Transformers and GPT models

GPT Capabilities

- Text generation
- Question answering
- Language translation
- Summarization
- Code generation
- … audio, images

Considerations:

- Ethical concerns: Bias, misinformation
- Computational resources: Training large models is expensive
- Limitations: Lack of true understanding, potential for errors

Let’s train our own GPT model

```
>> python run.py preprocess
The corpus has 142 unique tokens.
SUCCESS
```

Let’s train our own GPT model

```
>> python run.py train
Initializing a new model.
Parameters to be optimized: 4790926

10:10:27 | step 0: train loss 4.9567, valid loss 4.9876
10:11:56 | step 500: train loss 0.1477, valid loss 5.2037
10:13:49 | step 1000: train loss 0.1291, valid loss 5.8455
10:15:12 | step 1500: train loss 0.1191, valid loss 6.1366
10:16:35 | step 2000: train loss 0.1167, valid loss 6.5854
10:17:58 | step 2500: train loss 0.1106, valid loss 6.6639
10:19:20 | step 3000: train loss 0.1105, valid loss 6.7045
10:20:47 | step 3500: train loss 0.1099, valid loss 6.8777
10:22:07 | step 4000: train loss 0.1095, valid loss 6.8912
10:23:28 | step 4500: train loss 0.1105, valid loss 7.3837
10:24:50 | step 4999: train loss 0.1085, valid loss 6.8775
```

Let’s train our own GPT model

```
>> python run.py train --update
```

Let’s train our own GPT model

```
>> python run.py chat
```

FLAIR **Example 1:** Tag Entities in Text

- Let's run **named entity recognition** (NER) over the following example sentence: "*I love Berlin and New York.*"
- Our goal is to identify names in this sentence, and their types.
- To do this, all you need is to make a **Sentence** for this text, load a pre-trained model and use it to predict tags for the sentence:

```
from flair.data import Sentence
from flair.nn import Classifier

# make a sentence
sentence = Sentence('I love Berlin and New York.')
# load the NER tagger
tagger = Classifier.load('ner')
# run NER over sentence
tagger.predict(sentence)
# print the sentence with all annotations
print(sentence)
Sentence[7]: "I love Berlin and New York." → ["Berlin"/LOC, "New York"/LOC]
```

The output shows that both "Berlin" and "New York" were tagged as **location entities** (LOC) in this sentence.

FLAIR **Example 2:** Detect Sentiment

Let's run **sentiment analysis** over the same sentence to determine whether it is **POSITIVE** or **NEGATIVE.**
You can do this with essentially the same code as above. Just instead of loading the 'ner' model, you now load the 'sentiment' model:

```
from flair.data import Sentence
from flair.nn import Classifier

# make a sentence
sentence = Sentence('I love Berlin and New York.')
# load the sentiment tagger
tagger = Classifier.load('sentiment')
# run sentiment analysis over sentence
tagger.predict(sentence)
# print the sentence with all annotations
print(sentence)
Sentence[7]: "I love Berlin and New York." → POSITIVE (0.9982)
```

Tagging parts-of-speech

Syntax is fundamentally language-specific, so each language has different fine-grained parts-of-speech. Flair offers models for many languages:

… in English

```python
from flair.nn import Classifier
from flair.data import Sentence

# load the model
tagger = Classifier.load('pos')
# make a sentence
sentence = Sentence('Dirk went to the store.')
# predict NER tags
tagger.predict(sentence)
# print sentence with predicted tags
print(sentence)
```

```
Sentence[6]: "Dirk went to the store." → ["Dirk"/NNP, "went"/VBD, "to"/IN, "the"/DT, "store"/NN, "."/.]
```

"*Dirk*" is a proper noun (tag: NNP), and "*went*" is a past tense verb (tag: VBD).

