```julia
using LinearAlgebra, Random, Plots, PlutoUI, Images, Interact#, MLDatasets
```

Our project is to implement a simple Vision Transformer in Julia!

The Transformer architecture, introduced in the paper *Attention Is All You Need* (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) and it quickly replaced convolutional neural networks for image-based tasks.

```julia
md"""
### Our project is to implement a simple Vision Transformer in Julia!

The Transformer architecture, introduced in the paper _Attention Is All You Need_ (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) and it quickly replaced convolutional neural networks for image-based tasks.
"""
```

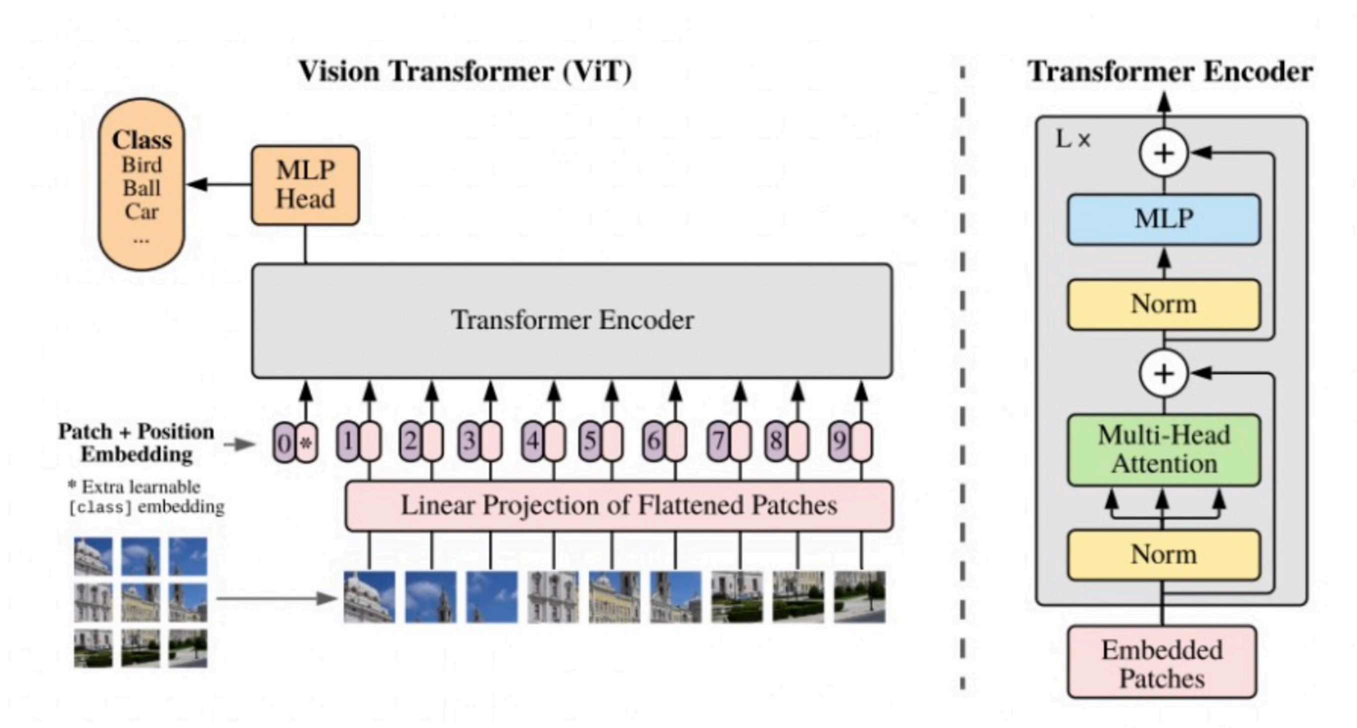


Figure 2: ViT Architecture, Figure from Dosovitskiy et al. [2020]

```julia
md"""
![ViT Model](https://github.com/qsimeon/julia_class_project/blob/e698587c2c2b7455404e6126c06f4ec04c463032/vit_arch.jpg?raw=true)
"""
```

