Transformer e GPT: Dalla teoria all'implementazione 🚀

Obiettivi della presentazione:

- Capire i concetti chiave dei Transformer
- Analizzare i componenti fondamentali dell'architettura
- Approfondire l'architettura GPT e la sua implementazione
- Vedere esempi pratici di codice e ottimizzazioni
- Confrontare BERT e GPT e discutere applicazioni

> Chi ha già usato modelli come ChatGPT o BERT? In che contesti?

Perché i Transformer hanno rivoluzionato l'NLP?

- Elaborazione **parallela** delle sequenze (non più sequenziale come le RNN)
- **Self-attention**: relazioni dirette tra tutte le parole della sequenza
- Encoding posizionale esplicito per mantenere l'ordine
- **Scalabilità**: modelli sempre più grandi e potenti

> Domanda: Quali problemi delle RNN vengono superati dai Transformer?

Quiz: Cosa rende speciale i Transformer? 🧩

A) Usano solo reti convoluzionali
B) Processano sequenze elemento per elemento
C) Si basano sul meccanismo di self-attention
D) Richiedono meno dati per l'addestramento

> Qual è la risposta corretta? Perché?

Struttura generale del Transformer

- **Encoder**: elabora il contesto dell'input (usato in BERT)
- **Decoder**: genera una sequenza in modo autoregressivo (usato in GPT)
- **Self-attention**: permette interazione tra tutte le posizioni
- **Feed-Forward**: trasforma le rappresentazioni posizione per posizione
- **LayerNorm & Residual**: stabilità e profondità

> Secondo voi, qual è il componente più determinante per il successo dei Transformer?

Tabella dei componenti chiave

Componente	Descrizione	Scopo principale
Multi-Head Attention	Più "canali" di attenzione	Cattura molteplici relazioni tra posizioni
Feed-Forward Network	Strati densi per posizione	Aumenta la complessità e la capacità del modello
Positional Encoding	Informazione di posizione	Permette di distinguere l'ordine dei token
Layer Normalization	Normalizza ogni sotto-strato	Stabilizza e accelera l'addestramento
Residual Connection	Collegamenti diretti tra strati	Facilita il flusso del gradiente nelle reti profonde
Dropout	Disattiva connessioni in modo casuale	Riduce l'overfitting

Embedding: rappresentare i token come vettori

Ogni parola/token viene trasformato in un vettore continuo (embedding) che il modello può elaborare.

Il cuore del Transformer: Self-Attention ❤️

Query, Key, Value per ogni token:

- **Query (Q):** Cosa sta cercando il token?
- **Key (K):** Cosa offre agli altri token?
- **Value (V):** Qual è il contenuto da trasmettere?

> Come aiuta questo meccanismo a cogliere relazioni tra parole anche lontane nella frase?

Calcolo dell'attenzione: i 4 step chiave

1. **Scores = $Q \times K^t$**
2. **Scaling:** Scores / $\sqrt{d_k}$
3. **Softmax:** pesi di attenzione
4. **Output:** somma pesata dei Value

> Perché dividiamo per $\sqrt{d_k}$? Cosa accadrebbe senza scaling?

Multi-Head Attention:
Più "teste" di attenzione in parallelo → ogni testa può imparare relazioni diverse (es. sintattiche, semantiche, a lungo termine).

Positional Encoding: come dare il senso dell’ordine 📍

- Il Transformer non "vede" l’ordine dei token: serve un encoding posizionale.
- Si usano funzioni sinusoidali di diverse frequenze per ogni posizione e dimensione.

> Perché è utile usare funzioni periodiche invece di numeri sequenziali?

Feed-Forward e Layer Normalization

```
# Feed-Forward Network
FFN(x) = max(0, x * W_1 + b_1) * W_2 + b_2

# Layer Normalization
LayerNorm(x) = γ * (x − μ) / √(σ² + ε) + β
```

- **Feed-Forward:** trasforma ogni posizione in modo indipendente, aggiunge non-linearità.
- **LayerNorm:** normalizza ogni esempio → stabilità anche con sequenze di lunghezza variabile.
- **Residual:** somma input e output di ogni blocco.

> Perché LayerNorm è preferita a BatchNorm nei Transformer?

