoPE)

RoPE was proposed in April 2021[3] by a team of researchers from the Chinese AI company Zhuiyi Technology. Surprisingly, this method works without adding any

3 Jianlin Su et al., "RoFormer: Enhanced Transformer with Rotary Position Embedding", arXiv preprint arXiv:2104.09864 (2021).

parameters to the model: it just rotates each query token and each key token by an angle proportional to its absolute position.

Its *absolute* position? How can this be a relative PE technique if it's based on absolute positions? Well, let's pretend the tokens live in 2D space, and consider a query token **q** at position 19, and a key token **k** at position 5. Figure D-2 shows how the query token gets rotated by an angle of 19θ and becomes $\mathbf{q}'$, while the key token gets rotated by an angle of 5θ and becomes $\mathbf{k}'$ (we'll come back to the increment angle θ shortly). Let's call α the angle between **k** and **q**, and α' the angle between $\mathbf{k}'$ and $\mathbf{q}'$. If you look carefully at this diagram,[4] you will see that $\alpha' = \alpha + (19 - 5)\theta$. That's where the relative distance $r = i - j$ comes in: $\alpha' = \alpha + r\theta$.

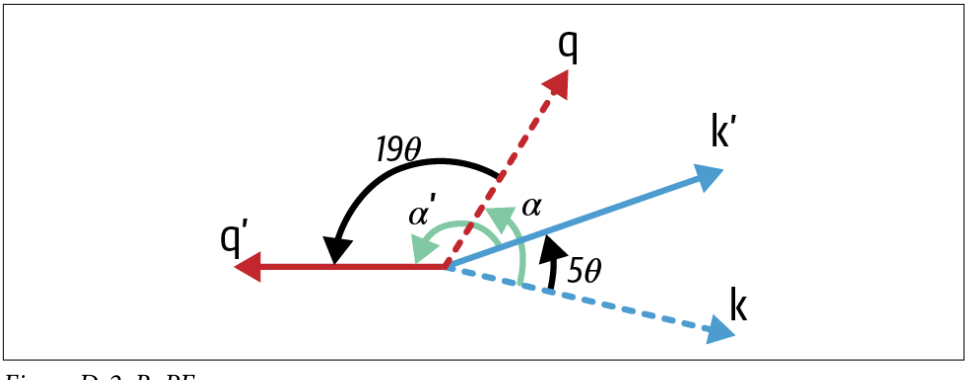

Figure D-2. RoPE

Why does this matter? Well, recall that the attention equation computes the dot product $\mathbf{q}'\cdot\mathbf{k}'$, which is equal to $\mathbf{q}'\cdot\mathbf{k}' = \cos(\alpha')\times\|\mathbf{q}'\|\times\|\mathbf{k}'\|$: that's the cosine of α' times the length of each vector. Since rotating a vector doesn't affect its length, we have $\|\mathbf{q}'\| = \|\mathbf{q}\|$ and $\|\mathbf{k}'\| = \|\mathbf{k}\|$, so putting everything together we obtain $\mathbf{q}'\cdot\mathbf{k}' = \cos(\alpha + r\theta)\times\|\mathbf{q}\|\times\|\mathbf{k}\|$. Without RoPE, we would have computed $\mathbf{q}\cdot\mathbf{k} = \cos(\alpha)\times\|\mathbf{q}\|\times\|\mathbf{k}\|$ instead. As you can see, the only difference is that RoPE adds $r\theta$ to the angle α. The absolute positions play no role, they have vanished, and we are left with a tweak that only depends on the relative distance between the tokens. Isn't that pretty?

However, this trick only works in 2D space, but the tokens are high-dimensional. This is why RoPE groups the token dimensions into pairs, forming 2D subspaces, then it rotates each subspace. Moreover, it uses a different angle θ_k for each subspace k: the RoPE authors proposed using $\theta_k = B^{-2k/d}$ where the base B is a constant (typically equal to 10,000), and d is the dimensionality of the token. This equation makes θ_k smaller and smaller as k increases, so the first subspaces will get more rotations

4 The angle between $\mathbf{k}'$ and **q** is $\alpha - 5\theta$, and the angle between **q** and $\mathbf{q}'$ is 19θ. If we add both angles, we get $\alpha' = \alpha - 5\theta + 19\theta = \alpha + (19 - 5)\theta$.

per position increment than the last subspaces. Therefore, the first subspaces will be useful to model nearby relationships between tokens, while the last will be good for distant relationships.

