MACHINE LEARNING IN PYTHON

(PART 5)

LLMS: TRANSFORMER MODELS IN PYTORCH

LUKE SHENEMAN
MACHINE LEARNING IN PYTHON SERIES

Part 1: The Basics

Part 2: Generative Adversarial Networks (GANs) - PyTorch

Part 3: Semantic Segmentation in TensorFlow

Part 4: Diffusion Models In PyTorch
Generative AI: Learn a \textit{latent} representation of the distribution of our complex training data and then sample from it.

Deep Learning

Diffusion, etc.

Transformers, etc.

Training Data

Deep Learning

ChatGPT

Training data
(e.g. 64x64x3≈12K dims)

Sampling

Learning

\( p(x) \)

\( \hat{p}(x) \)

The cat sat on the 
\textit{table}

Input

\textit{chair}

\textit{mat}

Prediction
We are going to build a Shakespeare GPT using a transformer model.

It will learn from scratch from the complete works of Shakespeare.

It will emit an eternal stream of Shakespeare
REVIEW

- Recap from Parts 1-4
  - Machine Learning Basics
  - Neural Networks
- Tensors
- GPUs and CUDA
- PyTorch
REVIEW OF BASICS

• Machine learning is a data-driven method for creating models for prediction, optimization, classification, generation, and more

• Python and scikit-learn, TensorFlow, PyTorch

• MNIST

• Artificial Neural Networks (ANNs)
**NEURAL NETWORK BASICS**

```
from sklearn.neural_network import MLPClassifier

# Instantiate a Neural Network and train our model
nn = MLPClassifier(hidden_layer_sizes=(50, 25),
                    max_iter=50,
                    n_iter_no_change=50,
                    activation='relu',
                    solver='adam',
                    random_state=42,
                    verbose=True)

nn.fit(X_train, y_train)
```

**Activation Functions**

- **Sigmoid**
  \[ \sigma(x) = \frac{1}{1+e^{-x}} \]

- **Tanh**
  \[ \tanh(x) \]

- **ReLU**
  \[ \text{max}(0, x) \]

- **Leaky ReLU**
  \[ \text{max}(0.1x, x) \]

- **Maxout**
  \[ \text{max}(w_1^T x + b_1, w_2^T x + b_2) \]

- **ELU**
  \[ \begin{cases} 
  x & x \geq 0 \\
  \alpha(e^x - 1) & x < 0 
  \end{cases} \]
A tensor is an N-dimensional array of data.
FULLY-CONNECTED NEURAL NETWORKS

Images are tensors!
We need image filters to help us extract features.
“Moore’s Law for CPUs is Dead”

GPU vs. CPU
WHY GPUs EXACTLY?

- CNNs are all about matrix and vector operations (multiplication, addition)
- GPUs can perform parallel multiplication and addition steps per each clock cycle.
TRANSFORMER MODELS
BACK IN THE OLDEN DAYS (5 YEARS AGO)…

- Sophisticated language models are not possible with simpler, traditional neural networks.

- **Multi-level Perceptrons (MLPs):** sequential information is lost (forgotten).

- **Recurrent Neural Networks (RNNs):** They can work with sequential data! Yay!

  ![RNN Diagram]

  "Old man tinkering with RNNs" - Midjourney

  But…
  * Training is slow and sequential.
  * Vanishing gradients limit context length

- **Long Short-Term Memory (LSTMs):** A form of RNN with improved memory…but still slow to train without parallelism.
Attention Is All You Need

Ashish Vaswani*  Google Brain
avaswani@google.com

Noam Shazeer*  Google Brain
noam@google.com

Niki Parmar*  Google Research
nikip@google.com

Jakob Uszkoreit*  Google Research
usz@google.com

Llion Jones*  Google Research
llion@google.com

Aidan N. Gomez†  University of Toronto
aidan@cs.toronto.edu

Łukasz Kaiser*  Google Brain
lukaszkaiser@google.com

Ilia Polosukhin* †
ilia.polosukhin@gmail.com

Abstract
The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

1 Introduction
Recurrent neural networks, long short-term memory [12] and gated recurrent [7] neural networks in particular, have been firmly established as state of the art approaches in sequence modeling and
TRANSFORMER MODELS