Decoder Transformer: la base di GPT 🧠

- **Masked self-attention:** ogni token vede solo i precedenti
- Stack di blocchi identici: attention, feed-forward, LayerNorm, residual
- Output: layer lineare + softmax (predizione della parola successiva)

> Perché è fondamentale mascherare le posizioni future nel decoder?

Implementazione GPT step-by-step 💻

Cosa vedremo:

- Analisi dei componenti chiave in codice PyTorch
- Implementazione di un mini-GPT da zero
- Esempio pratico di forward, loss e generazione

> Chi ha già usato PyTorch o implementato modelli deep learning da zero?

Setup e configurazione: la classe di base

```
import math
import torch
import torch.nn as nn
import torch.nn.functional as F

class GPTConfig:
    def __init__(self, vocab_size, block_size, n_embd=768,
                 n_layer=12, n_head=12, dropout=0.1):
        self.vocab_size = vocab_size  # dimensione del vocabolario
        self.block_size = block_size  # lunghezza massima della sequenza
        self.n_embd = n_embd          # dimensione degli embedding
        self.n_layer = n_layer        # numero di blocchi del decoder
        self.n_head = n_head          # numero di teste di attenzione
        self.dropout = dropout        # probabilità di dropout
```

Self-Attention: implementazione in PyTorch

```
class SelfAttention(nn.Module):
    def __init__(self, config):
        super().__init__()
        assert config.n_embd % config.n_head == 0
        # key, query, value projections
        self.key = nn.Linear(config.n_embd, config.n_embd)
        self.query = nn.Linear(config.n_embd, config.n_embd)
        self.value = nn.Linear(config.n_embd, config.n_embd)
        # output projection
        self.proj = nn.Linear(config.n_embd, config.n_embd)
        # regularization
        self.attn_dropout = nn.Dropout(config.dropout)
        self.resid_dropout = nn.Dropout(config.dropout)
        # causal mask
        self.register_buffer("mask", torch.tril(torch.ones(config.block_size,
                                     config.block_size)).view(1, 1, config.block_size, config.block_size))
        self.n_head = config.n_head
        self.n_embd = config.n_embd
```

Self-Attention: forward pass

```
def forward(self, x):
    B, T, C = x.size() # batch size, sequence length, embedding dimensionality

    # calculate query, key, values for all heads in batch
    k = self.key(x).view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
    q = self.query(x).view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)
    v = self.value(x).view(B, T, self.n_head, C // self.n_head).transpose(1, 2) # (B, nh, T, hs)

    # causal self-attention: (B, nh, T, hs) x (B, nh, hs, T) -> (B, nh, T, T)
    att = (q @ k.transpose(-2, -1)) * (1.0 / math.sqrt(k.size(-1)))
    att = att.masked_fill(self.mask[:,:,:T,:T] == 0, float('-inf'))
    att = F.softmax(att, dim=-1)
    att = self.attn_dropout(att)
    y = att @ v # (B, nh, T, T) x (B, nh, T, hs) -> (B, nh, T, hs)
    y = y.transpose(1, 2).contiguous().view(B, T, C) # re-assemble all head outputs side by side

    # output projection
    y = self.resid_dropout(self.proj(y))
    return y
```

Feed-Forward Network: codice essenziale

```
class FeedForward(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(config.n_embd, 4 * config.n_embd),
            nn.GELU(),  # Utilizziamo GELU invece di ReLU come in GPT
            nn.Linear(4 * config.n_embd, config.n_embd),
            nn.Dropout(config.dropout),
        )

    def forward(self, x):
        return self.net(x)
```

Un blocco Transformer: combinare attention e feed-forward

```
class Block(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.ln1 = nn.LayerNorm(config.n_embd)
        self.attn = SelfAttention(config)
        self.ln2 = nn.LayerNorm(config.n_embd)
        self.ffn = FeedForward(config)

    def forward(self, x):
        # Utilizziamo la normalizzazione pre-layer come in GPT-2
        x = x + self.attn(self.ln1(x))
        x = x + self.ffn(self.ln2(x))
        return x
```

Modello GPT completo: struttura principale

```
class GPT(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.config = config

        # input embedding
        self.tok_emb = nn.Embedding(config.vocab_size, config.n_embd)
        self.pos_emb = nn.Parameter(torch.zeros(1, config.block_size, config.n_embd))
        self.drop = nn.Dropout(config.dropout)

        # transformer blocks
        self.blocks = nn.Sequential(*[Block(config) for _ in range(config.n_layer)])

        # final layer norm
        self.ln_f = nn.LayerNorm(config.n_embd)

        # language modeling head
        self.head = nn.Linear(config.n_embd, config.vocab_size, bias=False)

        # initialize weights
        self.apply(self._init_weights)
```

