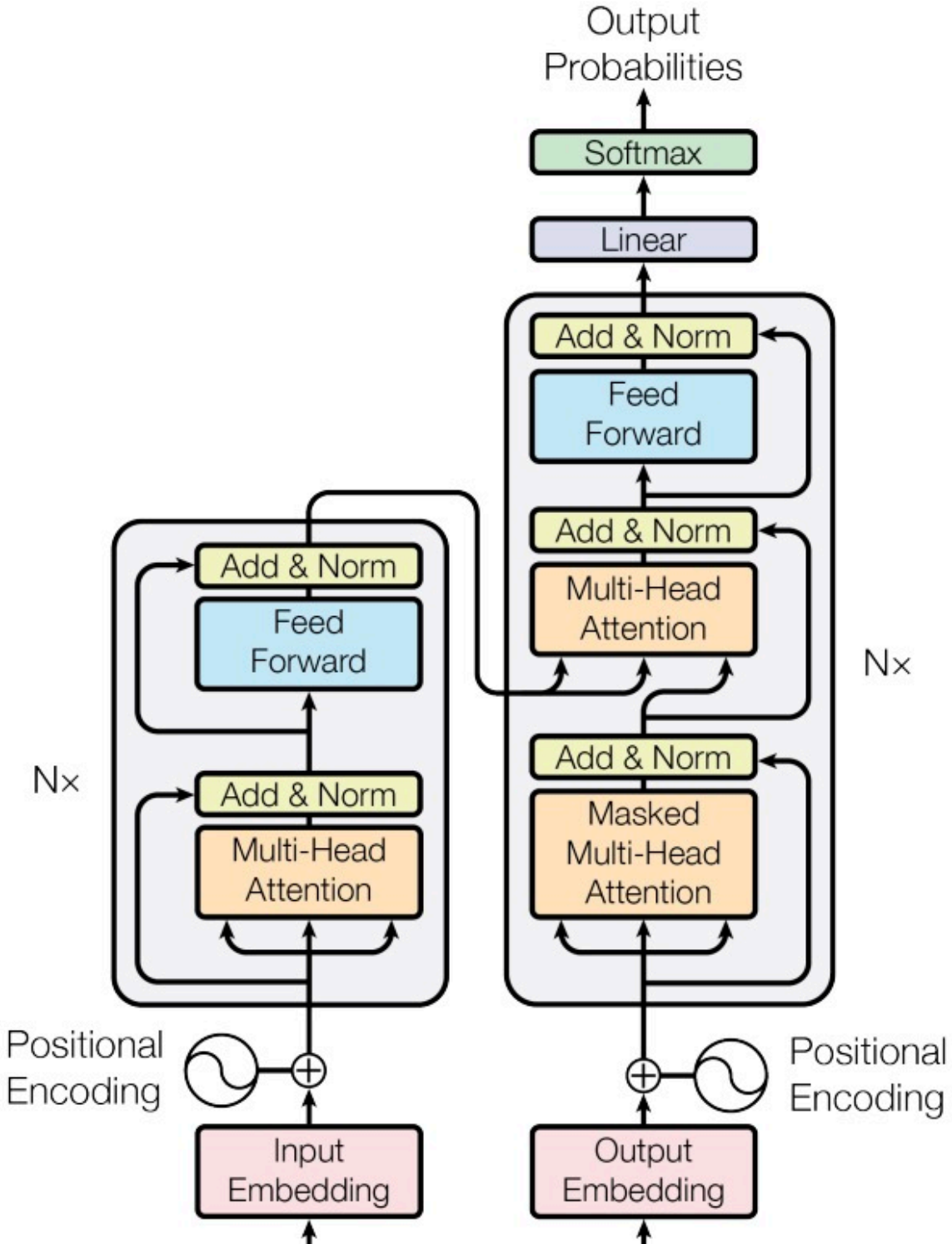
Implementing a Vision Transformer (ViT) in Julia!

The **Transformer** architecture, introduced in the paper *Attention Is All You Need* by Vaswani et al. (2017), is the most ubiquitous neural network architecture in modern machine learning. Its parallelism and scalability to large problems has seen it adopted in domains beyong those it was traditionally considered for (sequential data).


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding

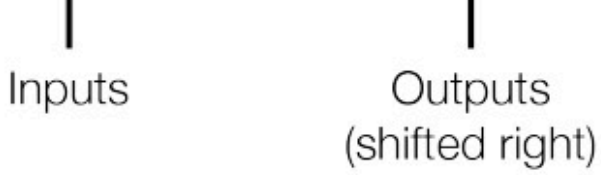


Figure 1: The Transformer - model architecture.

NOTE: We adapt/borrow a lot of material/concepts from Torralba, A., Isola, P., & Freeman, W. T. (2021, December 1). *Foundations of Computer Vision*. MIT Press; The MIT Press, Massachusetts Institute of Technology.

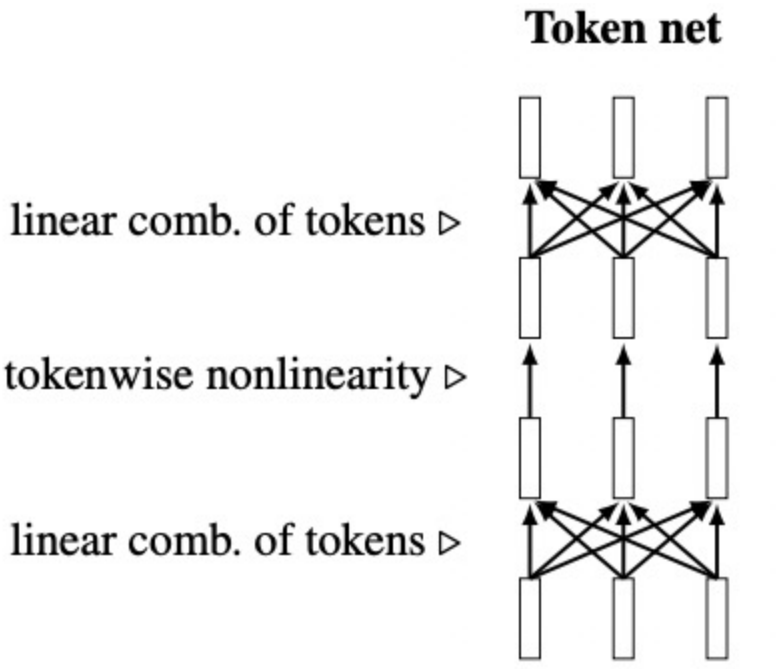
One thing to keep in mind throughout this notebook is that Transformers operate on **tokens**.

A sequence of tokens will be denoted by a matrix $\mathbf{T} \in \mathbb{R}^{N \times 1}$, in which each token in the sequence, $\mathbf{t}_1, \ldots, \mathbf{t}_N$, is transposed to become a row of the matrix:

$$\mathbf{T} = \begin{bmatrix} \mathbf{t}_1^\top \\ \vdots \\ \mathbf{t}_N^\top \end{bmatrix}$$

Conceptually, the transformer architecture may be thought of as a *token net*, which is abtraction or generalization of the more familiar *neural nets*.

Token nets are just like neural nets, alternating between layers that mix nodes in linear combinations (e.g., fully connected linear layers, convolutional layers, etc.) and layers that apply a pointwise nonlinearity to each node (e.g., relus, per-token MLPs).



The idea to keep in mind is that *tokens are to transformers as neurons are to neural nets*.

Until recently, the best performing models for image classification had been convolutional neural networks (CNNs) introduced in LeCun et al. (1998). Nowadays, transformer architectures have been shown to have similar to better performance. One such model, called Vision Transformer by Dosovitskiy et al. (2020) splits up images into regularly sized patches. The patches are treated as a sequence and attention weights are learned as in a standard transformer model.

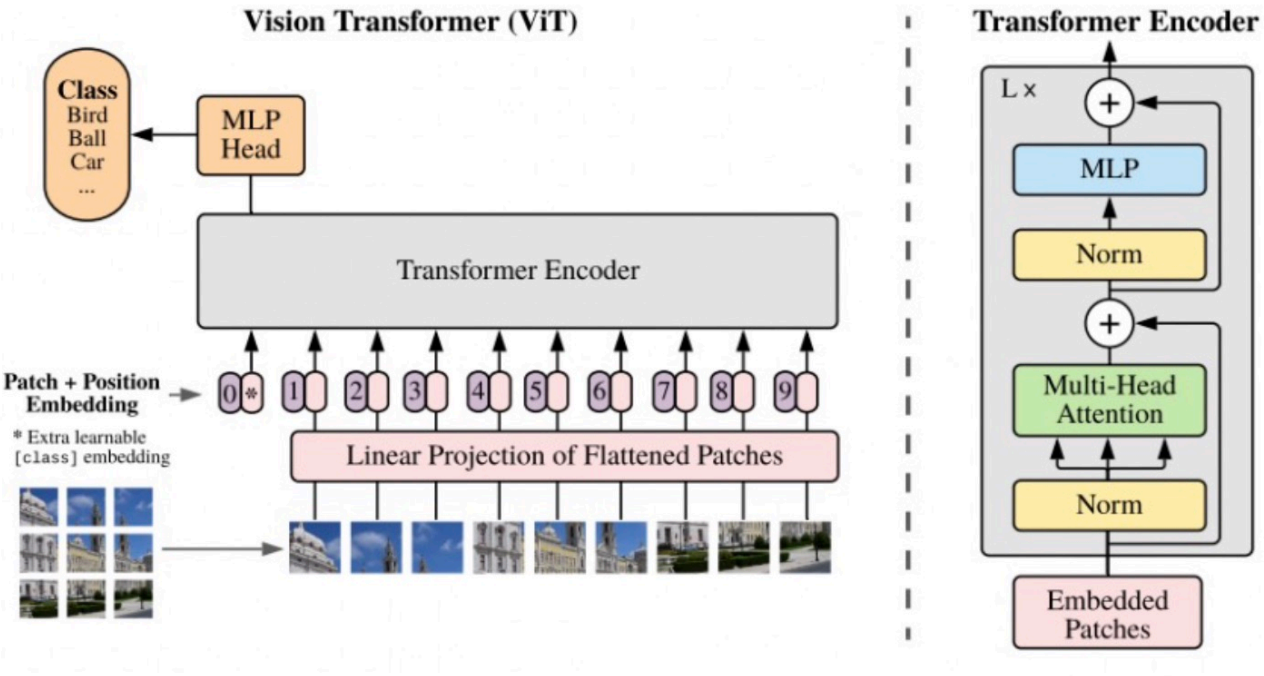


Figure 2: ViT Architecture, Figure from Dosovitskiy et al. [2020]

Transformers basics

In this section, we will learn how a basic transformer attention head works.