Let's implement RoPE in PyTorch. First, we will define a function that precomputes all the cosines and sines of angles $p\theta_k$ for each possible pair of absolute position p and subspace k:

```
def precompute_rope_cos_sin(d, max_len, base=10_000):
    theta = base ** (-torch.arange(0, d, 2).float() / d)  # θ_k, shape: [d // 2]
    positions = torch.arange(max_len)  # p, shape: [max_len]
    freqs = torch.outer(positions, theta)  # p * θ_k, shape: [max_len, d // 2]
    freqs_twice = freqs.repeat_interleave(2, dim=-1)  # shape: [max_len, d]
    return freqs_twice.cos(), freqs_twice.sin()  # shape: both [max_len, d]

d, max_len = 64, 4_096
cos_theta, sin_theta = precompute_rope_cos_sin(d, max_len)
```

We could compute these values on the fly to avoid setting a maximum sequence length, but the transformer would be significantly slower. Instead, we can use a large maximum length, since the memory cost isn't too bad.

Going Beyond the Maximum Sequence Length

When a transformer imposes a maximum length, it's still possible to force it to process longer sequences.

The simplest way to do that (other than cropping the sequence) is to linearly interpolate the token positions to the allowed range. This is called *position interpolation*. For example, if the maximum length is 4,096 but we want to process a sequence of length 5,000, we can divide each position by s = 5,000 / 4,096, rounding down. This technique works with any transformer that imposes a maximum sequence length (including the original Transformer). Unfortunately, quality drops as the input sequence gets longer.

When using RoPE, a much better approach is to use *NTK-Aware Scaled RoPE*, proposed in 2023 by user bloc97 in a Reddit post: it works by scaling the base B by a factor $\lambda = s^{2/(d-2)}$, where s is the sequence length divided by the maximum length (just like in position interpolation), and d is the dimensionality. For example, if the input sequence has 5,000 tokens and the maximum length is 4,096, and d = 64, then $\lambda \approx 1.0065$, therefore $B = 10{,}000\lambda = 10{,}065$. This technique has proven to work extremely well, making it possible to process much longer sequences than the maximum sequence length.

At the time of writing, one of the best RoPE variants is YaRN,[5] which tweaks the base B like NTK-Aware Scaled RoPE, and combines it with position interpolation, while also scaling the attention score to counteract the loss of precision that can occur during interpolation.

Next, let's define the function that will apply RoPE to a given tensor. This implementation rotates all of the 2D subspaces simultaneously, for every token in the tensor, across all sequences in the batch, and all attention heads. To understand how this works, first note that rotating a 2D vector (x, y) by an angle θ produces the vector $(x\cos(\theta) - y\sin(\theta), y\cos(\theta) + x\sin(\theta))$. If we focus on a single token $\mathbf{t}$ located at position p, we can treat its vector components as a sequence of 2D coordinates: $\mathbf{t} = (x_0, y_0, x_1, y_1, \ldots)$. Then we can create a copy and tweak it to get $\mathbf{t'} = (-y_0, x_0, -y_1, x_1, \ldots)$, and finally compute $\mathbf{t} \otimes \mathbf{cos} + \mathbf{t'} \otimes \mathbf{sin}$ where **cos** and **sin** are the precomputed cosines and sines. This will give us the desired result, where every subspace is rotated by the appropriate angle: $(x_0\cos(p\theta_0) - y_0\sin(p\theta_0), y_0\cos(p\theta_0) + x_0\sin(p\theta_0), x_1\cos(p\theta_1) - y_1\sin(p\theta_1), y_1\cos(p\theta_1) + x_1\sin(p\theta_1), \ldots)$.

```
def rope_rotation(t, cos_theta, sin_theta):
    t_grouped = t.reshape(*t.shape[:-1], -1, 2)  # group pairs of dims
    t_swapped = t_grouped[..., [1, 0]]  # swap 2D axes: (x, y) -> (y, x)
    t_swapped[..., 0] *= -1  # for each pair, (y, x) -> (-y, x)
    t_rotated_half = t_swapped.flatten(start_dim=-2)  # [-y0, x0, -y1, x1,...]
    L = t.size(-2)
    return t * cos_theta[:L] + t_rotated_half * sin_theta[:L]  # same shape as t
```

Let's apply RoPE to a query and key:

```
batch_size, n_heads, Lq, Lk, d = 32, 8, 800, 800, 64
Q = torch.randn(batch_size, n_heads, Lq, d)
K = torch.randn(batch_size, n_heads, Lk, d)
Q_rope = rope_rotation(Q, cos_theta, sin_theta)
K_rope = rope_rotation(K, cos_theta, sin_theta)
```

Alternatively, you can use an off-the-shelf implementation, such as `LlamaRotaryEmbedding` in the Hugging Face Transformers library, or `RotaryEmbedding` from the rotary-embedding-torch library.

Combining relative PE techniques with sparse attention or approximate attention techniques can be a bit tricky. For example, using RoPE along with multi-head latent attention (MLA) and KV caching (see Chapter 17) is not trivial. Indeed, the KV cache stores the latent key/value input $\mathbf{L}_{KV}$ and recomputes the full projected key $\mathbf{K}$ and

5 Bowen Peng et al., "YaRN: Efficient Context Window Extension of Large Language Models", arXiv preprint arXiv:2309.00071 (2023).

value **V** (including the past tokens) each time we generate a new token. We could apply RoPE to **K** and **V** on the fly, but it would be computationally expensive. So instead, the MLA authors proposed a way to keep the content and position information separate. This allows them to calculate a lightweight positional score based on the new token's query position relative to all previous key positions, avoiding the expensive re-computation.