Modello GPT completo: pesi e forward pass

```
def _init_weights(self, module):
    if isinstance(module, (nn.Linear, nn.Embedding)):
        module.weight.data.normal_(mean=0.0, std=0.02)
        if isinstance(module, nn.Linear) and module.bias is not None:
            module.bias.data.zero_()
    elif isinstance(module, nn.LayerNorm):
        module.bias.data.zero_()
        module.weight.data.fill_(1.0)

def forward(self, idx, targets=None):
    B, T = idx.size()
    assert T <= self.config.block_size, f"Cannot forward sequence of length {T}, block size is only {self.config.block_size}"

    # forward the GPT model
    token_embeddings = self.tok_emb(idx) # (B, T, n_embd)
    position_embeddings = self.pos_emb[:, :T, :] # (1, T, n_embd)
    x = self.drop(token_embeddings + position_embeddings)
    x = self.blocks(x)
    x = self.ln_f(x)
    logits = self.head(x) # (B, T, vocab_size)

    # if we are given some desired targets also calculate the loss
    loss = None
    if targets is not None:
        loss = F.cross_entropy(logits.view(-1, logits.size(-1)), targets.view(-1))

    return logits, loss
```

Generazione di testo autoregressiva

```python
def generate(self, idx, max_new_tokens, temperature=1.0, top_k=None):
    """
    Generate new tokens beyond the context provided in idx.

    Args:
        idx: (B, T) tensor of indices in the current context
        max_new_tokens: number of new tokens to generate
        temperature: softmax temperature, higher values increase diversity
        top_k: if set, only sample from the top k most probable tokens

    Returns:
        (B, T+max_new_tokens) tensor of indices
    """
    for _ in range(max_new_tokens):
        # crop context if needed
        idx_cond = idx if idx.size(1) <= self.config.block_size else idx[:, -self.config.block_size:]
        # forward pass
        logits, _ = self.forward(idx_cond)
        # focus on the last time step
        logits = logits[:, -1, :] / temperature
        # optionally crop probabilities to only the top k options
        if top_k is not None:
            v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
            logits[logits < v[:, [-1]]] = -float('Inf')
        # apply softmax to convert to probabilities
        probs = F.softmax(logits, dim=-1)
        # sample from the distribution
        idx_next = torch.multinomial(probs, num_samples=1)
        # append sampled index to the running sequence
        idx = torch.cat((idx, idx_next), dim=1)

    return idx
```

Esempio pratico: come usare il modello GPT

```
# Esempio di configurazione per un modello piccolo
config = GPTConfig(
    vocab_size=50257,  # Dimensione del vocabolario GPT-2
    block_size=1024,   # Lunghezza massima della sequenza
    n_embd=768,        # Dimensione degli embedding
    n_layer=12,        # Numero di blocchi
    n_head=12,         # Numero di teste di attenzione
    dropout=0.1        # Dropout
)

# Inizializzazione del modello
model = GPT(config)

# Esempio di input (batch di 2 sequenze di 10 token ciascuna)
x = torch.randint(0, config.vocab_size, (2, 10))
targets = torch.randint(0, config.vocab_size, (2, 10))

# Forward pass
logits, loss = model(x, targets)
print(f"Input shape: {x.shape}")
print(f"Logits shape: {logits.shape}")
print(f"Loss: {loss.item()}")

# Generazione di testo
context = torch.tensor([[464, 1842, 2181, 1447, 373]])  # Esempio di contesto iniziale
generated = model.generate(context, max_new_tokens=20, temperature=0.8, top_k=40)
print(f"Generated sequence shape: {generated.shape}")
```