Let's start by defining key components of a Vision Transformer (ViT) model using Julia structs and parametric types, similar to the structure we implemented in Homework 3. We will implement the `AttentionHead`, `MultiHeadedAttention`, and `FeedForwardNetwork` layers as Julia structs. This will set up the parts which get combined together in the `Transformer` model.

```
md"""
Let's start by defining key components of a Vision Transformer (ViT) model using Julia structs and parametric types, similar to the structure we implemented in Homework 3. We will implement the `AttentionHead`, `MultiHeadedAttention`, and `FeedForwardNetwork` layers as Julia structs. This will set up the parts which get combined together in the `Transformer` model.
"""
```

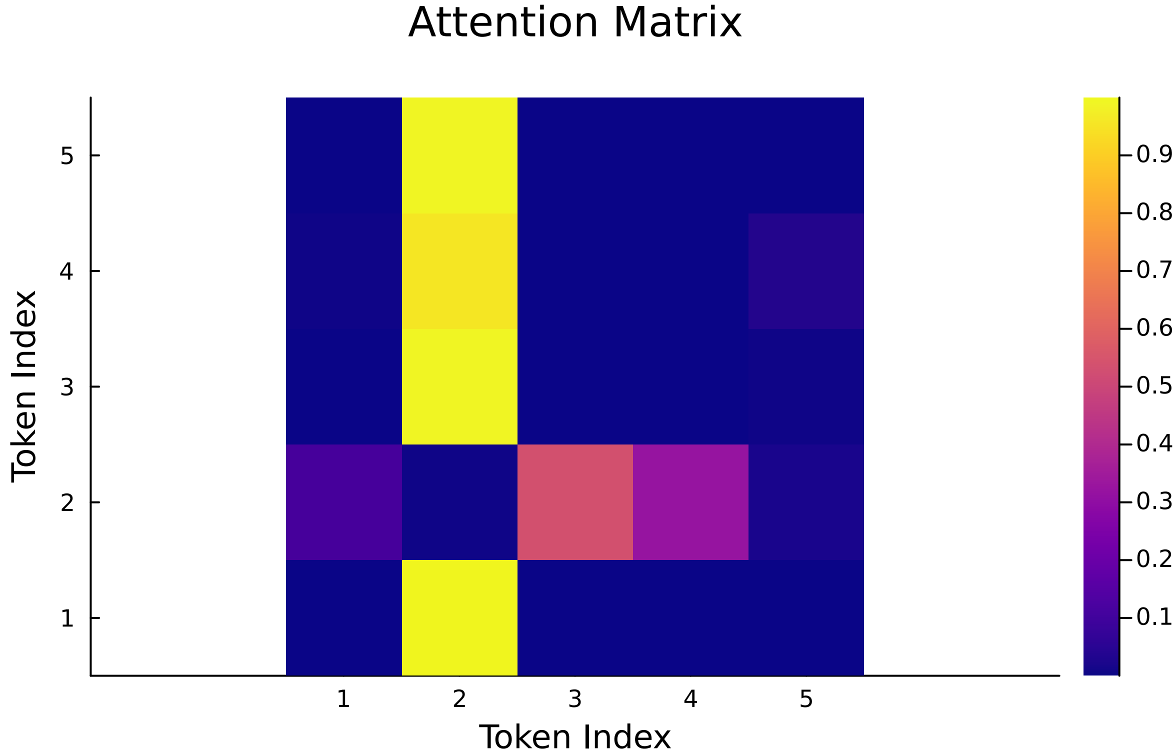
```
softmax (generic function with 1 method)
```

```
# 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::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