... in German

```
from flair.nn import Classifier
from flair.data import Sentence

# load the model
tagger = Classifier.load('de-pos')
# make a sentence
sentence = Sentence('Dort hatte er einen Hut gekauft.')
# predict NER tags
tagger.predict(sentence)
# print sentence with predicted tags
print(sentence)
```

```
Sentence[7]: "Dort hatte er einen Hut gekauft." →
["Dort"/ADV, "hatte"/VAFIN, "er"/PPER, "einen"/ART,
"Hut"/NN, "gekauft"/VVPP, "."/$.]
```

... in Ukrainian

```
from flair.nn import Classifier
from flair.data import Sentence

# load the model
tagger = Classifier.load('pos-ukrainian')
# make a sentence
sentence = Sentence("Сьогодні в Знам’янці проживають
нащадки поета — родина Шкоди.")
# predict NER tags
tagger.predict(sentence)
# print sentence with predicted tags
print(sentence)
```

... in Arabic

```
from flair.nn import Classifier
from flair.data import Sentence

# load the model
tagger = Classifier.load('ar-pos')
# make a sentence
sentence = Sentence('عمرو عادلي أستاذ للاقتصاد السياسي المساعد في الجامعة الأمريكية بالقاهرة .')
# predict NER tags
tagger.predict(sentence)
# print sentence with predicted tags
print(sentence)
```

Tagging universal parts-of-speech (uPoS)

Universal parts-of-speech are a set of minimal syntactic units that exist across languages.

For instance, most languages will have VERBs or NOUNs.

FLAIR was trained over 14 langages to tag upos in **multilingual text**.

```
from flair.nn import Classifier
from flair.data import Sentence

# load model
tagger = Classifier.load('pos-multi')

# text with English and German sentences
sentence = Sentence('George Washington went to Washington. Dort kaufte er einen Hut.')
# predict PoS tags
tagger.predict(sentence)
# print sentence with predicted tags
print(sentence)
Sentence: "George Washington went to Washington . Dort kaufte er einen Hut ."
→ ["George"/PROPN, "Washington"/PROPN, "went"/VERB, "to"/ADP, "Washington"/PROPN, "."/PUNCT]
→ ["Dort"/ADV, "kaufte"/VERB, "er"/PRON, "einen"/DET, "Hut"/NOUN, "."/PUNCT]
```

List of POS Models

ID	Task	Language	Training Dataset	Accuracy	Contributor / Notes
‘pos’	POS-tagging	English	Ontonotes	**98.19** (Accuracy)	
‘pos-fast’	POS-tagging	English	Ontonotes	**98.1** (Accuracy)	(fast model)
‘upos’	POS-tagging (universal)	English	Ontonotes	**98.6** (Accuracy)	
‘upos-fast’	POS-tagging (universal)	English	Ontonotes	**98.47** (Accuracy)	(fast model)
‘pos-multi’	POS-tagging	Multilingual	UD Treebanks	**96.41** (average acc.)	(12 languages)

List of POS Models

ID	Task	Language	Training Dataset	Accuracy	Contributor / Notes
'da-pos'	POS-tagging	Danish	Danish Dependency Treebank		AmaliePauli
'pt-pos-clinical'	POS-tagging	Portuguese	PUCPR	**92.39**	LucasFerroHAILab for clinical texts
'pos-ukrainian'	POS-tagging	Ukrainian	Ukrainian UD	**97.93** (F1)	dchaplinsky

Classic Word Embeddings

Classic word embeddings are static and word-level.

- Each distinct word gets exactly one pre-computed embedding.
- Most embeddings fall under this class, including the popular GloVe or Komninos embeddings (e.g. Wikipedia).

```python
from flair.embeddings import WordEmbeddings

# init embedding
glove_embedding = WordEmbeddings('glove')
# create sentence.
sentence = Sentence('The grass is green .')
# embed a sentence using glove.
glove_embedding.embed(sentence)
# now check out the embedded tokens.
for token in sentence:
    print(token)
    print(token.embedding)
```

This prints out the tokens and their embeddings. GloVe embeddings are Pytorch vectors of dimensionality 100.