Addestrare il modello: ciclo base

```python
def train(model, data_loader, optimizer, epochs, device):
    model.train()
    for epoch in range(epochs):
        total_loss = 0
        for batch_idx, (x, y) in enumerate(data_loader):
            x, y = x.to(device), y.to(device)

            # Forward pass
            optimizer.zero_grad()
            logits, loss = model(x, y)

            # Backward pass
            loss.backward()
            optimizer.step()

            total_loss += loss.item()

            if batch_idx % 100 == 0:
                print(f"Epoch {epoch+1}/{epochs}, Batch {batch_idx}, Loss: {loss.item():.4f}")

        avg_loss = total_loss / len(data_loader)
        print(f"Epoch {epoch+1}/{epochs}, Average Loss: {avg_loss:.4f}")
```

Ottimizzazioni pratiche e best practices

1. **Efficienza computazionale**:
    - Mixed precision training
    - Gradient accumulation
    - Model parallelism
2. **Tokenizzazione**: Byte-Pair Encoding (BPE)
3. **Strategie di addestramento**:
    - Learning rate scheduling
    - Weight decay
    - Gradient clipping
4. **Generazione di testo**:
    - Beam search
    - Nucleus sampling

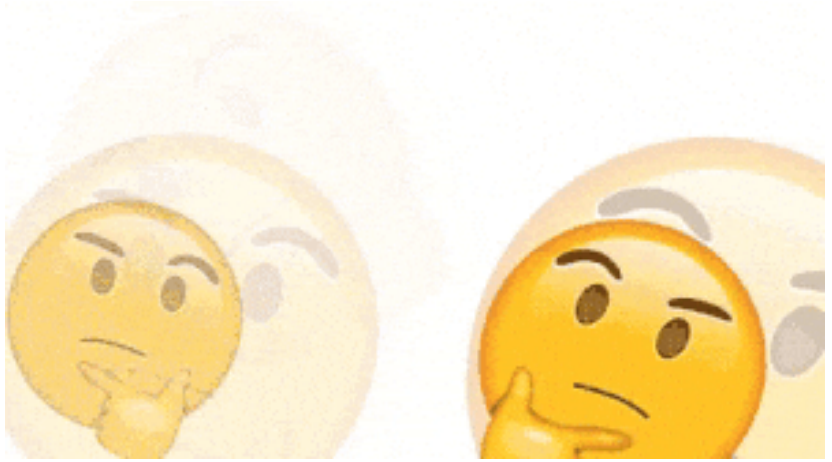
BERT vs GPT: differenze chiave

Caratteristica	BERT	GPT
Architettura	Encoder	Decoder
Direzionalità	Bidirezionale	Unidirezionale (left-to-right)
Pre-addestramento	Masked Language Modeling (MLM)	Predizione della parola successiva
Applicazioni tipiche	Comprensione (classificazione, NER, QA)	Generazione (completamento, traduzione, riassunto)

Domande & Discussione finale 🤔

Hai dubbi sull'architettura o l'implementazione?

- Differenze con LSTM/CNN?
- Limiti attuali dei Transformer?
- Come adattare GPT a un task specifico?
- Implicazioni etiche dei LLM?
- Come interpretare cosa "vede" un Transformer?



Conclusioni e takeaway 🎯

- Il Transformer ha rivoluzionato l'NLP grazie a self-attention e parallelizzazione
- GPT sfrutta il decoder per la generazione di testo
- L'implementazione da zero aiuta a capire ogni componente
- Questi modelli sono la base di ChatGPT, LLM e molte applicazioni attuali

Riferimenti principali:

- Vaswani et al. (2017), "Attention is All You Need"
- Radford et al. (2018, 2019), "Generative Pre-Training"

> Come immagini il futuro dei modelli linguistici? Quali nuove applicazioni potrebbero nascere?

Una domanda per il futuro... 🤔

Se un modello GPT potesse scrivere il suo stesso codice, come si implementerebbe?

> Riflettiamo: l'IA può contribuire al proprio sviluppo?

Risorse per approfondire 📚

- The Illustrated Transformer (Jay Alammar)
- The Annotated Transformer (Harvard NLP)
- minGPT (Andrej Karpathy)
- Hugging Face Transformers
- Attention Is All You Need (paper)

> Consiglio: partite da "The Illustrated Transformer" per una spiegazione visiva!

Quiz e domande interattive 🎲

1. **Cosa distingue i Transformer dalle RNN?**
2. **Perché serve il positional encoding?**
3. **Qual è il ruolo della maschera nel decoder di GPT?**
4. **In quali task useresti BERT invece di GPT?**
5. **Come influenzano temperature e top_k la generazione di testo?**

> Discutiamone insieme!