```

5

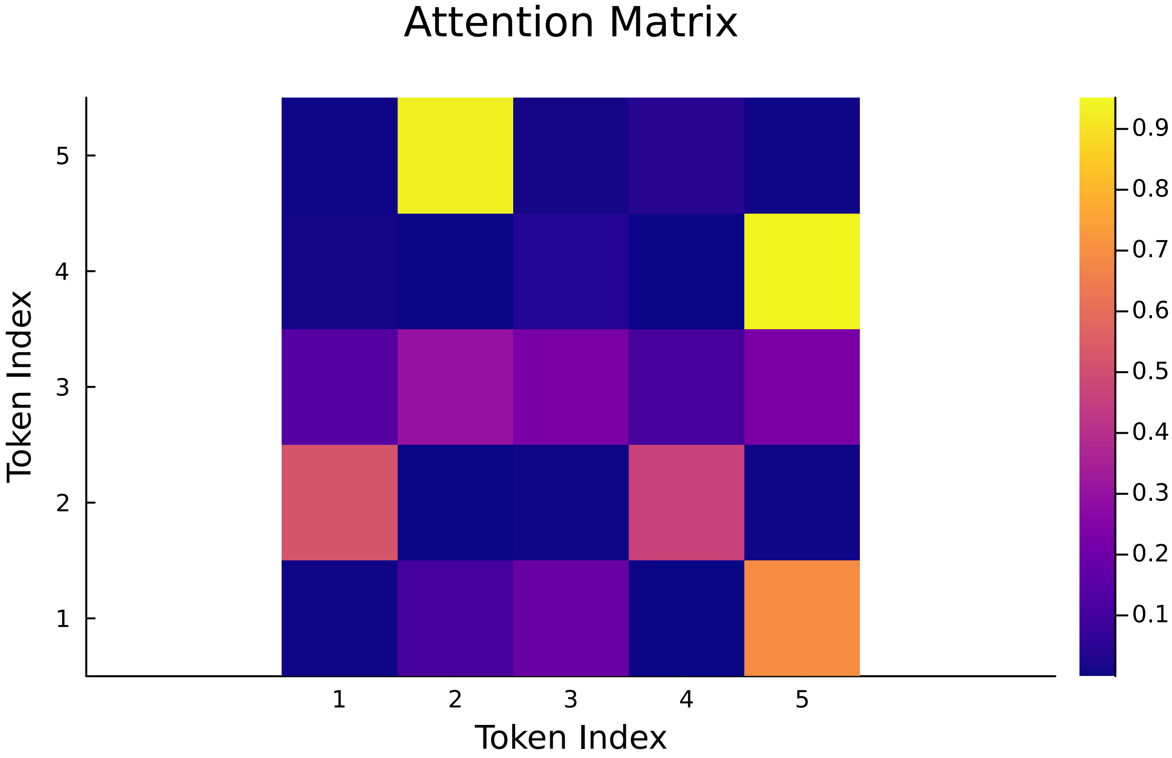
```julia
@bind n_tokens Slider(5:20, show_value=true)
```




```julia
# Test `AttentionHead` implementation
let
    dim, attn_dim = 3, 8
    head = AttentionHead{Float64}(dim, attn_dim)
    X = randn(Float64, n_tokens, dim)  # example 3-D input of n_tokens
    attn_output, alpha = head(X)
    println("attention output shape: ", size(attn_output))
    println("attention weight shape: ", size(alpha))
    heatmap(
        alpha,
        aspect_ratio=:equal,
        xlabel="Token Index",
        ylabel="Token Index",
        title="Attention Matrix",
        c=:plasma,                      # Choose a colormap, e.g., :viridis or :plasma

        clabel="Weight",                 # Label for the color bar
        colorbar=true,                   # Show the color bar
        grid=false,                      # Turn off the grid
        # framestyle=:none,         # Removes the axis lines
        xticks=1:n_tokens,  # Ensures ticks are at each integer index
        yticks=1:n_tokens,
    )

end
```

```
attention output shape: (5, 3)
attention weight shape: (5, 5)
```

```
### 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::Matrix{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
```




```julia
# Test `MultiHeadedAttention` implementation
let
    dim, attn_dim, num_heads = 3, 8, 5
    heads = [AttentionHead{Float64}(dim, attn_dim) for _ in 1:num_heads]
    W_msa = randn(Float64, dim, num_heads * dim)
    X = randn(Float64, n_tokens, dim)  # example 3-D input of n_tokens

    outputs, alphas = [], []
    for head in heads
        attn_out, alpha = head(X)
        push!(outputs, attn_out)
        push!(alphas, alpha)
    end

    concatenated = cat(outputs...; dims=2) # Shape: (N, num_heads*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)
    attn_output = concatenated * transpose(W_msa) # Shape: (N, dim)

    println("attention output shape: ", size(attn_output))
    println("attention weights shape: ", size(attn_alphas))
    # plot the attention mask from the last head
    heatmap(
        attn_alphas[end,:,:],
        aspect_ratio=:equal,
        xlabel="Token Index",
        ylabel="Token Index",
        title="Attention Matrix",
        c=:plasma,                      # Choose a colormap, e.g., :viridis or :plasma
        clabel="Weight",                # Label for the color bar
        colorbar=true,                  # Show the color bar
        grid=false,                     # Turn off the grid
        # framestyle=:none,           # Removes the axis lines
        xticks=1:n_tokens,  # Ensures ticks are at each integer index
        yticks=1:n_tokens,
```

```
    )
end
```

```
attention output shape: (5, 3)
attention weights shape: (5, 5, 5)
```

```
### 3. Feed-Forward Network (FFN)
struct FeedForwardNetwork{T<:Real}
    W1::Matrix{T} # Shape: (n_hidden, dim)
    W2::Matrix{T} # Shape: (dim, n_hidden)
    b1::Vector{T} # Shape: (n_hidden,)
    b2::Vector{T} # Shape: (dim,)

    function FeedForwardNetwork{T}(dim::Int, n_hidden::Int) where T<:Real
        # Our FFN outputs tokens with the same dimension as the input tokens
        return new{T}(randn(T, n_hidden, dim), randn(T, dim, n_hidden), randn(T, n_hidden), randn(T, dim))
    end

    function (ffn::FeedForwardNetwork{T})(X::Matrix{T}) where {T<:Real}
        # X is expected to be an input token matrix with shape (N, dim)
        X = X * transpose(ffn.W1) .+ ffn.b1'  # Shape: (N, n_hidden)
        X = max.(0, X)  # ReLU activation
        return X * transpose(ffn.W2) .+ ffn.b2'  # Shape: (N, dim)
    end
end