- Works only on sequential data
- Uses a new method called “Attention” or ”Self-Attention”
- Replacing CNNs and RNNs across many data modalities
  - (text, images, video, audio)
- Fast to train (very parallelizable)
USES OF TRANSFORMERS

- Large Language Models
- Science and Engineering
- Audio and Speech
- Computer Vision

AlphaFold uses transformers for protein folding

Chemistry
Feed-forward network: after taking information from other tokens, take a moment to think and process this information

Decoder-encoder attention: target token looks at the source queries – from decoder states; keys and values from encoder states

Encoder self-attention: tokens look at each other queries, keys, values are computed from encoder states

Decoder self-attention (masked): tokens look at the previous tokens queries, keys, values are computed from decoder states

Residual connections and layer normalization
DECODER-ONLY TRANSFORMERS

Look at current text sequence  
Predict the next word  
Add word to our current text sequence  
Repeat

No, really, it predicts next tokens.

And that is all. (But that is a lot!)
PARTS OF A DECODER-ONLY TRANSFORMER

- Tokenizer
- Embedding Layer
- Positional Encoding
- Decoder Blocks
  - Self-Attention
  - Multiple-Head Attention
  - Feed-Forward Neural Network
  - Normalization
- Output Layer:
  - Linear layer + Softmax

*ChatGPT is Decoder-Only*
• Neural networks work with numbers, so let’s convert text to numbers first.
• Tokenize at character, sub-word, word, or partial sentence levels.
• Most language models use subword tokens

```
from transformers import AutoTokenizer

tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

# Example of tokenizing text

text = "Hello, world!"
encoded_input = tokenizer(text)
print(encoded_input)
```

• We’re going to go with character-level tokens
  Vocabulary Size = about 70 tokens
  Trivial tokenizer! (Just a lookup table.)

Vocabulary Size = about 70 tokens
Trivial tokenizer! (Just a lookup table.)
### Token Embeddings

Vector = amplitude and direction

<table>
<thead>
<tr>
<th>Token String</th>
<th>Token ID</th>
<th>Embedded Token Vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>'&lt;s&gt;'</td>
<td>0</td>
<td>[0.1150, -0.1438, 0.0555, ...]</td>
</tr>
<tr>
<td>'&lt;pad&gt;'</td>
<td>1</td>
<td>[0.1149, -0.1438, 0.0547, ...]</td>
</tr>
<tr>
<td>'&lt;/s&gt;'</td>
<td>2</td>
<td>[0.0010, -0.0922, 0.1025, ...]</td>
</tr>
<tr>
<td>'&lt;unk&gt;'</td>
<td>3</td>
<td>[0.1149, -0.1439, 0.0548, ...]</td>
</tr>
<tr>
<td>'.'</td>
<td>4</td>
<td>[-0.0651, -0.0622, -0.0002, ...]</td>
</tr>
<tr>
<td>'the'</td>
<td>5</td>
<td>[-0.0340, 0.0068, -0.0844, ...]</td>
</tr>
<tr>
<td>','</td>
<td>6</td>
<td>[0.0483, -0.0214, -0.0927, ...]</td>
</tr>
<tr>
<td>'to'</td>
<td>7</td>
<td>[-0.0439, 0.0201, 0.0189, ...]</td>
</tr>
<tr>
<td>'and'</td>
<td>8</td>
<td>[0.0523, -0.0208, -0.0254, ...]</td>
</tr>
<tr>
<td>'of'</td>
<td>9</td>
<td>[-0.0732, 0.0070, -0.0286, ...]</td>
</tr>
<tr>
<td>'a'</td>
<td>10</td>
<td>[-0.0194, 0.0302, -0.0838, ...]</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
EMBEDDING LAYER

- It is a 2D tensor of shape: $vocabulary\_size \times embedding\_dimension$
- OpenAI GPT-3 (Davinci) embedding layer is: $50,257 \times 12,888$

Vector embeddings capture the deeper semantic context of a word or text chunk...or image...or anything.

