```
class VisionTransformer(nn.Module):
def __init__(self, img_size, patch_size, embed_dim, num_heads, num_layers, num_classes):
super(VisionTransformer, self).__init__()

self.num_patches = (img_size // patch_size) ** 2
self.patch_size = patch_size
self.patch_dim = 3 * patch_size * patch_size # 3 color channels

self.patch_embed = nn.Linear(self.patch_dim, embed_dim)
self.cls_token = nn.Parameter(torch.randn(1, 1, embed_dim))
self.pos_embed = nn.Parameter(torch.randn(1, self.num_patches + 1, embed_dim))

self.transformer = nn.TransformerEncoder(
nn.TransformerEncoderLayer(embed_dim, num_heads, dim_feedforward=embed_dim*4), num_layers
)
self.mlp_head = nn.Linear(embed_dim, num_classes)

def forward(self, x):
B, _, _, _ = x.shape # Get batch size. We don't need the channels, height, or width
x = x.unfold(2, self.patch_size, self.patch_size).unfold(3, self.patch_size, self.patch_size)
x = x.permute(0, 2, 3, 1, 4, 5).contiguous().view(B, self.num_patches, -1)
x = self.patch_embed(x)

cls_tokens = self.cls_token.expand(B, -1, -1)
```

```
class VisionTransformer(nn.Module):
def __init__(self, img_size, patch_size, embed_dim, num_heads, num_layers, num_classes):
super(VisionTransformer, self).__init__()

self.num_patches = (img_size // patch_size) ** 2
self.patch_size = patch_size
self.embed_dim = embed_dim

# Conv2d-based Patch Embedding (Better Stability)
self.patch_embed = nn.Conv2d(3, embed_dim, kernel_size=patch_size, stride=patch_size)

# Class Token & Positional Embedding (Fixed Initialization)
self.cls_token = nn.Parameter(torch.zeros(1, 1, embed_dim))
self.pos_embed = nn.Parameter(torch.zeros(1, self.num_patches + 1, embed_dim))

# Transformer with Pre-Normalization (More Stable Training)
encoder_layer = nn.TransformerEncoderLayer(embed_dim, num_heads, dim_feedforward=embed_dim*4,
dropout=0.1, activation="gelu", batch_first=True)
self.transformer = nn.TransformerEncoder(encoder_layer, num_layers)

# Classification Head (LayerNorm before Output Improves Stability)
self.norm = nn.LayerNorm(embed_dim)
self.mlp_head = nn.Linear(embed_dim, num_classes)

# Initialize Weights
self._init_weights()

# Initialize Weights with Truncated Normal Distribution
def _init_weights(self):
trunc_normal_(self.pos_embed, std=0.02)
trunc_normal_(self.cls_token, std=0.02)
self.apply(self._init_mlp_weights)

# Initialize Weights for MLP Head
def _init_mlp_weights(self, m):
if isinstance(m, nn.Linear):
trunc_normal_(m.weight, std=0.02)
if m.bias is not None:
nn.init.constant_(m.bias, 0)
elif isinstance(m, nn.LayerNorm):
nn.init.constant_(m.bias, 0)
nn.init.constant_(m.weight, 1.0)

def forward(self, x):
B = x.shape[0]

# Patch Embedding (Conv2d)
```

```
x = torch.cat((cls_tokens, x), dim=1)
x += self.pos_embed


x = self.transformer(x)
x = x[:, 0, :]


return self.mlp_head(x)

def build_transformer(config):
return VisionTransformer(
config['img_size'], config['patch_size'], config['embed_dim'], config['num_heads'],
config['num_layers'], config['num_classes']
)
```

```
x = self.patch_embed(x) # [B, embed_dim, H/patch_size, W/patch_size]
x = x.flatten(2).transpose(1, 2) # [B, num_patches, embed_dim]

# Class Token
cls_tokens = self.cls_token.expand(B, -1, -1) # [B, 1, embed_dim]
x = torch.cat((cls_tokens, x), dim=1) # [B, num_patches+1, embed_dim]

# Positional Encoding
x = x + self.pos_embed[:, :x.shape[1], :]

# Transformer Encoder with Pre-Normalization
x = self.transformer(x)

# Classification Head
x = self.norm(x[:, 0, :]) # Use CLS Token
return self.mlp_head(x)

def build_transformer(config):
return VisionTransformer(
config['img_size'], config['patch_size'], config['embed_dim'], config['num_heads'],
config['num_layers'], config['num_classes']
)
```