```

```
# Test `FeedForwardNetwork` implementation
let
    dim, mlp_dim = 3, 8
    ffn = FeedForwardNetwork{Float64}(dim, mlp_dim)
    X = randn(Float64, n_tokens, dim)  # example 3-D input of n_tokens
    ffn_output = ffn(X)
    println("feedforward output shape: ", size(ffn_output))
end
```

```
feedforward 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, dim)) to an intermediate hidden representation (shape: $(N, \text{n\_hidden})$) 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, dim), maintaining the same dimension as the input tokens.

```
md"""
### Recap so far

1. **`AttentionHead` Implementation**:
    - Projects the input token matrix `X` (shape: $$(N, \text{dim})$$) 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: $$(N, \text{dim})$$) 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: $$(N, \text{dim})$$) to an intermediate hidden representation (shape: $$(N, \text{n\_hidden})$$) 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 $$(N, \text{dim})$$, maintaining the same dimension as the input tokens.
"""

```

```julia
### 4. Attention Residual
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::Matrix{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
### 5. Attention Residual
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

```

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.

```
md"""
### 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
let
    dim, attn_dim, mlp_dim, num_heads = 8, 16, 32, 3
    residual_block = AttentionResidual{Float64}(dim, attn_dim, mlp_dim, num_heads)
    X = randn(Float64, n_tokens, dim)  # example input with n_tokens, each of `dim` dimensions
    output, alphas = residual_block(X)
    println("AttentionResidual output shape: ", size(output))
    println("Attention weights shape (from one layer): ", size(alphas))
end
```

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

```
# Test `Transformer` implementation
let
    dim, attn_dim, mlp_dim, num_heads, num_layers = 8, 16, 32, 3, 6
    transformer = Transformer{Float64}(dim, attn_dim, mlp_dim, num_heads, num_layers)
    X = randn(Float64, n_tokens, dim)  # example input with n_tokens, each of `dim` dimensions
    output, collected_alphas = transformer(X)
    println("Transformer output shape: ", size(output))
    println("Collected attention weights shape: ", size(collected_alphas[1]), " for ", num_layers, " layers")
end
```

```
Transformer output shape: (5, 8)
Collected attention weights shape: (3, 5, 5) for 6 layers
```

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.

We've set up a basic structure of a Transformer using callable structs, parametric types, and matrix operations.

Let's view our Julia callable struct implemenation side-by-side with a bare-bones implementation in PyTorch.

TODO: Side-by-side comparison.