First, we must break the image up into tokens that can be processed by the transformer. Our tokens come from patches of the image.

```
visualize_patches (generic function with 1 method)
```

```
function visualize_patches(image, patch_size)
    # Extract patches from image
    patches = extract_patches(image, patch_size)

    # Calculate grid dimensions (assuming a square grid)
    n_patches = length(patches)
    grid_dim = ceil(Int, sqrt(n_patches))

    # Initialize a list to hold the individual patch plots
    plot_list = []

    # Generate the grid of patches
    for idx in 1:n_patches
        # Create a heatmap for each patch without axis or colorbar
        p = heatmap(patches[idx], color=:grays, axis=false, colorbar=false)
        push!(plot_list, p)
    end

    # Display the grid of patches with custom grid padding
    plot(
        plot_list...,
        layout=(grid_dim, grid_dim),
        margin=0mm,
        size=(900, 900),
    )
end
```

```
extract_patches (generic function with 1 method)
```

```
# Define function to extract patches
function extract_patches(image, patch_size)
    patches = []
    for i in 1:patch_size:size(image, 1)
        for j in 1:patch_size:size(image, 2)
            try
                patch = view(image, i:i+patch_size-1, j:j+patch_size-1)
                push!(patches, patch)
            catch
                temp_img = colorview(RGB, permutedims(image, (3, 2, 1)))
                patch = view(temp_img, i:i+patch_size-1, j:j+patch_size-1)
                push!(patches, patch)
            end
        end
    end
    return patches
end
```

We will use the CIFAR dataset here

```
dataset CIFAR10:
  metadata  =>    Dict{String, Any} with 2 entries
  split     =>    :test
  features  =>    32×32×3×10000 Array{Float32, 4}
  targets   =>    10000-element Vector{Int64}
```

```
begin
    train_dataset = CIFAR10(dir="cifar/", split=:train)
    test_dataset = CIFAR10(dir="cifar/", split=:test)
end
```

1

```
begin
    println("CIFAR10 index")
    @bind cifar_img_num Slider(1:1:1000, show_value=true)
end
```

CIFAR10 index

First, we will break our image up into patches. We can choose different patch sizes.

These patches are the basis for our tokens (in practice using some sort of embedding discussed later).

```
md"""
First, we will break our image up into patches. We can choose different patch sizes.

These patches are the basis for our tokens (in practice using some sort of embedding discussed later).
"""
```

16

patch size

"frog"

```
extract_patches_num (generic function with 1 method)
```

The attention aspect involves queries, keys, and values.

The queries are like the question you are asking, which should direct the attention. The keys tell you how much each token contributes in the query.

The attention matrix is essentially the multiplication of the query and key matrices.

The value matrix maps the attention matrix to the new token values in the next layer.

```
md"""
The attention aspect involves queries, keys, and values.

The queries are like the question you are asking, which should direct the attention.
The keys tell you how much each token contributes in the query.

The attention matrix is essentially the multiplication of the query and key matrices.

The value matrix maps the attention matrix to the new token values in the next layer.

---
"""
```

Example with prefixed keys and queries set to mean color of patch (in reality they are learned during training and we don't know exactly what they are)

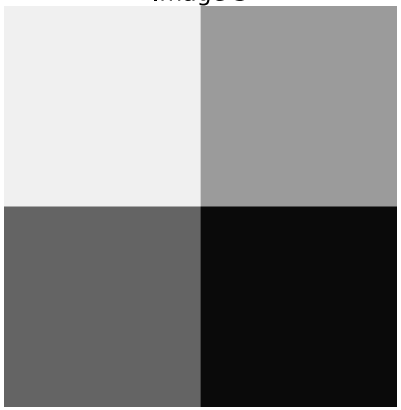
1

patch selector

Image 1

Image 2

Image 3



The attention matrix comes from the product of the key and query matrices, multiplied by the tokens.

Finally, the next layer of tokens is calculated by multiplying the attention matrix and the value matrix.

Building a transformer

Let's start by defining key components of a **Transformer** model using Julia structs and parametric types, similar to the structure we implemented in *Homework 3*.

We will implement the `AttentionHead`, `MultiHeadedAttention`, and `FeedForwardNetwork` modules as Julia structs. This will set up the parts which get combined together in the `Transformer` model.

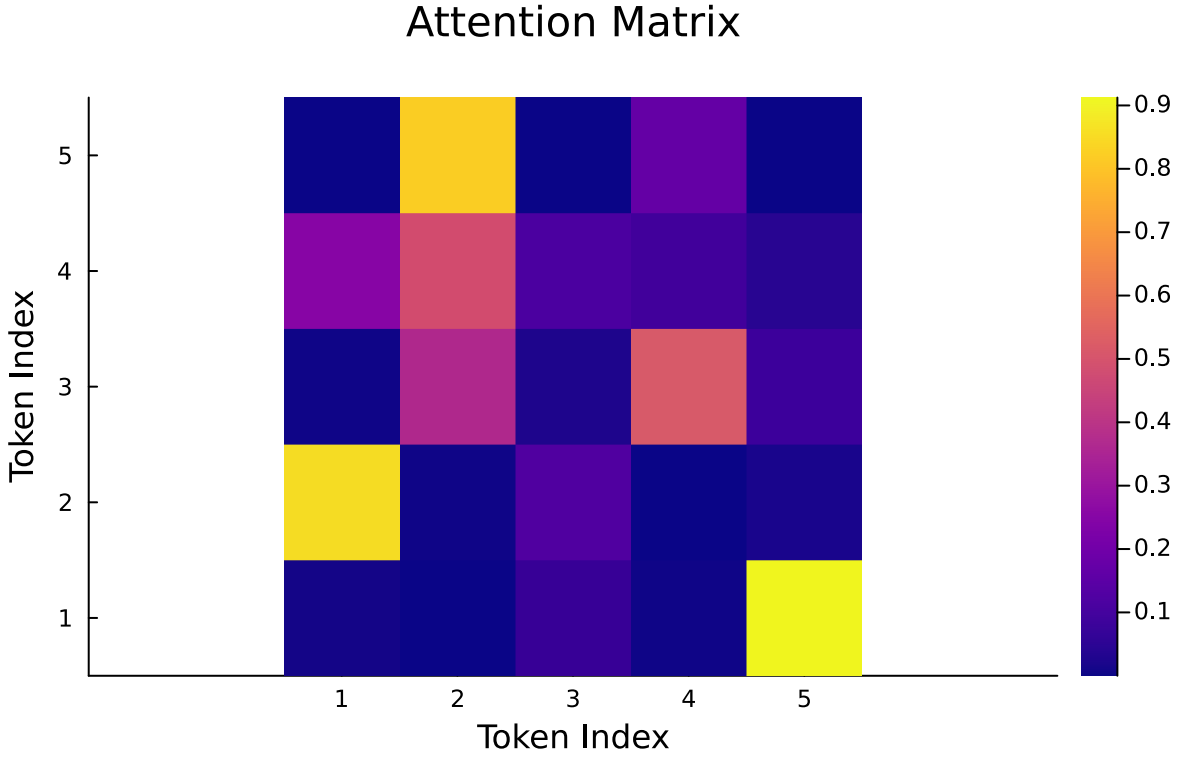
```
softmax (generic function with 1 method)
```

```julia
# Stable softmax implementation
function softmax(x; dims=1)
    exp_x = exp.(x .- maximum(x, dims=dims))  # stability trick
    return exp_x ./ sum(exp_x, dims=dims)
end
```

```julia
### 1. Attention Head
struct AttentionHead{T<:Real}
    W_K::Matrix{T} # Shape: (n_hidden, dim)
    W_Q::Matrix{T} # Shape: (n_hidden, dim)
    W_V::Matrix{T} # Shape: (dim, dim)
    n_hidden::Int # dimensionality of key and query vectors

    function AttentionHead{T}(dim::Int, n_hidden::Int) where T<:Real
        return new{T}(randn(T, n_hidden, dim), randn(T, n_hidden, dim), randn(T, dim, dim), n_hidden)
    end

    function (head::AttentionHead{T})(X::Array{T}, attn_mask::Union{Nothing, Matrix{T}}=nothing) where {T<:Real}
        # X is expected to be an input token matrix with shape (N, dim)
        # Project input tokens to query, key, and value representations
        Q = X * transpose(head.W_Q)  # Shape: (N, n_hidden)
        K = X * transpose(head.W_K)  # Shape: (N, n_hidden)
        V = X * transpose(head.W_V)  # Shape: (N, dim)

        # Compute scaled dot-product attention
        scores = Q * transpose(K) / sqrt(head.n_hidden)  # Shape: (N, N)

        # Apply attention mask if provided
        if attn_mask !== nothing
            scores = scores .* attn_mask .+ (1 .- attn_mask) * -Inf
        end

        # Apply softmax along the last dimension
        alpha = softmax(scores, dims=ndims(scores))  # Shape: (N, N)

        # Compute attention output as weighted sum of values
        attn_output = alpha * V  # Shape: (N, dim)

        # attn_output is the (N, dim) output token matrix
        # alpha is the (N, N) attention matrix
        return attn_output, alpha
    end
end
```

5

number of tokens




```
Test `AttentionHead` implementation
attention output shape: (5, 3)
attention weight shape: (5, 5)
```

To compute the query, key, and value for a set of input tokens, $\mathbf{T}_{\text{in}}$, we apply the same linear transformations to each token in the set, resulting in matrices $\mathbf{Q}_{\text{in}}, \mathbf{K}_{\text{in}} \in \mathbb{R}^{N \times m}$ and $\mathbf{V}_{\text{in}} \in \mathbb{R}^{N \times d}$, where each row is the query/key/value for each token:

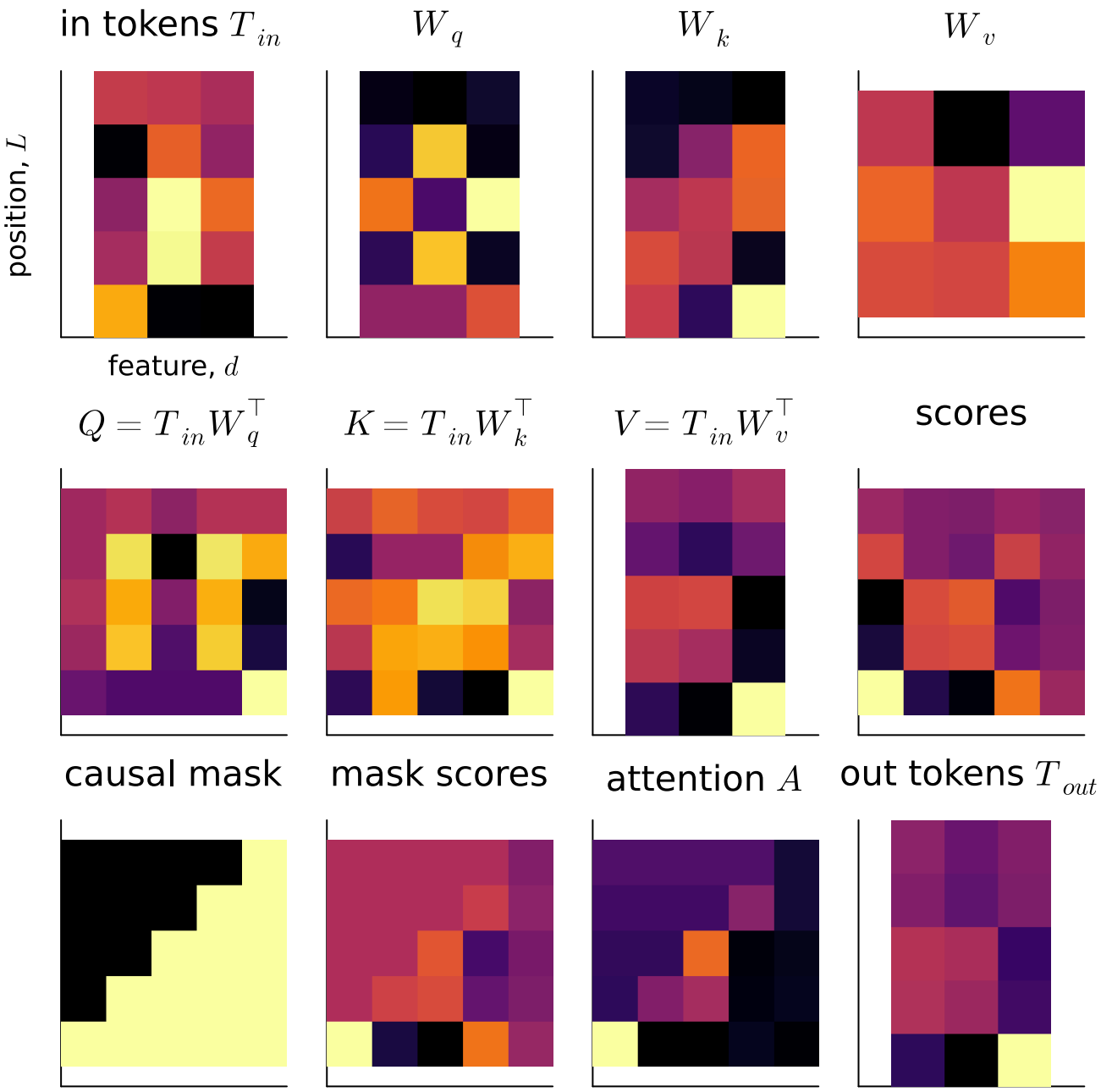
$$
\begin{aligned}
\mathbf{Q}_{\text{in}} &= \begin{bmatrix} \mathbf{q}_1^\top \\ \vdots \\ \mathbf{q}_N^\top \end{bmatrix} = \begin{bmatrix} (\mathbf{W}_q \mathbf{t}_1)^\top \\ \vdots \\ (\mathbf{W}_q \mathbf{t}_N)^\top \end{bmatrix} = \mathbf{T}_{\text{in}} \, \mathbf{W}_q^\top \quad \triangleleft \quad \textbf{query matrix} \\
\mathbf{K}_{\text{in}} &= \begin{bmatrix} \mathbf{k}_1^\top \\ \vdots \\ \mathbf{k}_N^\top \end{bmatrix} = \begin{bmatrix} (\mathbf{W}_k \mathbf{t}_1)^\top \\ \vdots \\ (\mathbf{W}_{\mathbf{k}} \mathbf{t}_N)^\top \end{bmatrix} = \mathbf{T}_{\text{in}} \, \mathbf{W}_k^\top \quad \triangleleft \quad \textbf{key matrix} \\
\mathbf{V}_{\text{in}} &= \begin{bmatrix} \mathbf{v}_1^\top \\ \vdots \\ \mathbf{v}_N^\top \end{bmatrix} = \begin{bmatrix} (\mathbf{W}_v \mathbf{t}_1)^\top \\ \vdots \\ (\mathbf{W}_v \mathbf{t}_N)^\top \end{bmatrix} = \mathbf{T}_{\text{in}} \, \mathbf{W}_v^\top \quad \triangleleft \quad \textbf{value matrix}
\end{aligned}
$$

> Note that the query and key vectors must have the same dimensionality, m, because we take a dot product between them. Conversely, the value vectors must match the dimensionality of the token code vectors, d, because these are summed up to produce the new token code vectors.

Finally, we have the attention equation:

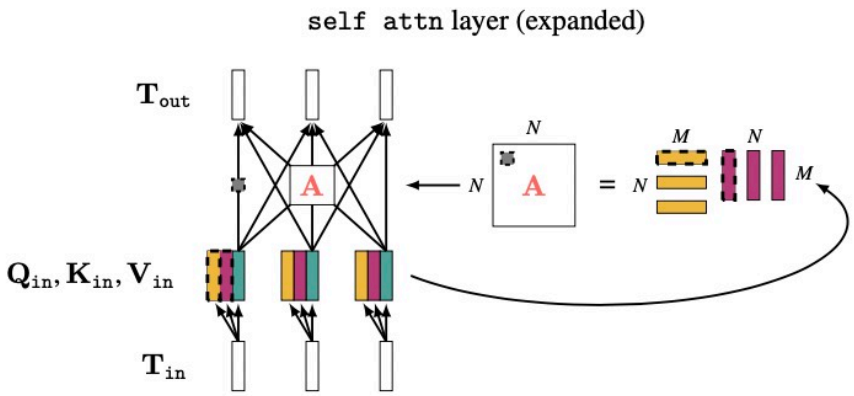
$$
\begin{aligned}
\mathbf{A} &= \text{softmax}\left(\frac{\mathbf{Q}_{\text{in}} \mathbf{K}_{\text{in}}^\top}{\sqrt{m}}\right) \quad \triangleleft \quad \textbf{attention matrix} \\
\mathbf{T}_{\text{out}} &= \mathbf{A}\mathbf{V}_{\text{in}}
\end{aligned}
$$

where the softmax is taken within each row (i.e., over the vector of matches for each separate query vector.

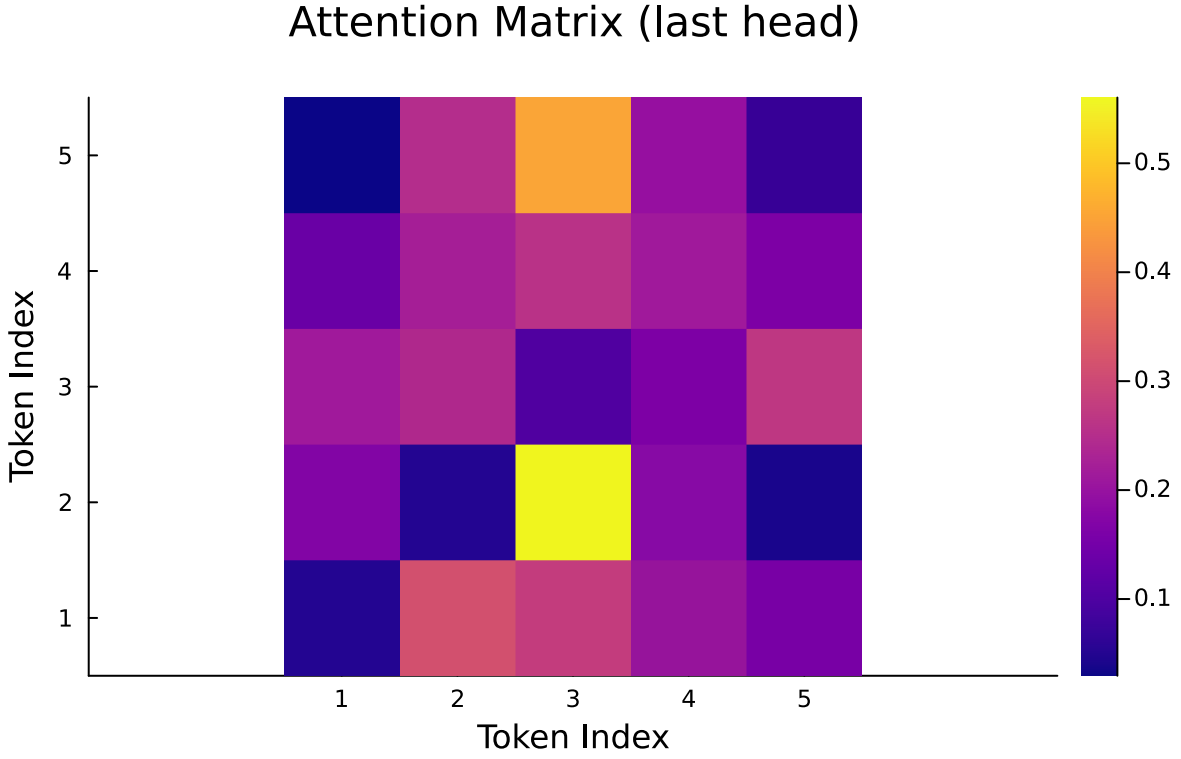


In expanded detail, here are the full mechanics of a self-attention layer, which is the kind of attention layer used in transformers.

