Architettura Transformer e Implementazione di GPT from Scratch 🚀

Perché i Transformer hanno rivoluzionato l'NLP? 🤔

- **Parallelizzazione completa** vs elaborazione sequenziale delle RNN
- **Self-attention**: cattura dipendenze a lungo termine direttamente
- **Rappresentazioni posizionali** esplicite
- **Architettura scalabile** per modelli sempre più grandi

Quiz: Cosa rende speciale l'architettura Transformer? 🧩

A) Utilizza solo reti convoluzionali

B) Processa sequenze elemento per elemento

C) Si basa sul meccanismo di self-attention

D) Richiede meno dati per l'addestramento

Componenti principali dell'architettura Transformer 🧱

- **Encoder**: comprende il contesto di input
- **Decoder**: genera output sequenziale
- **Self-Attention**: il cuore del modello
- **Feed-Forward Networks**: elaborazione per posizione
- **Layer Normalization e Residual Connections**: stabilità

Componenti principali dell'architettura Transformer 🧱

Componente	Descrizione	Scopo
Attenzione Multi-Head	Meccanismo per concentrarsi su diverse parti dell'input	Cattura le dipendenze tra diverse posizioni nella sequenza
Reti Feed-Forward	Strati completamente connessi per posizione	Trasforma gli output dell'attenzione, aggiungendo complessità
Codifica Posizionale	Aggiunge informazioni posizionali agli embedding	Fornisce il contesto dell'ordine della sequenza al modello
Normalizzazione Layer	Normalizza gli input di ogni sotto-strato	Stabilizza l'addestramento, migliora la convergenza
Connessioni Residue	Scorciatoie tra gli strati	Aiuta nell'addestramento di reti più profonde riducendo i problemi di gradiente
Dropout	Azzera casualmente alcune connessioni della rete	Previene l'overfitting regolarizzando il modello

Embedding

Self-Attention: Il cuore del Transformer ❤️

Query, Key e Value

Per ogni elemento della sequenza:

- **Query (Q)**: cosa l'elemento "sta cercando"
- **Key (K)**: cosa l'elemento "offre" agli altri
- **Value (V)**: contenuto informativo dell'elemento

Calcolo dell'attenzione in 4 passaggi 🧮

1. **Calcolo dei punteggi**: `Scores = Q * K^T`
2. **Scaling**: `Scores_scaled = Scores / √d_k`
3. **Softmax**: `Weights = softmax(Scores_scaled)`
4. **Aggregazione pesata**: `Output = Weights * V`

Positional Encoding: Aggiungere informazioni sulla posizione 📍

Feed-Forward Networks 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
```

Architettura del Decoder (utilizzata in GPT)

- **Masked Self-Attention**: vede solo le posizioni precedenti
- **Stack di blocchi identici**:
  - Masked multi-head self-attention
  - Feed-forward network
  - Layer normalization e residual connections
- **Linear Layer e Softmax finale**

Implementazione di un modello GPT from scratch 💻

Obiettivi:

- Comprendere ogni componente dell'architettura
- Implementare un modello GPT semplificato
- Utilizzare PyTorch per l'implementazione

Setup e configurazione del modello 🛠️

```
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
```

Implementazione del Self-Attention 👁️

```
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
```

Implementazione del forward pass del Self-Attention 🔄

```
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
```

Implementazione del Feed-Forward Network 🔄

```
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)
```

Implementazione di un blocco del Transformer 🧱

```
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
```

Implementazione del modello GPT completo (Parte 1) 🏗️

```python
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)
```

Implementazione del modello GPT completo (Parte 2) 🏗️

```
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
```

Implementazione della generazione di testo 📝

```
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 di utilizzo del modello 🚀

```
# 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}")
```

Addestramento del modello 🏋️

```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}")
```

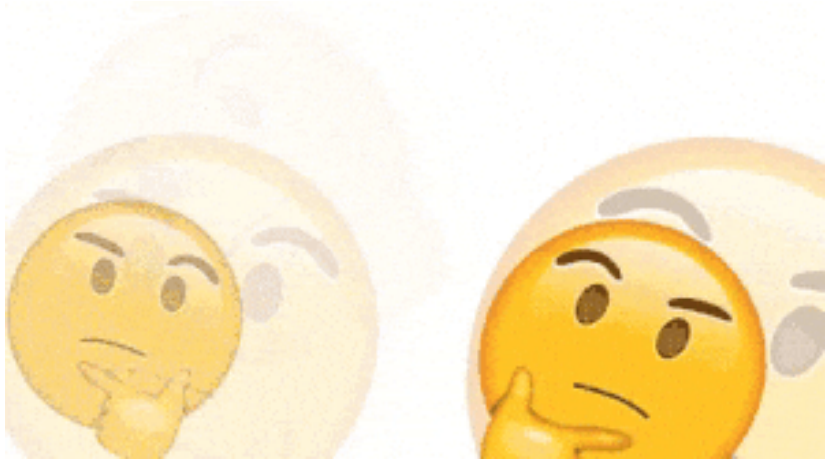
Ottimizzazioni e considerazioni pratiche 🔧

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

Confronto tra BERT e GPT 🔍

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 🤔



Conclusione 🎯

- L'architettura Transformer ha rivoluzionato l'NLP grazie al meccanismo di self-attention
- I modelli GPT utilizzano la parte decoder del Transformer per generazione di testo
- L'implementazione from scratch ci ha permesso di comprendere ogni componente
- Questi modelli sono alla base di sistemi come ChatGPT e altri LLM

Riferimenti

- Vaswani et al. (2017). "Attention is All You Need"
- Radford et al. (2018). "Improving Language Understanding by Generative Pre-Training"
- Radford et al. (2019). "Language Models are Unsupervised Multitask Learners"

Hai mai pensato... 🤔

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

Risorse per approfondire 📚

- The Illustrated Transformer - Jay Alammar
- The Annotated Transformer - Harvard NLP
- minGPT - Andrej Karpathy
- Hugging Face Transformers - Documentazione
- Attention Is All You Need - Paper originale