```
md"""
#### 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.

---

We've set up a basic structure of a Transformer using callable structs, parametric types, and matrix operations.

Let's view our Julia callable struct implemenation side-by-side with a bare-bones implementation in PyTorch.

**TODO:** Side-by-side comparison.
"""
```

We want to make a Vision Transformer. This requires some additional layers for image processing: patch embedding and positional encoding.

TODO: Implement `PatchEmbed` in Julia and make a visual example of applying it to some image like this:

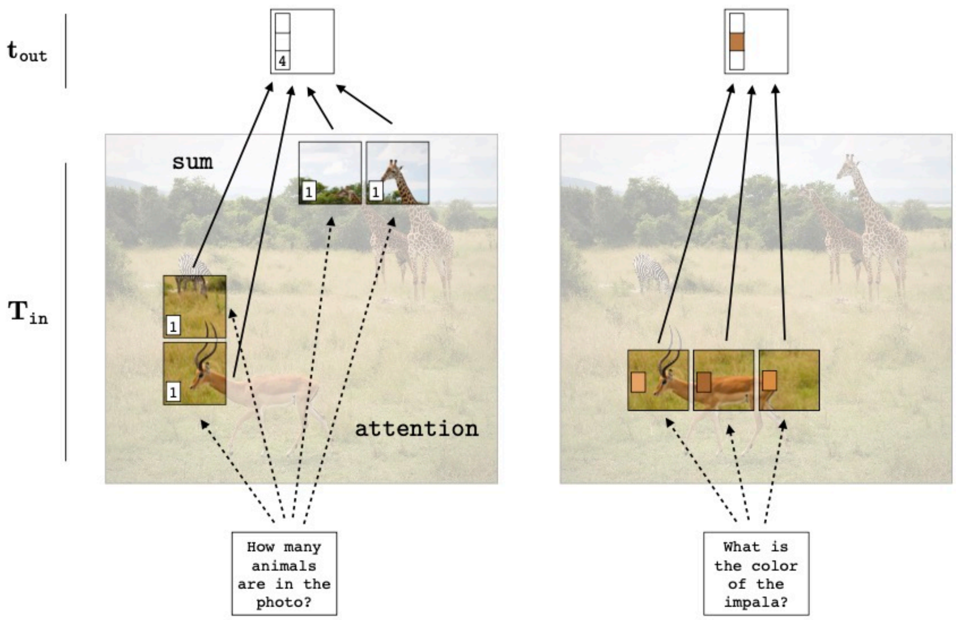


Figure 26.7: How attention can be allocated across different regions (tokens) in an image. The token code vectors consist of multiple dimensions and each can encode a different attribute of the token. To the left we show a dimension that encodes number of animal heads. To the right we show a different dimension that encodes color (or this could be three dimensions, coding RGB). The output token is a weighted sum over all the tokens attended to.

```
md"""
We want to make a Vision Transformer. This requires some additional layers for image processing: patch embedding and positional encoding.