In short, RoPE has many advantages: it doesn't use any extra parameters, it's fairly fast to compute (especially when precomputing the cosines and sines), it's compatible with FlashAttention, and it gives excellent results. It is currently the most popular positional encoding method along with ALiBi.

Attention with Linear Biases (ALiBi)

ALiBi was proposed in August 2021 by Ofir Press et al.[6] Just like bias-based RPE, ALiBi adds a single bias to each attention score depending on the relative distance between tokens. However, unlike bias-based RPE (but just like RoPE), ALiBi doesn't use any trainable parameters at all. Indeed, the bias is computed using the equation: $b_{a,i,j} = -m_a|i - j|$, where a is the attention head, i and j are the positions of the query and key tokens, and m_a is the slope used in the attention head a. The authors proposed using the slope $m_a = 2^{-8a/h}$, where h is the total number of attention heads, and a is 1-indexed. For example, the first attention head in an 8-head model uses the slopes $m_1 = 1/2$, $m_2 = 1/4$, …, $m_8 = 1/256$. As you can see, the bias simply decreases linearly as tokens get further away from each other (hence the name of this method). Since each attention head uses a different slope m_a, some heads focus on local relationships while others focus on distant relationships.

The following function generates the ALiBi biases for a given number of heads and a given sequence length. The output tensor has a shape of `[n_heads, seq_len, seq_len]`:

```
def get_alibi_biases(n_heads, seq_len):
    head = torch.arange(1, n_heads + 1, dtype=torch.float32)  # shape: [n_heads]
    slopes = torch.pow(2, -8 * head / n_heads)  # [n_heads]
    pos = torch.arange(seq_len)  # [seq_len]
    distance_matrix = -(pos[None, :] - pos[:, None]).abs()  # [seq_len, seq_len]
    return slopes.view(-1, 1, 1) * distance_matrix
```

Let's compute the biases for an 8-head model and a sequence of 4 tokens, and look at the biases for the first attention head:

```
>>> biases = get_alibi_biases(n_heads=8, seq_len=4)
>>> biases[0]
```

6 Ofir Press et al., "Train Short, Test Long: Attention with Linear Biases Enables Input Length Extrapolation", arXiv preprint arXiv:2108.12409 (2021).

```
tensor([[ 0.0000, -0.5000, -1.0000, -1.5000],
        [-0.5000,  0.0000, -0.5000, -1.0000],
        [-1.0000, -0.5000,  0.0000, -0.5000],
        [-1.5000, -1.0000, -0.5000,  0.0000]])
```

That's all! Now you can just add the biases to the attention scores, just like we did with bias-based RPE.

The flash-attn library offers optimized FlashAttention-2 kernels for RoPE and ALiBi.

So when should you use Bias RPE, RoPE, or ALiBi? In a nutshell, Bias RPE is still common and performs well, especially when the input sequences have a fixed size; however, this technique does not extrapolate well beyond the maximum sequence length, and it does not work with FlashAttention. As a result, most modern transformers use either RoPE or ALiBi instead. For the highest quality, RoPE is usually the best choice. However, ALiBi is better at extrapolating to longer sequences: this allows you to "train short, test long". That said, things are moving quickly!

To conclude, Table D-1 compares the regular scaled dot-product attention equation with each of the equations used with the relative PEs discussed in this appendix.

Table D-1. Attention score equation depending on the positional encoding method used

Method	Attention score $e_{i,j}$	Output $\mathbf{o}_i$
Regular	$\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}}$	$\sum_{j=1}^{l_k} a_{i,j}\mathbf{v}_j$
RPE (full)	$\frac{\mathbf{q}_i \cdot \left(\mathbf{k}_j + \mathbf{a}^K_{i-j}\right)}{\sqrt{d_k}}$	$\sum_{j=1}^{l_k} a_{i,j}\left(\mathbf{v}_j + \mathbf{a}^V_{i-j}\right)$
RPE (bias)	$\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} + b_{j-i}$	$\sum_{j=1}^{l_k} a_{i,j}\mathbf{v}_j$
RoPE	$\frac{(R(i\theta)\mathbf{q}_i) \cdot \left(R(j\theta)\mathbf{k}_j\right)}{\sqrt{d_k}}$	$\sum_{j=1}^{l_k} a_{i,j}\mathbf{v}_j$
ALiBi	$\frac{\mathbf{q}_i \cdot \mathbf{k}_j}{\sqrt{d_k}} + m_a \cdot \lvert i-j \rvert$	$\sum_{j=1}^{l_k} a_{i,j}\mathbf{v}_j$