Figure 26.10: Self-attention layer expanded. The nodes with the dashed outline correspond to each other; they represent one query being matched against one key to result in a scalar similarity value, in the gray box, which acts as a weight in the weighted sum computed by **A**.



```julia
### 2. Multi-Headed Attention
struct MultiHeadedAttention{T<:Real}
    heads::Vector{AttentionHead{T}}
    W_msa::Matrix{T} # Shape: (dim, num_heads*dim)

    function MultiHeadedAttention{T}(dim::Int, n_hidden::Int, num_heads::Int) where T<:Real
        # Each head outputs dim-dimensional tokens
        heads = [AttentionHead{T}(dim, n_hidden) for _ in 1:num_heads]
        # Our MHA outputs tokens with the same dimension as the input tokens
        W_msa = randn(T, dim, num_heads * dim)
        return new{T}(heads, W_msa)
    end

    function (mha::MultiHeadedAttention{T})(X::Array{T}, attn_mask::Union{Nothing, Matrix{T}}=nothing) where {T<:Real}
        outputs, alphas = [], []
        for head in mha.heads
            out, alpha = head(X, attn_mask) # Shapes: (N, dim), (N, N)
            push!(outputs, out)
            push!(alphas, alpha)
        end
        # Concatenate along hidden dimension
        concatenated = cat(outputs...; dims=2) # Shape: (N, num_heads*dim)
        attn_output = concatenated * transpose(mha.W_msa) # Shape: (N, dim)
        attn_alphas = cat(alphas...; dims=3) # Shape: (N, N, num_heads)
        attn_alphas = permutedims(attn_alphas, (3, 1, 2)) # Shape: (num_heads, N, N)
        return attn_output, attn_alphas
    end
end
```




```
Test `MultiHeadedAttention` implementation
attention output shape: (5, 3)
attention weights shape: (5, 5, 5)
```

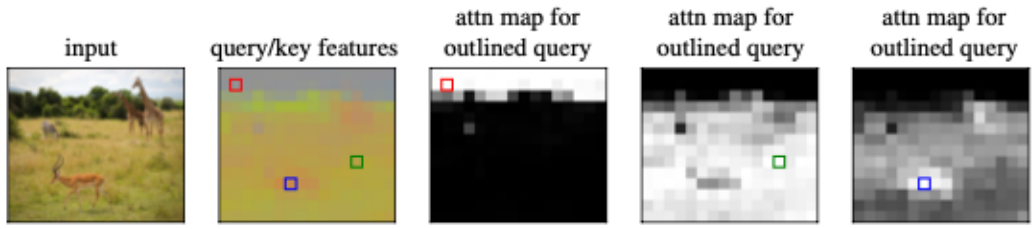


Figure 26.12: Example of self-attention maps where each token is an image patch and the query and key vectors are both set to the mean color of the patch, normalized to be a unit vector.

```julia
### 3. Linear struct
struct Linear{T<:Real}
    W::Matrix{T}  # Weight matrix
    b::Vector{T}  # Bias vector

    # Constructor: Initialize weights and biases
    function Linear{T}(in_features::Int, out_features::Int) where T<:Real
        W = randn(T, out_features, in_features) / sqrt(in_features)  # Xavier initialization
        b = zeros(T, out_features)
        return new{T}(W, b)
    end

    # Apply the linear transformation
    function (linear::Linear{T})(X::Array{T}) where T<:Real
        @assert size(X, 2) == size(linear.W, 2) "Input dimension must match the weight matrix"
        return X * transpose(linear.W) .+ linear.b'
    end
end
```

```
Test `Linear` implementation
Input shape: (20, 64)
Output shape: (20, 10)
```

```julia
### 4. FeedForwardNetwork struct
struct FeedForwardNetwork{T<:Real}
    layer1::Linear{T}  # First linear transformation
    layer2::Linear{T}  # Second linear transformation

    function FeedForwardNetwork{T}(dim::Int, n_hidden::Int) where T<:Real
        # Initialize the two linear layers
        layer1 = Linear{T}(dim, n_hidden)  # Shape: (n_hidden, dim)
        layer2 = Linear{T}(n_hidden, dim)  # Shape: (dim, n_hidden)
        return new{T}(layer1, layer2)
    end

    function (ffn::FeedForwardNetwork{T})(X::Array{T}) where {T<:Real}
        # Apply the first linear transformation
        X = ffn.layer1(X)  # Shape: (N, n_hidden)
        X = max.(0, X)     # ReLU activation
        # Apply the second linear transformation
        return ffn.layer2(X)  # Shape: (N, dim)
    end
end
```

```
Test `FeedForwardNetwork` implementation
Input shape: (5, 3)
Output shape: (5, 3)
```

Recap so far

1. `AttentionHead` **Implementation**:
    - Projects the input token matrix `X` (shape: (N, dim)) to query, key, and value matrices.
    - Computes scaled dot-product attention, applies an optional mask, and then applies softmax to get attention weights `alpha` with shape (N, N).
    - Returns the attention output `attn_output` (shape: (N, dim)) and attention weights `alpha`.
2. `MultiHeadedAttention` **Implementation**:
    - Creates multiple `AttentionHead` instances and collects their outputs.
    - Concatenates these outputs along the hidden dimension, applies a linear transformation (`W_msa`), and stacks the attention weights from each head into a 3D tensor with shape `(num_heads, N, N)`.

3. `FeedForwardNetwork` **(FFN) Implementation**:
    - A two-layer feed-forward network with an intermediate hidden layer of size `n_hidden`.
    - Projects the input token matrix `X` (shape: $(N, \mathbf{dim})$) to an intermediate hidden representation (shape: $(N, \mathbf{n\_hidden})$) using `W1` and `b1`, followed by a ReLU activation.
    - Transforms the hidden representation back to the original input dimension `dim` using `W2` and `b2`.
    - Returns the output with shape $(N, \mathbf{dim})$, maintaining the same dimension as the input tokens.

```julia
### 5. Attention Residual struct
struct AttentionResidual{T<:Real}
    attn::MultiHeadedAttention{T}  # Multi-headed attention mechanism
    ffn::FeedForwardNetwork{T}      # Feed-forward network

    # Constructor: initializes attention and feed-forward sub-layers
    function AttentionResidual{T}(dim::Int, attn_dim::Int, mlp_dim::Int,
    num_heads::Int) where T<:Real
        attn_layer = MultiHeadedAttention{T}(dim, attn_dim, num_heads)
        ffn_layer = FeedForwardNetwork{T}(dim, mlp_dim)
        return new{T}(attn_layer, ffn_layer)
    end

    # Apply the AttentionResidual block to input x
    function (residual::AttentionResidual{T})(X::Array{T}, attn_mask::Union{Nothing,
    Matrix{T}}=nothing) where {T<:Real}
        # Apply the multi-headed attention layer
        attn_out, alphas = residual.attn(X, attn_mask)  # attn_out: (N, dim),
        alphas: (num_heads, N, N)
        # First residual connection with attention output
        X = X .+ attn_out
        # Apply the feed-forward network and add the second residual connection
        X = X .+ residual.ffn(X)
        # Return the final output and attention weights
        return X, alphas
    end
end
```

```julia
### 6. Transformer struct
struct Transformer{T<:Real}
    layers::Vector{AttentionResidual{T}}  # Sequence of AttentionResidual blocks

    # Constructor: initializes a sequence of attention residual blocks
    function Transformer{T}(dim::Int, attn_dim::Int, mlp_dim::Int, num_heads::Int, num_layers::Int) where T<:Real
        layers = [AttentionResidual{T}(dim, attn_dim, mlp_dim, num_heads) for _ in 1:num_layers]
        return new{T}(layers)
    end

    # Apply the Transformer model to input X
    function (transformer::Transformer{T})(X::Array{T}; attn_mask::Union{Nothing, Matrix{T}}=nothing, return_attn::Bool=false) where T<:Real
        collected_alphas = []  # To store attention weights from each layer
        for layer in transformer.layers
            X, alphas = layer(X, attn_mask)  # Apply each residual block
            if return_attn
                push!(collected_alphas, alphas)  # Collect attention weights
            end
        end
        # Return the final output and collected attention weights from all layers (if required)
        if return_attn
            return X, collected_alphas
        else
            return X, nothing
        end
    end
end
```

Testing the `AttentionResidual` and `Transformer`

Let's test the `AttentionResidual` and `Transformer` structs to confirm that they work as expected with the previously implemented components.

```
Test `AttentionResidual` implementation
AttentionResidual output shape: (5, 8)
Attention weights shape (from one layer): (3, 5, 5)
```

```
Test `Transformer` implementation
Transformer output shape: (5, 8)
Collected attention weights shape (6 layers): [(3, 5, 5), (3, 5, 5), (3, 5, 5),
(3, 5, 5), (3, 5, 5), (3, 5, 5)]
```

Our modules so far build up the Transfomer

- **AttentionHead**: Implements a single attention head, creating query, key, and value projections, computing the attention scores, and applying a softmax.
- **MultiHeadedAttention**: Combines multiple `AttentionHead`s, concatenates their outputs, and applies a final linear transformation.
- **FeedForwardNetwork**: A simple feed-forward network with two linear layers and a ReLU activation in between.
- **AttentionResidual**: Combines multi-head attention and feed-forward network layers with residual connections.
- **Transformer**: Stacks multiple `AttentionResidual` layers to form the complete Transformer encoder.

What we've accomplished so far

Using *callable structs*, parametric types, and matrix operations_, we set up the basic components and combined them to create a Transformer module .

Let's view our Julia callable struct implemenations of the `AttentionHead` and `Transformer` modules side-by-side with a bare-bones implementation in PyTorch.