The semantics of an object are defined by its multi-dimensional and multi-scale co-occurrence and relationships with other objects in the training data.

Semantic vector embeddings are learned from vast amounts of data.
EMBEDDING LAYER

- A compressed, lossy version of the entire training corpus.
- Gestalt representation of human language, concepts, facts, and more as scraped off the internet.
- Semantics expressed only as hyperdimensional relationships between tokens.
- GPT3: Internet compressed to 2.6GB spreadsheet
- The semantic embedding layer is what the transformer operates on.
- Token embeddings are *learned* as the LLM trains on next token prediction.
OUR AI SHAKESPEARE EMBEDDING LAYER

• Vocab Size = 107 unique characters
• Embedding Size = 64
• Number of Layers = 6
• Number of Attention Heads per Layer: 8
Self-attention
We used multiple heads
Each head focuses on a different semantic or contextual aspect of the input sequence (i.e. context window)
The heads learn what to focus on to optimize the networks
For every token in the input sequence (context window) compute three vectors:

- **Query (Q)** – A vectorized encoding that captures how much each token in the sequence should be attended to relative to the current token.

- **Key (K)** – A vectorized encoding used to represent a scoring of each token’s query.

- **Value (V)** – A vectorized representation of the actual content of the token.
SELF-ATTENTION SCORE CALCULATION

• For every token, compute the dot product between this token’s Q vector and all other K vectors.

• Attention Score = Q * K^T

• The result is an n X n matrix where n = sequence length (context window size)

• Scale by sqrt(len(K))

• Softmax() -> attention weights

• Output = Attention Weights * V vector

Dot product

Algebraic definition:
Taking the vectors $a = [a_1, a_2, ..., a_n]$ and $b = [b_1, b_2, ..., b_n]$ with vector space $n$, the dot product is

$$a \cdot b = \sum_{i=1}^{n} a_i b_i = a_1 b_1 + a_2 b_2 + \cdots + a_n b_n$$

Geometric definition:
The dot product of two Euclidean vectors $a$ and $b$ is

$$a \cdot b = ||a|| ||b|| \cos \theta,$$
where $\theta$ is the angle between vectors and $||a||$ and $||b||$ their respective magnitudes

Big O Notation

Pairwise dot products are $O(n^2)$
Transformers are $O(n^2)$

$n =$ size of context window
SELF-ATTENTION SCORES – PART 2

Multiple heads in parallel
(OpenAI GPT3 = 96 heads)

All dot products and self-attention scores for a given transformer block can be computed in parallel.
(yay GPUs!)

Successive transformer block layers must be sequential

Repeat layers
(GPT3 uses 96 layers)
FEED FORWARD NETWORK (FFN) LAYER

Unlike self-attention scoring, operates on each position separately.

Adds representation capability for each token in the sequence (context window) independently.

Allows network to make complex transformations on the sequence data after self-attention scores are computed.
Transformers work on sequences BUT they don’t track sequential order (attention scores are calculated in parallel!)

Sequential order of tokens is pretty important in language 😊

We need to tell the transformer about the order of words
### POSITIONAL ENCODING

We obviously need another tensor!