ID	Language	Embedding
‘en-glove’ (or ‘glove’)	English	GloVe embeddings
‘en-extvec’ (or ‘extvec’)	English	Komninos embeddings
‘en-crawl’ (or ‘crawl’)	English	FastText embeddings over Web crawls
‘en-twitter’ (or ‘twitter’)	English	Twitter embeddings
‘en’ (or ‘en-news’ or ‘news’)	English	FastText embeddings over news and wikipedia data
‘de’	German	German FastText embeddings
‘fr’	French	French FastText embeddings
‘es’	Spanish	Spanish FastText embeddings

Custom embeddings

- Loading Gensim embeddings:

```
custom_embedding = WordEmbeddings('path/to/your/custom/embeddings.gensim')
```

- Converting FastText embeddings to Gensim:

```
import gensim
word_vectors = gensim.models.KeyedVectors.load_word2vec_format('/path/to/embeddings.txt', binary=False)
word_vectors.save('/path/to/converted’)
```

- However, FastText embeddings have the functionality of returning vectors for out of vocabulary words using the sub-word information.

Prompts engineering, multimodal learning, and LLMs

Alejandro Veloz

What is a **prompt**?

- A prompt is a specific input or instruction given to an AI model to elicit a desired response or output.
- It serves as a starting point for the AI to generate text, answer questions, or perform tasks. Prompts can range from simple questions to complex scenarios or instructions.

How prompts are fed into OpenAI tools

- OpenAI tools, such as GPT-3 or ChatGPT, process prompts through their **API** or **user interface.**
- General workflow:

1. User inputs a prompt
2. The prompt is tokenized and encoded
3. The encoded prompt is fed into the AI model
4. The model generates a response based on its training
5. The response is decoded and presented to the user

Types of prompts

There are several types of prompts, including:

1. **Open-ended**: General questions or statements that allow for diverse responses
2. **Instructional**: Specific directions for the AI to follow
3. **Context-based**: Prompts that provide background information
4. **Role-playing**: Asking the AI to assume a particular persona
5. **Completion**: Partial sentences or ideas for the AI to finish
6. **Creative**: Prompts designed to generate imaginative content

Types of models

There are several types of models, including:

Models overview

The OpenAI API is powered by a diverse set of models with different capabilities and price points. You can also make customizations to our models for your specific use case with fine-tuning.

MODEL	DESCRIPTION
GPT-4o	Our high-intelligence flagship model for complex, multi-step tasks
GPT-4o mini	Our affordable and intelligent small model for fast, lightweight tasks
o1-preview and o1-mini	Language models trained with reinforcement learning to perform complex reasoning.
GPT-4 Turbo and GPT-4	The previous set of high-intelligence models
GPT-3.5 Turbo	A fast, inexpensive model for simple tasks
DALL·E	A model that can generate and edit images given a natural language prompt
TTS	A set of models that can convert text into natural sounding spoken audio
Whisper	A model that can convert audio into text

Examples of prompts (user message)

```
const response = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: [
    {
      "role": "user",
      "content": [
        {
          "type": "text",
          "text": "Write a haiku about programming."
        }
      ]
    }
  ]
});
```

Examples of prompts (system messages)

```
const response = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: [
    {
      "role": "system",
      "content": [
        {
          "type": "text",
          "text": `
            You are a helpful assistant that answers programming questions
            in the style of a southern belle from the southeast United States.
          `
        }
      ]
    },
    {
      "role": "user",
      "content": [
        {
          "type": "text",
          "text": "Are semicolons optional in JavaScript?"
        }
      ]
    }
  ]
});
```

Examples of prompts (assistant messages)

Capture the results of a previous text generation result, and making a new request based on that.

They can also be used to provide examples to the model for how it should respond to the current request - a technique known as few-shot learning.

```
const response = await openai.chat.completions.create({
  model: "gpt-4o",
  messages: [
    {
      "role": "user",
      "content": [{ "type": "text", "text": "knock knock." }]
    },
    {
      "role": "assistant",
      "content": [{ "type": "text", "text": "Who's there?" }]
    },
    {
      "role": "user",
      "content": [{ "type": "text", "text": "Orange." }]
    }
  ]
});
```

How ChatGPT was trained

ChatGPT was trained using a method called Reinforcement Learning from Human Feedback (RLHF). The process involved:

1. **Pretraining**: Learning from a large corpus of internet text
2. **Fine-tuning**: Using supervised learning on human-written conversations
3. **Reward modeling**: Training a reward model based on human preferences
4. **Reinforcement learning**: Optimizing the model’s responses using the reward model

Can ChatGPT learn more from our data?