Side-by-side comparison:

AttentionHead implemented in **Python** (left) and **Julia** (right)

```python
class AttentionHead(nn.Module):
    def __init__(self, dim: int, n_hidden: int):
        super().__init__()

        self.W_K = nn.Linear(dim, n_hidden) # W_K weight matrix
        self.W_Q = nn.Linear(dim, n_hidden) # W_Q weight matrix
        self.W_V = nn.Linear(dim, n_hidden) # W_V weight matrix
        self.n_hidden = n_hidden

    def forward(self, x: torch.Tensor, attn_mask:
                Optional[torch.Tensor]) -> Tuple[torch.Tensor, torch.Tensor]:
        # x: input tensor of shape (B x T x dim)
        # attn_mask: optional mask tensor of shape (B x T x T)

        # ======= Answer START ========
        Q = self.W_Q(x)  # shape: (B x T x n_hidden)
        K = self.W_K(x)  # shape: (B x T x n_hidden)
        V = self.W_V(x)  # shape: (B x T x n_hidden)

        # Compute attention scores (QK^T)
        K_T = einops.rearrange(K, 'b t h -> b h t')  # Transpose key
        scores = torch.matmul(Q, K_T) / (self.n_hidden ** 0.5)  # (B x T x T)

        # Apply attention mask if provided
        if attn_mask is not None:
            scores = scores.masked_fill(attn_mask == 0, float('-inf'))

        # Compute attention weights (softmax)
        alpha = F.softmax(scores, dim=-1)  # (B x T x T)

        # Compute the attention output as a weighted sum of values
        attn_output = torch.matmul(alpha, V)  # (B x T x n_hidden)
        # ======= Answer END ========

        return attn_output, alpha
```

```julia
### 1. Attention Head
struct AttentionHead{T<:Real}
    W_K::Matrix{T} # Shape: (n_hidden, dim)
    W_Q::Matrix{T} # Shape: (n_hidden, dim)
    W_V::Matrix{T} # Shape: (dim, dim)
    n_hidden::Int # dimensionality of key and query vectors

    function AttentionHead{T}(dim::Int, n_hidden::Int) where T<:Real
        return new{T}(randn(T, n_hidden, dim), randn(T, n_hidden, dim), randn(T,
dim, dim), n_hidden)
    end

    function (head::AttentionHead{T})(X::Matrix{T}, attn_mask::Union{Nothing,
Matrix{T}}=nothing) where {T<:Real}
        # X is expected to be an input token matrix with shape (N, dim)
        # Project input tokens to query, key, and value representations
        Q = X * transpose(head.W_Q)  # Shape: (N, n_hidden)
        K = X * transpose(head.W_K)  # Shape: (N, n_hidden)
        V = X * transpose(head.W_V)  # Shape: (N, dim)

        # Compute scaled dot-product attention
        scores = Q * transpose(K) / sqrt(head.n_hidden)  # Shape: (N, N)

        # Apply attention mask if provided
        if attn_mask !== nothing
            scores = scores .* attn_mask .+ (1 .- attn_mask) * -Inf
        end

        # Apply softmax along the last dimension
        alpha = softmax(scores, dims=ndims(scores))  # Shape: (N, N)

        # Compute attention output as weighted sum of values
        attn_output = alpha * V  # Shape: (N, dim)

        # attn_output is the (N, dim) output token matrix
        # alpha is the (N, N) attention matrix
        return attn_output, alpha
    end
end
```

We don't handle batched inputs in the Julia implementation so assume batch_size (B) is 1.

Transformer implemented in **Python** (left) and **Julia** (right)

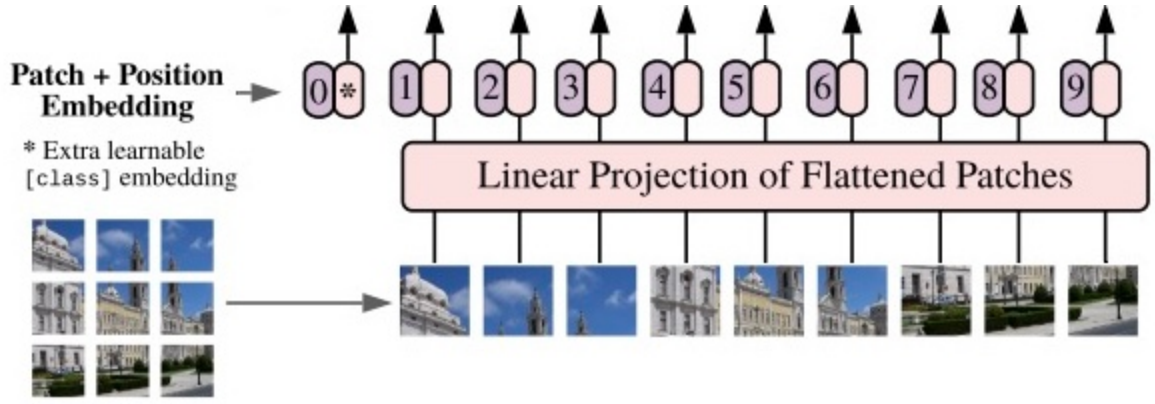
```python
class Transformer(nn.Module):
    def __init__(self, dim: int, attn_dim: int, mlp_dim: int, num_heads: int,
                 num_layers: int):
        # dim        the dimension of the input
        # attn_dim   the hidden dimension of the attention layer
        # mlp_dim    the hidden layer of the FFN
        # num_heads the number of heads in the attention layer
        # num_layers the number of attention layers.
        super().__init__()

        # TODO: set up the parameters for the transformer!
        #       You should set up num_layers of AttentionResiduals
        #       nn.ModuleList will be helpful here.

        # ======= Answer START ========
        self.num_layers = num_layers
        self.layers = nn.ModuleList([AttentionResidual(dim, attn_dim, mlp_dim,
                                                       num_heads) for _ in
                                     range(num_layers)])
        # ======= Answer END ========

    def forward(self, x: torch.Tensor, attn_mask: torch.Tensor,
                return_attn=False)-> Tuple[torch.Tensor, Optional[torch.Tensor]]:
        # x               the inputs. shape: (B x T x dim)
        # attn_mask       an attention mask. Pass this to each of the
        #                 AttentionResidual layers!
        #                 shape: (B x T x T)
        #
        # Outputs:
        # attn_output     shape: (B x T x dim)
        # attn_alphas     If return_attn is False, return None. Otherwise return
        #                 the attention weights
        #                 of each of each of the attention heads for each of
        #                 the layers.
        #                 shape: (B x Num_layers x Num_heads x T x T)

        output, collected_attns = None, None

        # TODO: Implement the transformer forward pass! Pass the input successively
        #       through each of the
        # AttentionResidual layers. If return_attn is True, collect the alphas
        # along the way.

        # ======= Answer START ========
        collected_attns = []
        for i in range(self.num_layers):
            x, attn = self.layers[i](x, attn_mask)
            if return_attn:
                collected_attns.append(attn)
        output = x
        if return_attn:
            collected_attns = einops.rearrange(collected_attns,
                                               'n b k t t2 -> b n k t t2')
        # ======= Answer END ========

        return output, collected_attns if return_attn else None
```

```julia
### 5. Transformer
struct Transformer{T<:Real}
    layers::Vector{AttentionResidual{T}}  # Sequence of AttentionResidual blocks

    # Constructor: initializes a sequence of attention residual blocks
    function Transformer{T}(dim::Int, attn_dim::Int, mlp_dim::Int, num_heads::Int,
num_layers::Int) where T<:Real
        layers = [AttentionResidual{T}(dim, attn_dim, mlp_dim, num_heads) for _ in
1:num_layers]
        return new{T}(layers)
    end

    # Apply the Transformer model to input X
    function (transformer::Transformer{T})(X::Matrix{T}, attn_mask::Union{Nothing,
Matrix{T}}=nothing) where {T<:Real}
        collected_alphas = []  # To store attention weights from each layer
        for layer in transformer.layers
            X, alphas = layer(X, attn_mask)  # Apply each residual block
            push!(collected_alphas, alphas)  # Collect attention weights
        end
        # Return the final output and collected attention weights from all layers
        return X, collected_alphas
    end
end
```

We don't handle batched inputs in the Julia implementation so assume batch_size (B) is 1.

But recall that what we want is to make is a **Vision Transformer**. This requires some additional layers for tokenizing an image. These are

- (1) patch embedding; and
- (2) positional encoding



We will explore these in detail next.

Patch Embedding

It turns out the patch embedding can be implemented by applying a strided convolution. However, we will take the more direct and visualizable approach of chopping up an image into patches and linearly projecting the vector that is the flattened patch to the desired dimensionality.

Remember, **transformers** *operate on tokens* i.e. the perform **transform**ations of tokens. What we are doing here is essentially the first step of *tokenizing* our image data.

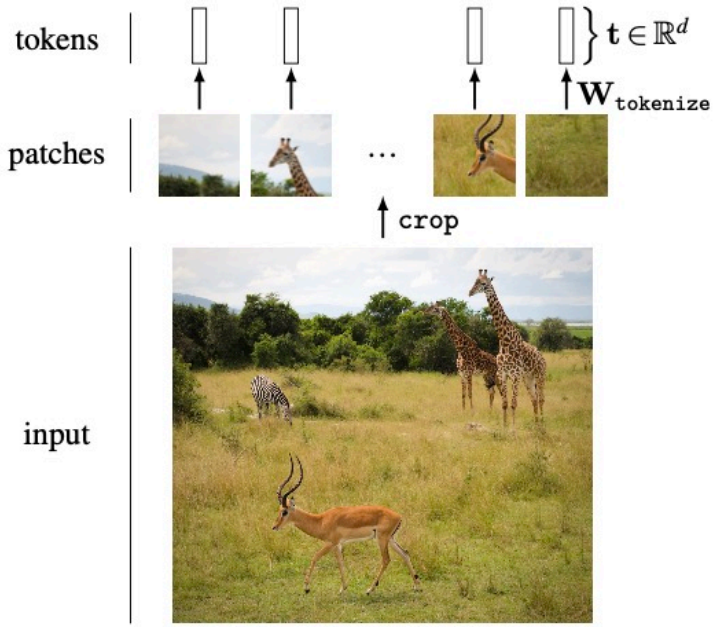
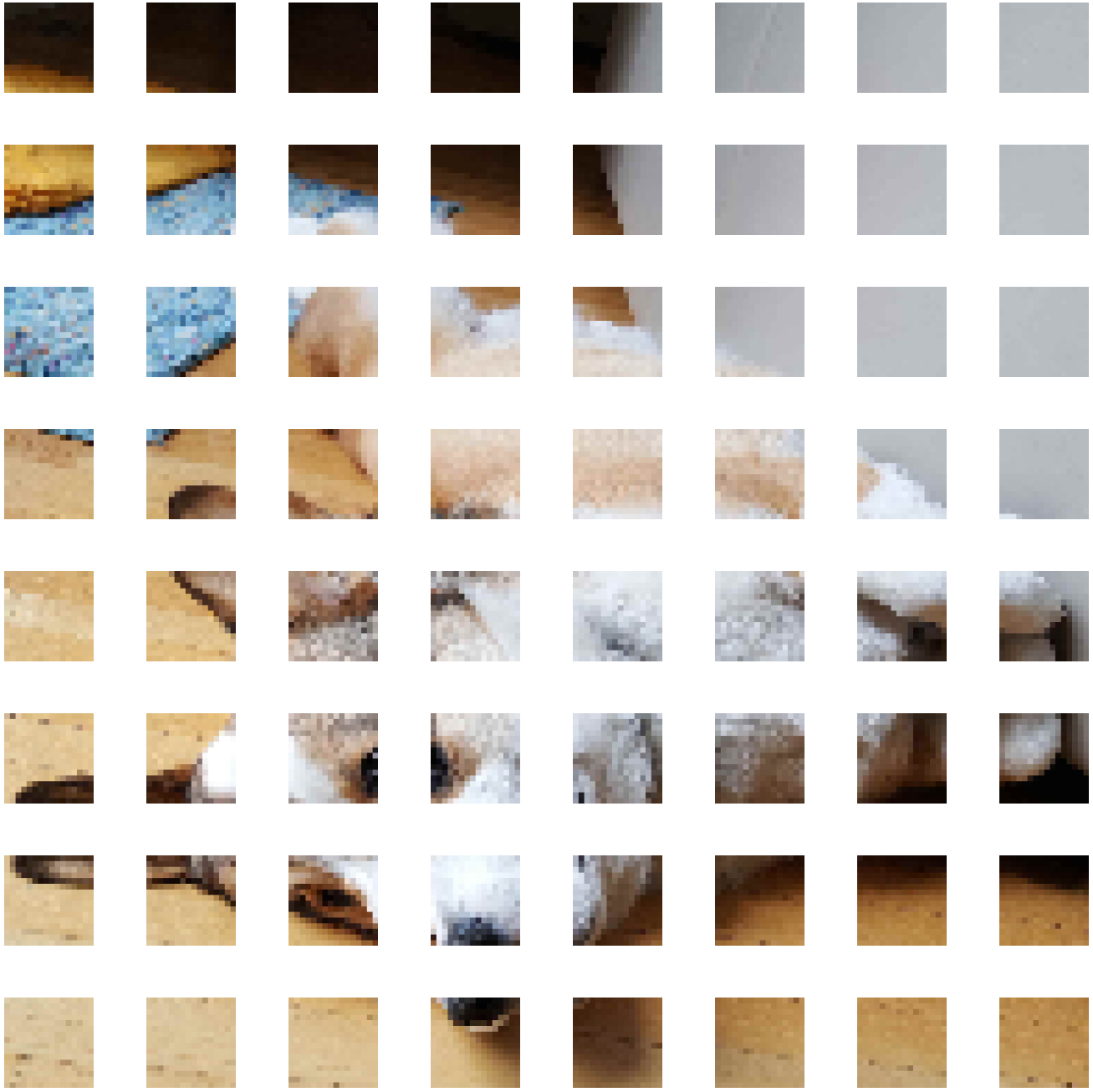