**Positional Encoding Matrix for the sequence ‘I am a robot’**

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Index of token</th>
<th>Positional Encoding Matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0</td>
<td>P₀₀ P₀₁ ... P₀d</td>
</tr>
<tr>
<td>am</td>
<td>1</td>
<td>P₁₀ P₁₁ ... P₁d</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>P₂₀ P₂₁ ... P₂d</td>
</tr>
<tr>
<td>Robot</td>
<td>3</td>
<td>P₃₀ P₃₁ ... P₃d</td>
</tr>
</tbody>
</table>

**Sinusoidal Position Encoding**

<table>
<thead>
<tr>
<th>Equation</th>
<th>Graph</th>
<th>Frequency</th>
<th>Wavelength</th>
</tr>
</thead>
<tbody>
<tr>
<td>sin(2πt)</td>
<td><img src="#" alt="Graph" /></td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sin(2 * 2πt)</td>
<td><img src="#" alt="Graph" /></td>
<td>2</td>
<td>1/2</td>
</tr>
<tr>
<td>sin(t)</td>
<td><img src="#" alt="Graph" /></td>
<td>1/2π</td>
<td>2π</td>
</tr>
<tr>
<td>sin(ct)</td>
<td>Depends on c</td>
<td>c/2π</td>
<td>2π/c</td>
</tr>
</tbody>
</table>
SINUSOIDAL POSITIONAL ENCODING

\[ P(k, 2i) = \sin \left( \frac{k}{n^{2i/d}} \right) \]

\[ P(k, 2i + 1) = \cos \left( \frac{k}{n^{2i/d}} \right) \]

Even positions are \( \sin() \)

Odd positions are \( \cos() \)
• If I add these sinusoidal waveforms to the existing values in my token embeddings for my current sequence, doesn’t that change the semantic meaning of my tokens???

• Absolutely. And that’s okay.

• A token’s semantic meaning DOES change depending on its relative position within a sequence.

• Relax. The transformer will figure this all out during training.
The Entire GPT3-175B (Davinci) Transformer Architecture on a Napkin

175 Billion Trainable Parameters

GPT-3 training data[1]:

<table>
<thead>
<tr>
<th>Dataset</th>
<th># tokens</th>
<th>Proportion of tokens within training</th>
</tr>
</thead>
<tbody>
<tr>
<td>Common Crawl</td>
<td>410 billion</td>
<td>60%</td>
</tr>
<tr>
<td>WebText2</td>
<td>19 billion</td>
<td>22%</td>
</tr>
<tr>
<td>Books1</td>
<td>12 billion</td>
<td>8%</td>
</tr>
<tr>
<td>Books2</td>
<td>55 billion</td>
<td>8%</td>
</tr>
<tr>
<td>Wikipedia</td>
<td>3 billion</td>
<td>3%</td>
</tr>
</tbody>
</table>

Training GPT3 Davinci:

- $4.6 Million in Azure
- 335 GPU Years (V100)
- Today: ~$500K
A SHORT WORD ABOUT CLIP
USING CLIP IS TRIVIAL

import torch
import clip
from PIL import Image

device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-B/32", device=device)

image = preprocess(Image.open("CLIP.png").unsqueeze(0).to(device))
text = clip.tokenize(["a diagram", "a dog", "a cat"]).to(device)

with torch.no_grad():
    image_features = model.encode_image(image)
text_features = model.encode_text(text)

    logits_per_image, logits_per_text = model(image, text)
    probs = logits_per_image.softmax(dim=-1).cpu().numpy()

print("Label probs:", probs)  # prints: [[0.9927937 0.00421068 0.00299572]]

https://github.com/openai/CLIP
One way to train a multi-modal embedding layer

“A cute Welsh Corgi dog.”

Learning your semantic embeddings first (GPT)

Train your CNN to predict matching embedding vectors
No, really, it predicts next tokens.
SOME RESOURCES

nanoGPT

available GPT implementations

• https://github.com/karpathy/nanoGPT
  • A 300-line training loop + 300-line GPT model
  • Can load the GPT2 weights
  • Also has a Shakespeare demo. Better than mine. 😊

Andrej Karpathy

’15-17  Co-Founder OpenAI
’17-22  Tesla AI Director
’23        Back at OpenAI!

https://www.youtube.com/@AndrejKarpathy