ChatGPT’s base model doesn’t learn from individual interactions. However, OpenAI has introduced features like:

- **Fine-tuning**: Allowing organizations to customize the model on their data
- **GPT-4 with retrieval**: Enabling the model to access and use external knowledge bases
- **ChatGPT plugins**: Extending the model’s capabilities through third-party integrations

Can ChatGPT learn more from our data?

At a high level, fine-tuning involves the following steps:

1. Prepare and upload training data
2. Train a new fine-tuned model
3. Evaluate results and go back to step 1 if needed
4. Use your fine-tuned model

Once a model has been fine-tuned, **you won't need to provide as many examples in the prompt.** This saves costs and enables lower-latency requests.

Which models can be fine-tuned?

Fine-tuning is currently available for the following models:

- gpt-4o-2024-08-06
- gpt-4o-mini-2024-07-18
- gpt-4-0613
- gpt-3.5-turbo-0125
- gpt-3.5-turbo-1106
- gpt-3.5-turbo-0613

You can also fine-tune a fine-tuned model, which is useful if you acquire additional data and don’t want to repeat the previous training steps.

Example data format for fine-tuning


```
{"messages": [{"role": "system", "content": "Marv is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "What's the capital of France?"}, {"role": "assistant", "content": "Paris, as if everyone doesn't know that already."}]}
{"messages": [{"role": "system", "content": "Marv is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "Who wrote 'Romeo and Juliet'?"}, {"role": "assistant", "content": "Oh, just some guy named William Shakespeare. Ever heard of him?"}]}
{"messages": [{"role": "system", "content": "Marv is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "How far is the Moon from Earth?"}, {"role": "assistant", "content": "Around 384,400 kilometers. Give or take a few, like that really matters."}]}
```

Perform fine-tuning

```python
from openai import OpenAI
client = OpenAI()

client.fine_tuning.jobs.create(
  training_file="file-abc123",
  model="gpt-4o-mini-2024-07-18"
)
```

Using a fine-tuned model

```
from openai import OpenAI
client = OpenAI()

completion = client.chat.completions.create(
  model="ft:gpt-4o-mini:my-org:custom_suffix:id",
  messages=[
    {"role": "system", "content": "You are a
helpful assistant."},
    {"role": "user", "content": "Hello!"}
  ]
)
print(completion.choices[0].message)
```

Analyzing a fine-tuned model


```json
{
    "object": "fine_tuning.job.event",
    "id": "ftevent-abc-123",
    "created_at": 1693582679,
    "level": "info",
    "message": "Step 300/300: training loss=0.15, validation loss=0.27, full validation loss=0.40",
    "data": {
        "step": 300,
        "train_loss": 0.14991648495197296,
        "valid_loss": 0.26569826706596045,
        "total_steps": 300,
        "full_valid_loss": 0.4032616495084362,
        "train_mean_token_accuracy": 0.9444444179534912,
        "valid_mean_token_accuracy": 0.9565217391304348,
        "full_valid_mean_token_accuracy": 0.9089635854341737
    },
    "type": "metrics"
}
```

Integrating LLMs in everyday life

Large Language Models (LLMs) are being integrated into various aspects of daily life:

- **Personal assistants**: Enhancing digital assistants like Siri or Alexa
- **Content creation**: Assisting in writing, coding, and creative tasks
- **Education**: Providing personalized tutoring and explanations
- **Customer service**: Powering chatbots and support systems
- **Research and analysis**: Aiding in data interpretation and summarization

But life is multimodal

While text-based LLMs are powerful, human experience and interaction involve multiple senses and forms of data.

This realization has led to the development of multimodal AI systems that can process and generate various types of information.

What is multimodal data?

Multimodal data refers to information presented in multiple forms or "modalities." Common modalities include:

- Text
- Images
- Audio
- Video
- Numerical data
- Sensor readings

Multimodal data provides a richer, more comprehensive representation of information, mimicking how humans perceive and interact with the world.

Can ChatGPT-type models learn multimodal data?

Yes, ChatGPT-type models can be adapted to learn multimodal data. Recent developments include:

- **GPT-4**: Capable of processing both text and images
- **DALL-E**: Generating images from text descriptions
- **Whisper**: Transcribing and translating audio to text

These advancements demonstrate the potential for LLMs to handle diverse data types.