Figure 26.2: Tokenization: converting an image to a set of vectors. $\mathbf{W}_{\texttt{tokenize}}$ is a learnable linear projection from the dimensionality of the vectorized crops to d dimensions. This is just one of many possible ways to tokenize an image.

Let's illustrate patch embedding on an image of Philip.

16

patch size



A **patch embedding layer** is one that takes each of the image patches, like those displayed above, and then projects that into a vector. One approach is to simply flatten each patch and use a linear projection (using a matrix multiplication) to convert this into a vector. Since we are working on RGB

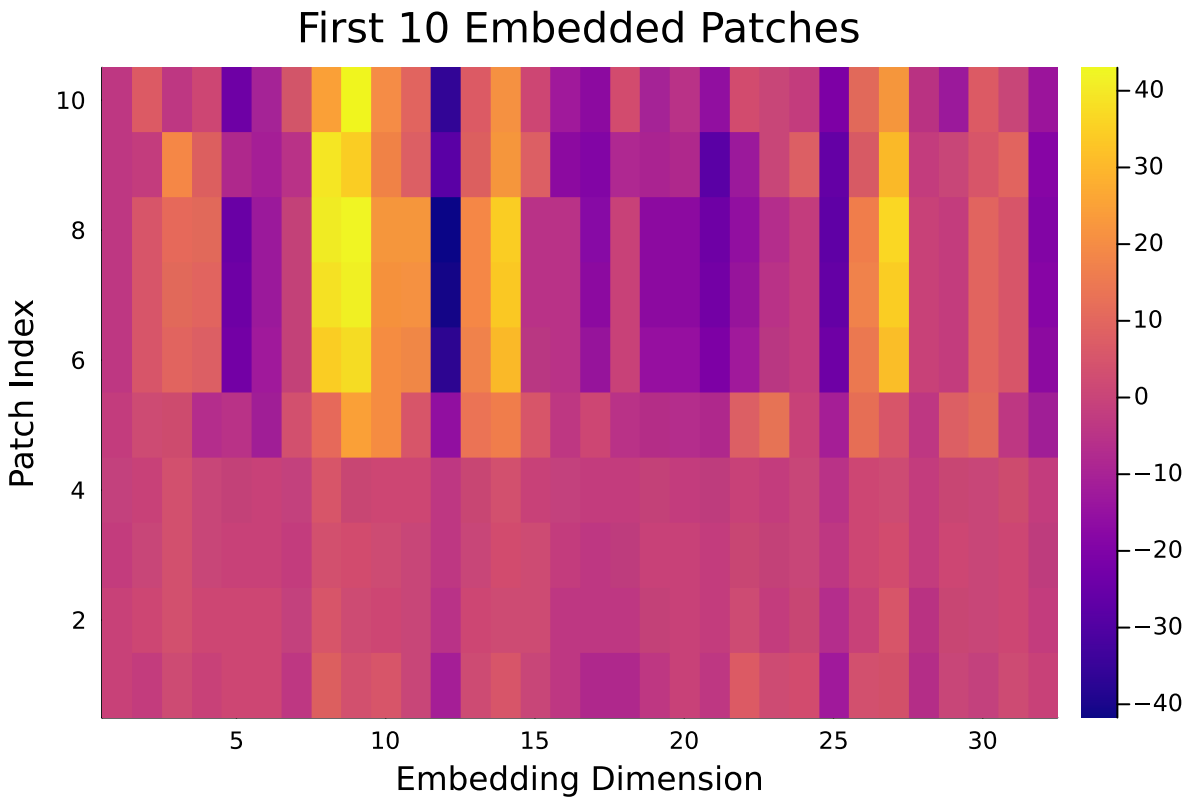
images (3 channels), we define a linear projection for each channel independently and then combine them.

```julia
### 7. PatchEmbedLinear
struct PatchEmbedLinear{T<:Real}
    img_size::Int
    patch_size::Int
    nin::Int  # Number of input channels (e.g., RGB → nin = 3)
    nout::Int # Desired output dimensionality for each patch
    num_patches::Int
    W::Vector{Matrix{T}} # Linear projection weights, one for each channel

    function PatchEmbedLinear{T}(img_size::Int, patch_size::Int, nin::Int, nout::Int) where T<:Real
        @assert img_size % patch_size == 0 "img_size must be divisible by patch_size"
        num_patches = (img_size ÷ patch_size)^2
        # Create a distinct weight matrix for each channel
        W = [randn(T, nout, patch_size^2) for _ in 1:nin]
        return new{T}(img_size, patch_size, nin, nout, num_patches, W)
    end

    function (embed::PatchEmbedLinear{T})(image::Matrix{<:RGB}) where T<:Real
        # Ensure image size matches expected dimensions
        img_size = size(image, 1)
        @assert img_size == embed.img_size "Image size does not match module configuration"

        # Split the RGB image into three separate channel matrices
        channels = channelview(image)  # Shape: (3, 256, 256)
        @assert size(channels, 1) == embed.nin "Number of image channels does not match nin"

        # Extract patches and project for each channel
        projected_channels = []
        for c in 1:embed.nin
            # Extract the channel matrix (2D slice)
            channel_matrix = Matrix{T}(channels[c, :, :])  # Shape: (256, 256)
            # Extract patches
            patches = extract_patches(channel_matrix, embed.patch_size)
            # Flatten patches and project
            patch_matrix = hcat([vec(patch) for patch in patches]...)'  # Shape: (num_patches, patch_size^2)
            projected = patch_matrix * transpose(embed.W[c])  # Shape: (num_patches, nout)
            push!(projected_channels, projected)
        end

```

```
        # Sum the projected embeddings across channels
        projected_patches = reduce(+, projected_channels)  # Shape: (num_patches,
nout)

        return projected_patches
    end
end
```

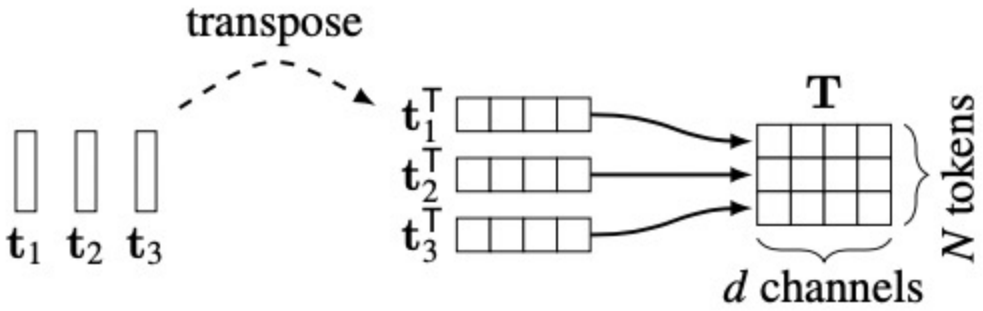
32

embedding dimension




```
Input image shape: (128, 128)
Number of patches: 64
Embedded patches shape: (64, 32)
```

You can embedded think of the embedded patches already as the tokens $\mathbf{T}$ that will be fed into our ViT model. Recall graphically, $\mathbf{T}$ is constructed from $\mathbf{t}_1, \ldots t_N$ like this

