Great! Let's go through **Attention Mechanisms** and **Transformers** in **great detail**, with **examples, mathematics**, and **coding in PyTorch**.

📘 Class 16: Attention Mechanisms and Transformers

🎯 Learning Objectives

1. Understand the **attention mechanism** (especially self-attention and multi-head attention).
2. Learn how **Transformers** work including:
   - Input embeddings
   - Positional encodings
   - Encoder-decoder architecture
   - Output layer

🔶 PART 1: Attention Mechanism

✳️ What is Attention?

The **attention mechanism** allows a model to **focus** on relevant parts of the input when making decisions.

Suppose you're translating:

"The cat sat on the mat."
You want the model to focus on different words at different steps of translation.

🔢 Mathematical Intuition of Attention

Given:

- Query vector **Q**
- Key vector **K**
- Value vector **V**

The **Scaled Dot-Product Attention** is:

Attention(Q,K,V)=softmax(QKTdk)V\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}} \right)V

Where:

- Q∈Rn×dkQ \in \mathbb{R}^{n \times d_k}
- K∈Rn×dkK \in \mathbb{R}^{n \times d_k}
- V∈Rn×dvV \in \mathbb{R}^{n \times d_v}
- dkd_k: dimension of key vectors (used for scaling)

🧠 Example: Dot Product Attention

Let's say:

Q = [[1, 0]]

K = [[1, 0], [0, 1]]

V = [[10], [20]]

Then:

1. QKT=[1∗1+0∗0,1∗0+0∗1]=[1,0]QK^T = [1*1 + 0*0, 1*0 + 0*1] = [1, 0]
2. softmax([1,0])=[0.731,0.269]\text{softmax}([1, 0]) = [0.731, 0.269]
3. Output: [0.731,0.269]⋅[[10],[20]]=[0.731∗10+0.269∗20]=[12.69][0.731, 0.269] \cdot [[10], [20]] = [0.731*10 + 0.269*20] = [12.69]

🔁 Self-Attention

Each word in the input attends to every other word (including itself) to generate context-aware embeddings.

🧮 Steps:

1. Compute Q, K, V from input embeddings.
2. Use the attention formula.
3. Result: For each word, a new representation capturing global context.

🔀 Multi-Head Attention

Instead of computing attention once, we do it **h times** in **parallel heads** with different projection matrices.

MultiHead(Q,K,V)=Concat(head1,…,headh)WO\text{MultiHead}(Q,K,V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O headi=Attention(QWiQ,KWiK,VWiV)\text{head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V)

Each head learns different aspects of relationships.

🔧 **Coding: Self-Attention and Multi-Head Attention (PyTorch)**

```
import torch

import torch.nn as nn

import torch.nn.functional as F


class ScaledDotProductAttention(nn.Module):

    def __init__(self, d_k):

        super().__init__()

        self.d_k = d_k


    def forward(self, Q, K, V):

        scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(torch.tensor(self.d_k,
dtype=torch.float32))

        attn = F.softmax(scores, dim=-1)

        return torch.matmul(attn, V), attn


class MultiHeadAttention(nn.Module):

    def __init__(self, d_model, num_heads):

        super().__init__()

        assert d_model % num_heads == 0


        self.d_k = d_model // num_heads

        self.num_heads = num_heads


        self.W_q = nn.Linear(d_model, d_model)

        self.W_k = nn.Linear(d_model, d_model)

        self.W_v = nn.Linear(d_model, d_model)
```

```
        self.W_o = nn.Linear(d_model, d_model)
        self.attention = ScaledDotProductAttention(self.d_k)

    def forward(self, Q, K, V):
        batch_size = Q.size(0)
        Q = self.W_q(Q).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        K = self.W_k(K).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)
        V = self.W_v(V).view(batch_size, -1, self.num_heads, self.d_k).transpose(1, 2)

        out, attn = self.attention(Q, K, V)
        out = out.transpose(1, 2).contiguous().view(batch_size, -1, self.num_heads * self.d_k)
        return self.W_o(out)
```

🔷 PART 2: Transformer Architecture

Proposed by Vaswani et al. in *"Attention is All You Need"* (2017).

🔹 Components of Transformer

1. Input Embeddings

- Word tokens are mapped to dense vectors using embeddings.

```
embedding = nn.Embedding(vocab_size, d_model)
```

2. Positional Encodings

Since there’s no recurrence, we inject position information:

$$PE_{(pos, 2i)} = \sin(pos / 10000^{2i / d_{\text{model}}})$$
$$PE_{(pos, 2i+1)} = \cos(pos / 10000^{2i / d_{\text{model}}})$$

```
class PositionalEncoding(nn.Module):
    def __init__(self, d_model, max_len=5000):
        super().__init__()
        pe = torch.zeros(max_len, d_model)
        pos = torch.arange(0, max_len).unsqueeze(1)
```

```
        div_term = torch.exp(torch.arange(0, d_model, 2) * -(torch.log(torch.tensor(10000.0)) / d_model))
        pe[:, 0::2] = torch.sin(pos * div_term)
        pe[:, 1::2] = torch.cos(pos * div_term)
        self.pe = pe.unsqueeze(0)

    def forward(self, x):
        return x + self.pe[:, :x.size(1)]
```

🔸 Encoder Layer

Each encoder block has:

- Multi-head self-attention
- Add & Norm
- Feed-forward layer
- Add & Norm

🔸 Decoder Layer

Each decoder block has:

- Masked multi-head self-attention (prevent attending to future tokens)
- Multi-head attention (encoder-decoder)
- Feed-forward
- LayerNorms

🔄 Transformer Flow

```
Input --> Embedding + Positional Encoding --> N Encoder Blocks --> Memory

Target --> Embedding + Positional Encoding --> N Decoder Blocks (with Masked Attention)
--> Final Linear + Softmax --> Output
```

🧱 Final Output Layer

Projects decoder output to vocabulary space:

output = nn.Linear(d_model, vocab_size)

🧪 Full Mini Transformer (Conceptual Code)

You can build a full transformer using:

- MultiHeadAttention
- PositionalEncoding
- FeedForward (two linear layers)
- LayerNorm

Let me know if you'd like a **complete Transformer architecture implemented in code** (like a small language translator).

Would you like me to give you a complete working example of a Transformer in PyTorch (e.g., for toy language translation)?