Examples of prompts (images)

What is in this image?

```python
from openai import OpenAI

client = OpenAI()

response = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[
    {
      "role": "user",
      "content": [
        {"type": "text", "text": "What’s in this image?"},
        {
          "type": "image_url",
          "image_url": {
            "url": "https://upload.wikimedia.org/...walk.jpg",
          },
        },
      ],
    }
  ],
  max_tokens=300,
)

print(response.choices[0])
```

Examples of prompts (local images)

```python
import base64
from openai import OpenAI

client = OpenAI()

# Function to encode the image
def encode_image(image_path):
  with open(image_path, "rb") as image_file:
    return base64.b64encode(image_file.read()).decode('utf-8')

# Path to your image
image_path = "path_to_your_image.jpg"

# Getting the base64 string
base64_image = encode_image(image_path)

response = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[
    {
      "role": "user",
      "content": [
        {
          "type": "text",
          "text": "What is in this image?",
        },
        {
          "type": "image_url",
          "image_url": {
            "url":  f"data:image/jpeg;base64,{base64_image}"
          },
        },
      ],
    }
  ],
)

print(response.choices[0])
```

What is in this image?

Examples of prompts (multiple images)

What is in this image?

```
from openai import OpenAI

client = OpenAI()
response = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[
    {
      "role": "user",
      "content": [
        {
          "type": "text",
          "text": "What are in these images? Is there any difference 
between them?",
        },
        {
          "type": "image_url",
          "image_url": {
            "url": "https://upload.wikimedia...-boardwalk.jpg",
          },
        },
        {
          "type": "image_url",
          "image_url": {
            "url": "https://upload.wikimedia.or...dwalk.jpg",
          },
        },
      ],
    }
  ],
  max_tokens=300,
)
print(response.choices[0])
```

Examples of prompts (low/high fidelity image understanding)

What is in this image?

```
from openai import OpenAI

client = OpenAI()

response = client.chat.completions.create(
  model="gpt-4o-mini",
  messages=[
    {
      "role": "user",
      "content": [
        {"type": "text", "text": "What's in this image?"},
        {
          "type": "image_url",
          "image_url": {
            "url":
"https://upload.wikimedia.org/wikipedia/commons/thumb/d/dd/Gfp-wisconsin-madison-the-nat
ure-boardwalk.jpg/2560px-Gfp-wisconsin-madison-the-nature-boardwalk.jpg",
            "detail": "high"
          },
        },
      ],
    }
  ],
  max_tokens=300,
)

print(response.choices[0].message.content)
```

Choosing low/high fidelity image understanding modes

By controlling the detail parameter, which has three options, low, high, or auto, you have control over how the model processes the image and generates its textual understanding.

By default, the model will use the auto setting which will look at the image input size and decide if it should use the low or high setting.

- low will enable the “low res” mode. The model will receive a low-res 512px x 512px version of the image, and represent the image with a budget of 85 tokens. This allows the API to return faster responses and consume fewer input tokens for use cases that do not require high detail.
- high will enable “high res” mode, which first allows the model to first see the low res image (using 85 tokens) and then creates detailed crops using 170 tokens for each 512px x 512px tile.

Prompts - limitations

- Medical images: The model is not suitable for interpreting specialized medical images like CT scans and shouldn’t be used for medical advice.
- Non-English: The model may not perform optimally when handling images with text of non-Latin alphabets, such as Japanese or Korean.
- Small text: Enlarge text within the image to improve readability, but avoid cropping important details.
- Rotation: The model may misinterpret rotated / upside-down text or images.
- Visual elements: The model may struggle to understand graphs or text where colors or styles like solid, dashed, or dotted lines vary.
- Spatial reasoning: The model struggles with tasks requiring precise spatial localization, such as identifying chess positions. Accuracy: The model may generate incorrect descriptions or captions in certain scenarios.
- Image shape: The model struggles with panoramic and fisheye images.
- Metadata and resizing: The model doesn’t process original file names or metadata, and images are resized before analysis, affecting their original dimensions.
- Counting: May give approximate counts for objects in images.
- CAPTCHAS: For safety reasons, we have implemented a system to block the submission of CAPTCHAs.