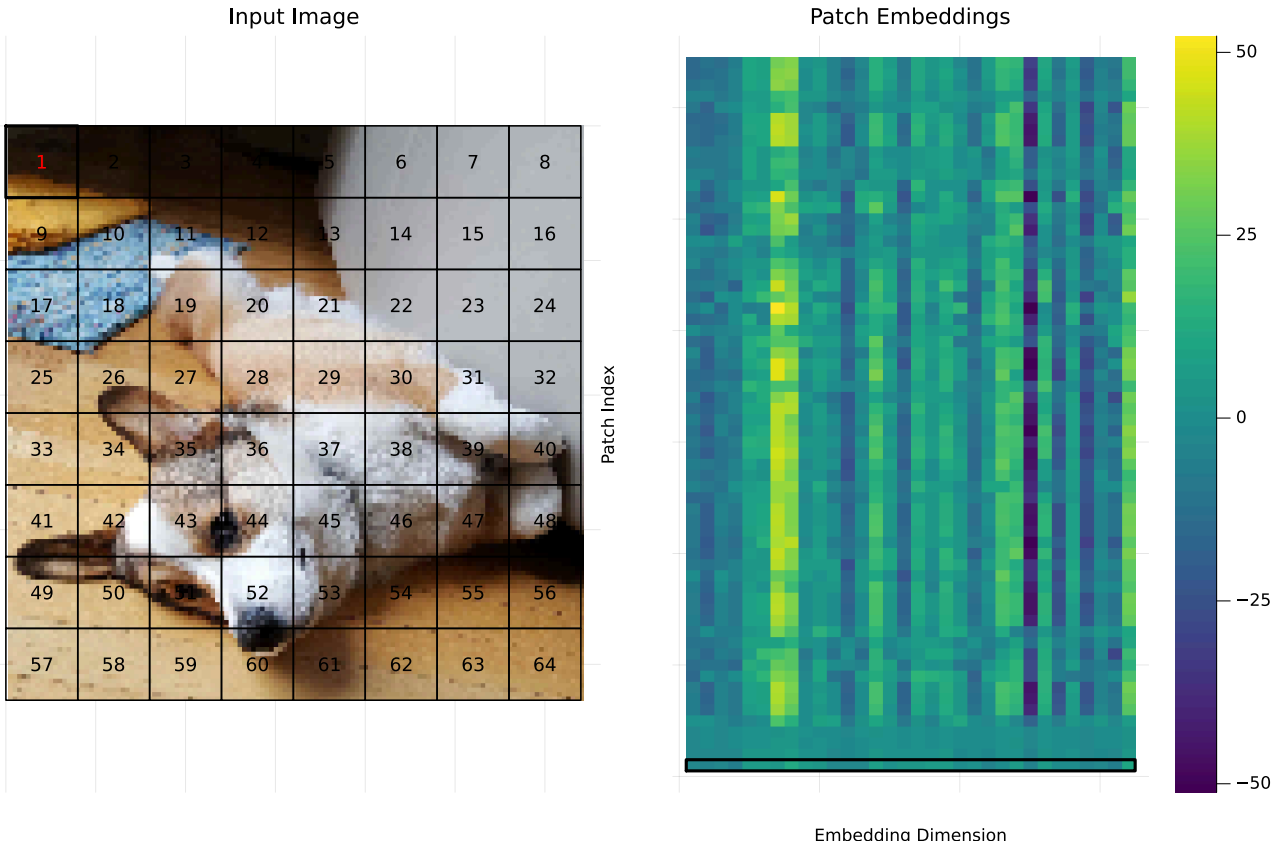


What we do next with **positional encoding** simply adds information about the position patch index, and so doesn't change the shape of the tokens.

```
visualize_patch_embedding (generic function with 1 method)
```

1

emphasize patch



Positional Encoding

One reason why CNNs worked so well for image recognition is because they have an inductive bias for local structure. In a Trasnformer, every token can attend to every other token in the sequence. Because self-attention operation is permutation invariant, it is important to use proper positional encoding to provide order information to the model. The positional encoding $\mathbf{P} \in \mathbb{R}^{L \times d}$ has the same dimension as the input embedding, so it can be added on the input directly.

The vanilla Transformer considered two types of encodings:

- (1) *Sinusoidal positional encoding*: Each dimension of the positional encoding corresponds to a sinusoid of different wavelengths in different dimensions.

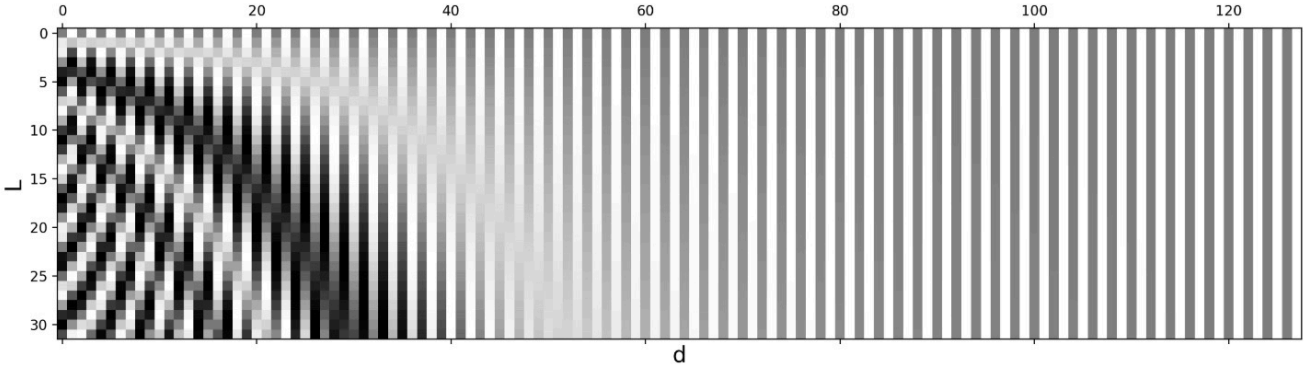
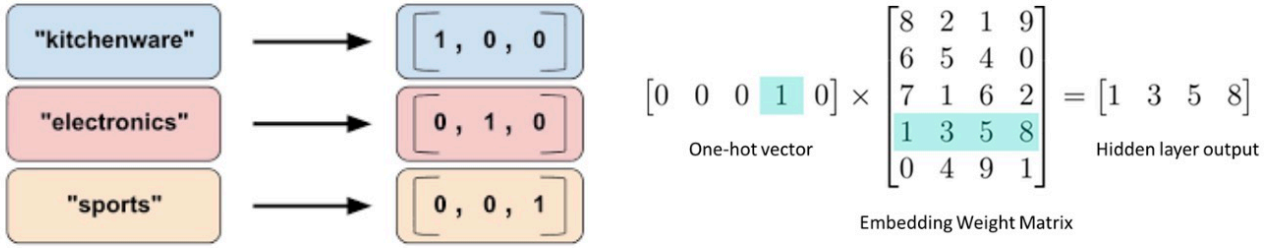


Fig. 3. Sinusoidal positional encoding with $L = 32$ and $d = 128$. The value is between -1 (black) and 1 (white) and the value 0 is in gray.

- (2) *Learned positional encoding*: As its name suggests, assignes each element in a sequence with a learned column vector which encodes its absolute position.

We will use the latter (2) strategy by implementing an `Embedding` module since it is straightforward and becuase embeddig layers are extremely useful and ubiquitous in deep learning code.



1) Convert indices of 3 training examples to one-hot encoding

$$\begin{bmatrix}2\\3\\1\end{bmatrix} \rightarrow \begin{bmatrix}0&0&1&0\\0&0&0&1\\0&1&0&0\end{bmatrix}$$

2) Multiply one-hot encoded inputs with weight matrix

$$\begin{bmatrix}0&0&1&0\\0&0&0&1\\0&1&0&0\end{bmatrix} \begin{bmatrix}0.3374&-0.1778&-0.3035&-0.5880&1.5810\\1.3010&1.2753&-0.2010&-0.1606&-0.4015\\0.6957&-1.8061&-1.1589&0.3255&-0.6315\\-2.8400&-0.7849&-1.4096&-0.4076&0.7953\end{bmatrix}$$

$$= \begin{bmatrix}0.6957&-1.8061&-1.1589&0.3255&-0.6315\\-2.8400&-0.7849&-1.4096&-0.4076&0.7953\\1.3010&1.2753&-0.2010&-0.1606&-0.4015\end{bmatrix}$$

```julia
### 8. Embedding struct
struct Embedding{T<:Real}
    emb::Matrix{T} # Shape: (num_embeddings, embedding_dim)

    function Embedding{T}(num_embeddings::Int, embedding_dim::Int) where T<:Real
        emb = randn(T, num_embeddings, embedding_dim) # Randomly initialize the embedding matrix
        return new{T}(emb)
    end

    function (embedding::Embedding{T})(indices::Vector{Int}) where T<:Real
        # Ensure indices are valid
        @assert all(1 .≤ indices .≤ size(embedding.emb, 1)) "Indices out of range"
        # Perform lookup for each index
        return embedding.emb[indices, :]
    end
end

```

```
Test `Embedding` implementation
Indices: [1, 10, 50, 100]
Embedding shape: (4, 64)
```

Almost there!

We just need to define a few more layers to put together the **ViT**.

```julia
# Sequential struct
struct Sequential{T<:Real}
    # This is an array of modules where each module is applied sequentially to the input.
    seq::AbstractVector

    function Sequential{T}(seq::AbstractVector) where T<:Real
        return new{T}(seq)
    end

    function (sequential::Sequential{T})(x::Array{T}) where T<:Real
        for mod in sequential.seq
            x = mod(x)
        end
        return x
    end
end
```

```
Test `Sequential` implementation
Input shape: (5, 3)
Output shape: (5, 5)
```

```julia
# LayerNorm struct
struct LayerNorm{T<:Real}
    dim::Int

    function LayerNorm{T}(dim::Int) where T<:Real
        return new{T}(dim)
    end

    function (layernorm::LayerNorm{T})(X::Array{T}) where T<:Real
        mean_val = mean(X; dims=layernorm.dim)
        std_val = std(X; dims=layernorm.dim)
        return (X .- mean_val) ./ (std_val .+ eps(T))
    end
end

