

① PATCH EMBEDDINGS

```
class PatchEmbeddings(nn.Module):
```

```
    """
    Convert the image into patches and then project them into a vector space.
    """
```

```
    def __init__(self, config):
```

```
        super().__init__()
```

```
        self.image_size = config["image_size"]
```

```
        self.patch_size = config["patch_size"]
```

```
        self.num_channels = config["num_channels"]
```

```
        self.hidden_size = config["hidden_size"]
```

```
        # Calculate the number of patches from the image size and patch size
```

```
        self.num_patches = (self.image_size // self.patch_size) ** 2
```

```
        # Create a projection layer to convert the image into patches
```

```
        # The layer projects each patch into a vector of size hidden_size
```

```
        self.projection = nn.Conv2d(self.num_channels, self.hidden_size, kernel_size=self.patch_size, stride=self.patch_size)
```

```
    def forward(self, x):
```

```
        # (batch_size, num_channels, image_size, image_size) -> (batch_size, num_patches, hidden_size)
```

```
        x = self.projection(x)
```

```
        x = x.flatten(2).transpose(1, 2)
```

```
        return x
```

equivalent of a convolution layer
20. Because we project
it to the dimension of out_size.

→ combines the final dimension.

→ Each patch becomes a vector of size P^2C that eventually gets projected to size hidden_size. The linear projection matrix outputs to a dimension 'D'.

$(P^2C \times D)$

$$E = X_P W_{proj}$$

$(N \times P^2C)$

→ shared across every patch.

→ Returns a $N \times D$ embedding.

② Embeddings

```
class Embeddings(nn.Module):
```

```
    """
    Combine the patch embeddings with the class token and position embeddings.
    """
```

the token that attends to all other tokens.

```
def __init__(self, config):
    super().__init__()
    self.config = config
    self.patch_embeddings = PatchEmbeddings(config)
    # Create a learnable [CLS] token
    # Similar to BERT, the [CLS] token is added to the beginning of the input sequence
    # and is used to classify the entire sequence
    self.cls_token = nn.Parameter(torch.randn(1, 1, config["hidden_size"]))
    # Create position embeddings for the [CLS] token and the patch embeddings
    # Add 1 to the sequence length for the [CLS] token
    self.position_embeddings = \
        nn.Parameter(torch.randn(1, self.patch_embeddings.num_patches + 1, config["hidden_size"]))
    self.dropout = nn.Dropout(config["hidden_dropout_prob"])

def forward(self, x):
    x = self.patch_embeddings(x)
    batch_size, _, _ = x.size()
    # Expand the [CLS] token to the batch size
    # (1, 1, hidden_size) -> (batch_size, 1, hidden_size)
    cls_tokens = self.cls_token.expand(batch_size, -1, -1)
    # Concatenate the [CLS] token to the beginning of the input sequence
    # This results in a sequence length of (num_patches + 1)
    x = torch.cat((cls_tokens, x), dim=1)
    x = x + self.position_embeddings
    x = self.dropout(x)
    return x
```

3 Attention Head

```
class AttentionHead(nn.Module):
```

```
    """
    A single attention head.
    This module is used in the MultiHeadAttention module.
    """
```

```
def __init__(self, hidden_size, attention_head_size, dropout, bias=True):
    super().__init__()
    self.hidden_size = hidden_size
    self.attention_head_size = attention_head_size
    # Create the query, key, and value projection layers
    self.query = nn.Linear(hidden_size, attention_head_size, bias=bias)
    self.key = nn.Linear(hidden_size, attention_head_size, bias=bias)
    self.value = nn.Linear(hidden_size, attention_head_size, bias=bias)

    self.dropout = nn.Dropout(dropout)

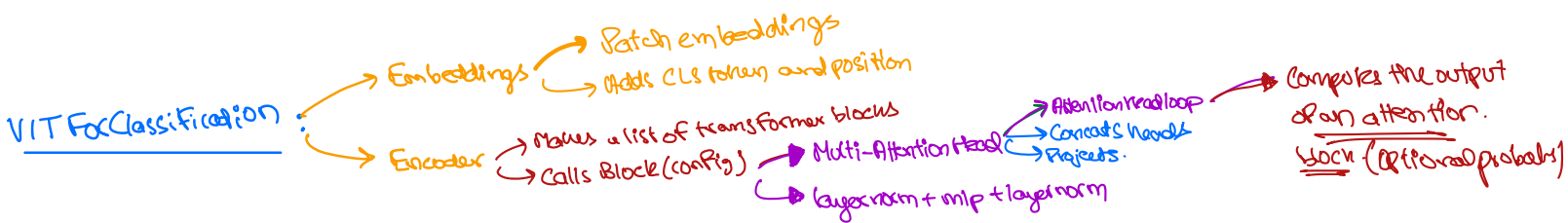
def forward(self, x):
    # Project the input into query, key, and value
    # The same input is used to generate the query, key, and value,
    # so it's usually called self-attention.
    # (batch_size, sequence_length, hidden_size) -> (batch_size, sequence_length, attention_head_size)
    query = self.query(x)
    key = self.key(x)
    value = self.value(x)

    # Calculate the attention scores
    # softmax(Q*K.T/sqrt(head_size))*V
    attention_scores = torch.matmul(query, key.transpose(-1, -2))
    attention_scores = attention_scores / math.sqrt(self.attention_head_size)
    attention_probs = nn.functional.softmax(attention_scores, dim=-1)
    attention_probs = self.dropout(attention_probs)
    # Calculate the attention output
    attention_output = torch.matmul(attention_probs, value)
    return (attention_output, attention_probs)
```

* MLP is standard

* A block class implements one attention block

* The encoder class implements the loops of blocks.



GELU ACTIVATION → $\text{GELU}(x) = x \cdot \phi(x)$