Examples of prompts (audio generation)

```
import base64
from openai import OpenAI

client = OpenAI()

completion = client.chat.completions.create(
    model="gpt-4o-audio-preview",
    modalities=["text", "audio"],
    audio={"voice": "alloy", "format": "wav"},
    messages=[
        {
            "role": "user",
            "content": "Is a golden retriever a good family dog?"
        }
    ]
)

print(completion.choices[0])

wav_bytes = base64.b64decode(completion.choices[0].message.audio.data)
with open("dog.wav", "wb") as f:
    f.write(wav_bytes)
```

Examples of prompts (text to speech)

```
from pathlib import Path
from openai import OpenAI
client = OpenAI()

speech_file_path = Path(__file__).parent /
"speech.mp3"
response = client.audio.speech.create(
  model="tts-1",
  voice="alloy",
  input="Today is a wonderful day to build something
people love!"
)

response.stream_to_file(speech_file_path)
```

Examples of prompts (speech to text)

```
from openai import OpenAI
client = OpenAI()

audio_file= open("/path/to/file/audio.mp3", "rb")
transcription = client.audio.transcriptions.create(
  model="whisper-1",
  file=audio_file
)
print(transcription.text)
{
  "text": "Imagine the wildest idea that you've ever had,
and you're curious about how it might scale to something
that's a 100, a 1,000 times bigger.
....
}
```

Examples of prompts (embeddings)

```python
from openai import OpenAI
client = OpenAI()

response = client.embeddings.create(
    input="Your text string goes here",
    model="text-embedding-3-small"
)

print(response.data[0].embedding)
```


```json
{
  "object": "list",
  "data": [
    {
      "object": "embedding",
      "index": 0,
      "embedding": [
        -0.006929283495992422,
        -0.005336422007530928,
        ... (omitted for spacing)
        -4.547132266452536e-05,
        -0.024047505110502243
      ],
    }
  ],
  "model": "text-embedding-3-small",
  "usage": {
    "prompt_tokens": 5,
    "total_tokens": 5
  }
}
```

LLM architectures in multimodal data

Multimodal LLM architectures typically involve:

1. **Encoders**: Specialized for each modality (e.g., vision transformers for images)
2. **Fusion mechanisms**: Combining information from different modalities
3. **Cross-attention layers**: Allowing interaction between modalities
4. **Decoder**: Generating outputs based on fused multimodal representations

Examples include CLIP (Contrastive Language-Image Pre-training) and FLAMINGO (Few-shot Learning with Auxiliary Modalities).

Taking better decisions based on multimodal data

Multimodal AI systems can enhance decision-making by:

- Providing more comprehensive insights
- Reducing ambiguity through cross-modal validation
- Capturing nuances that single-modality systems might miss
- Enabling more natural and context-aware interactions
- Supporting complex tasks that require integration of diverse information

Recommendation systems

Recommendation systems leverage multimodal data to provide personalized suggestions:

- **Content-based**: Using features from text, images, and user behavior
- **Collaborative filtering**: Incorporating diverse user interactions
- **Hybrid approaches**: Combining multiple recommendation strategies
- **Context-aware**: Considering time, location, and other contextual factors
- **Deep learning models**: Processing complex multimodal features for accurate recommendations

Example problem

Let’s consider a multimodal recommendation system for a streaming platform:

Input: - User viewing history (text and metadata) - Content thumbnails (images) - Movie/show trailers (video) - User ratings and reviews (text and numerical data)

Task: Recommend personalized content to the user

Approach: 1. Encode each modality separately 2. Fuse multimodal features 3. Use a deep neural network to predict user preferences 4. Generate and rank recommendations 5. Present top suggestions to the user

Challenges and future directions

- **Data alignment**: Ensuring coherence across modalities
- **Scalability**: Handling large-scale multimodal datasets
- **Interpretability**: Understanding decisions in multimodal systems
- **Privacy concerns**: Protecting user data across diverse modalities
- **Ethical considerations**: Addressing biases in multimodal AI
- **Advancements in cross-modal learning**: Improving transfer between modalities
- **Real-time processing**: Enabling faster multimodal interactions