```

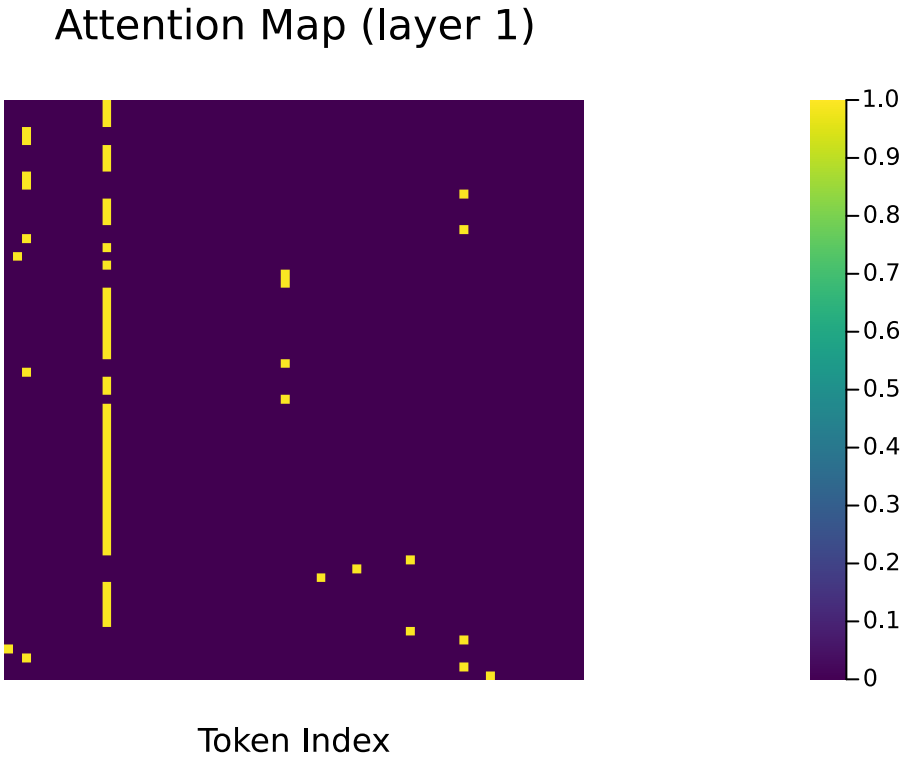
```julia
# Parameter struct
struct Parameter{T<:Real}
    param::Vector{T}

    function Parameter{T}(dim::Int) where T<:Real
        param = randn(T, dim)
        return new{T}(param)
    end
end
```

```julia
# VisionTransformer struct
struct VisionTransformer{T<:Real}
    patch_embed::PatchEmbedLinear{T}          # Patch embedding layer
    pos_E::Embedding{T}                       # Positional encoding
    cls_token::Parameter{T}                   # Learned class embedding token
    transformer::Transformer{T}               # Transformer encoder
    head::Sequential{T}                       # Classification head

    function VisionTransformer{T}(
        n_channels::Int, nout::Int, img_size::Int, patch_size::Int, dim::Int,
        attn_dim::Int, mlp_dim::Int, num_heads::Int, num_layers::Int
    ) where T<:Real
        # Initialize each component
        patch_embed = PatchEmbedLinear{T}(img_size, patch_size, n_channels, dim)
        pos_E = Embedding{T}((img_size ÷ patch_size)^2, dim)
        cls_token = Parameter{T}(dim)
        transformer = Transformer{T}(dim, attn_dim, mlp_dim, num_heads, num_layers)
        head = Sequential{T}([LayerNorm{T}(dim), Linear{T}(dim, nout)])  # LayerNorm
along embedding dim

        return new{T}(patch_embed, pos_E, cls_token, transformer, head)
    end

    function (vt::VisionTransformer{T})(img::Matrix{<:RGB}; return_attn::Bool=false)
where T<:Real
        # Generate patch embeddings
        embs = vt.patch_embed(img)  # Shape: (num_patches, dim)

        # Add positional encoding
        N, D = size(embs)  # Number of patches (N) and embedding dimension (D)
        pos_ids = collect(1:N)  # Generate positional indices
        embs .+= vt.pos_E(pos_ids)  # Add positional encodings to embeddings

        # Add the class token
        cls_token = vt.cls_token.param  # Shape: (dim,)
        x = vcat(cls_token', embs)  # Shape: (N+1, dim)

        # Apply the transformer
        x, alphas = vt.transformer(x; attn_mask=nothing, return_attn=return_attn)

        # Pass through the classification head
        cls_token_out = reshape(x[1, :], 1, :)  # Reshape into a matrix for
'Sequential'
        out = vt.head(cls_token_out)[1, :]  # Final output as a vector

```

```
        return out, alphas
    end
end
```

1

```
attention layer
```



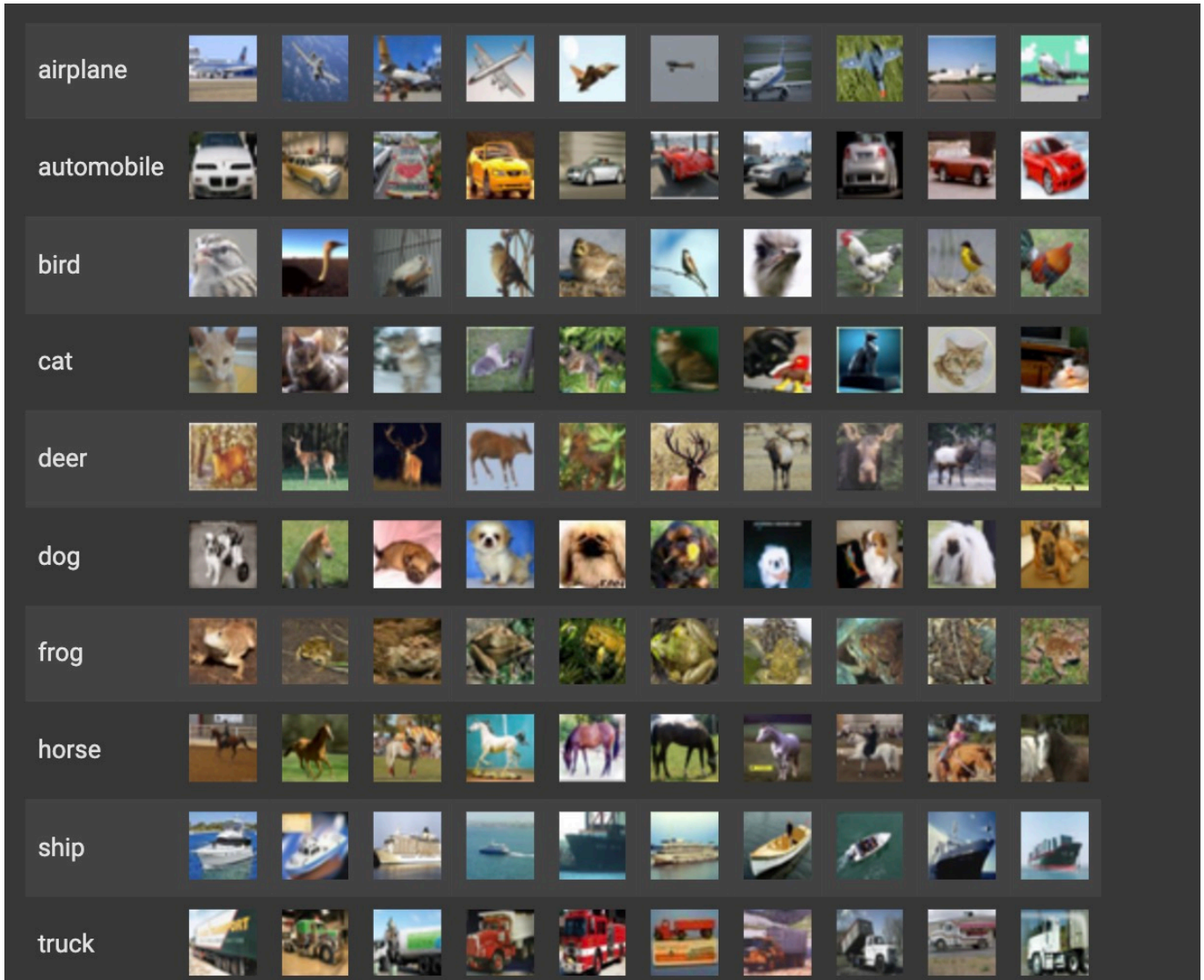

```
Test `VisionTransformer` struct
Output shape: (32,)
Attention weights shape: (4,)
```

Loading an Image Dataset (`CIFAR10`)

CIFAR-10 is an image classification dataset:

- Each data sample is an RGB 32×32 real image. A raw loaded image $\in \mathbb{R}^{3\times32\times32}$.

- Each image is associated with a label $\in \{0, 1, 2, \ldots, 9\}$.



(Table credit to Alex Krizhevsky's [webpage](https://www.cs.toronto.edu/~kriz/cifar.html).)

Our goal is to train a neural network classifier (with a *ViT* as the feature extractor "backbone") that takes such $3 \times 32 \times 32$ images and predicts a label.

1

Example CIFAR image

Cross-Entropy Loss Function

We will define the cross-entropy loss function. For multi-class classification with C classes, the cross-entropy loss for a single sample is given by:

$$\text{Loss} = -\sum_{c=1}^{C} y_c \log\left(\hat{y}_c\right)$$

Here:

- y_c is 1 if the sample belongs to class c, otherwise 0.
- $\hat{y}_c$ is the predicted probability for class c.

```
cross_entropy_loss (generic function with 1 method)
```

```
function cross_entropy_loss(predictions::Matrix{Float64}, targets::Vector{Int})
    # Convert targets to one-hot encoding
    num_samples, num_classes = size(predictions)
    one_hot_targets = zeros(Float64, num_samples, num_classes)
    for i in 1:num_samples
        one_hot_targets[i, targets[i]] = 1.0
    end
    # Compute log of softmax predictions
    log_probs = log.(softmax(predictions, dims=2))
    # Compute the loss
    loss = -sum(one_hot_targets .* log_probs) / num_samples
    return loss
end
```

We didn't get by to training the network 😞

```
compute_gradient (generic function with 1 method)
```

```julia
# Compute gradient for the VisionTransformer
function compute_gradient(vit_model::VisionTransformer{T}, img::Array{T},
target::Int) where T<:Real
    function loss_fn()
        # Forward pass through the model
        output, _ = vit_model(img)
        # Compute the loss (batch size = 1)
        y_pred = reshape(output, 1, :)  # Ensure output is a matrix
        return cross_entropy_loss(y_pred, [target])
    end

    # Compute gradients of the loss function with respect to model parameters
    grad = Enzyme.gradient(loss_fn, vit_model)
    return grad
end
```

Future steps:

- Figure out how to train our ViT model using an autodiff package like `Enzyme.jl`
- Visualize actual real attention maps for that trained model.

Here's an example of what we would expect:

Visualizing Attention Maps

In this project, our Vision Transformer (ViT) model is designed to process the CIFAR-10 dataset. Below is an example attention map visualization for the [CLS] token after training, highlighting which regions of the image the model focuses on during classification.