**TODO:** Implement `PatchEmbed` in Julia and make a visual example of applying it to some image like this:

![Patch Image](https://github.com/qsimeon/julia_class_project/blob/main/patch_embed.jpg?raw=true)
"""
```

It turns out the patch embedding is 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. transformations of tokens. What we are doing here is essentially *tokenizing* our image data.

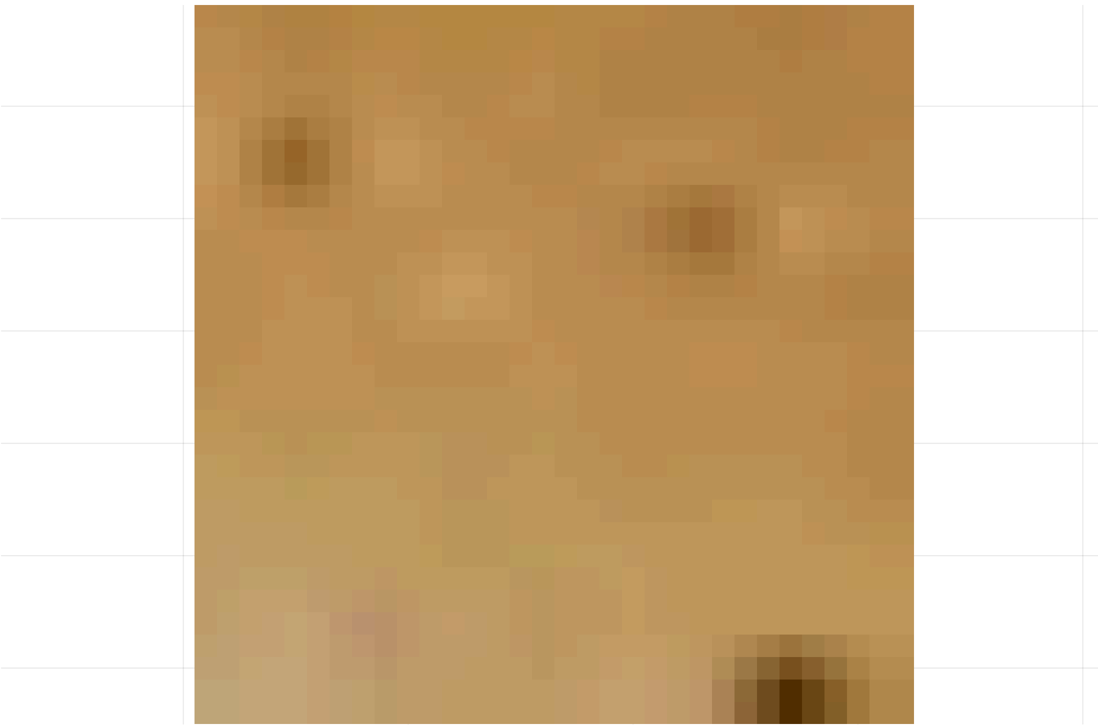
```
md"""
It turns out the patch embedding is can be implemented by applying a strided convolution. However, we will take the more direct and visualizable approach of chopping up an image into patches and linearly projecting the vector that is the flattened patch to the desired dimensionality.

Remember Transformers operate on tokens i.e. transformations of tokens. What we are doing here is essentially *tokenizing* our image data.
"""
```

```julia
### 5. PatchEmbed struct
begin
    struct PatchEmbed{T<:Real}
        img_size::Int
        patch_size::Int
        nin::Int
        nout::Int
        num_patches::Int
        W::Matrix{T} # Linear projection weights

        function PatchEmbed{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
            W = randn(T, nout, patch_size^2 * nin) # Linear projection matrix for each patch
            return new{T}(img_size, patch_size, nin, nout, num_patches, W)
        end
    end
end

```

```julia
# Load and preprocess the image
begin
    image_url = "https://github.com/qsimeon/julia_class_project/blob/e698587c2c2b7455404e6126c06f4ec04c463032/reduced_phil.png?raw=true"
    image_file = download(image_url)
    image = load(image_file)

    # Resize the image to a square (e.g., 256x256)
    img_size = 256
    image_square = imresize(image, (img_size, img_size))
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)
            push!(patches, view(image, i:i+patch_size-1, j:j+patch_size-1))
        end
    end
    return patches
end
```



```
begin
    # Parameters
    patch_size = 32 # Size of each patch (32x32)

    # Extract patches
    patches = extract_patches(image_square, patch_size)

    # Visualize patches in a grid
    n_patches = length(patches)
    grid_dim = Int(sqrt(n_patches)) # Assumes square grid for simplicity

    # Create a grid plot using 'plot(grid=true)'
    plot(layout=(grid_dim, grid_dim), title="Patches of Image", margin=5mm)

    for idx in 1:n_patches
        plot!(heatmap(patches[idx], color=:grays, axis=false, colorbar=false),
    layout=(grid_dim, grid_dim), subplot=idx)
    end

    # Display the grid of patches
    plot!()
end
```