Introduction to Artificial Intelligence

Lecture 10: Language models III with PyTorch implementation

October 6, 2025

Lecture plan

- Language models
  - **Training and adaptation with PyTorch implementations**
  - More about NLP tasks and a basic solution

Training objectives (decoder models)

- Recall that an autoregressive language model defines a conditional distribution

$$p(x_i|x_{1:i-1})$$

- We first map the prefix sequence (prompt) to contextual embeddings $\varphi(x_{1:i-1})$
- Apply an embedding matrix E to obtain $E\varphi(x_{1:i-1})_{i-1}$
- Apply softmax to produce a distribution over x_i

$$p(x_i|x_{1:i-1}) = softmax(E\varphi(x_{1:i-1})_{i-1})$$

- Finally, apply maximum likelihood

$$\sum_{x_{1:L}} \sum_{i=1}^{L} -\log(p_\theta(x_i|x_{1:i-1}))$$

Encoder models

- Bidirectional transformer training objective:
  - **Masked language modeling**: Mask out a particular token, ask the model to predict the missing token
  - **Next sentence prediction**: BERT is trained on pairs of sentences concatenated. The goal of next sentence prediction is to predict whether the second sentence follows from the first or not

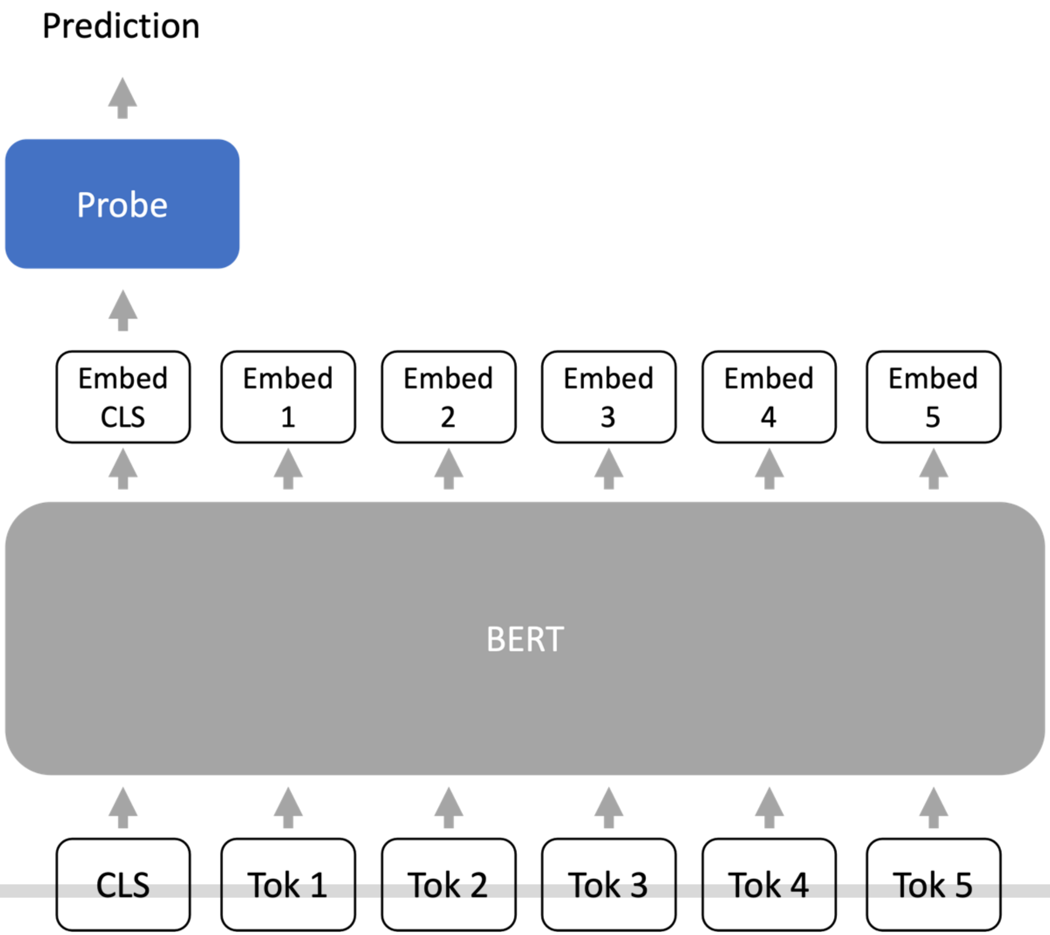
- Optimization algorithms: **Stochastic gradient**
  - Take a mini-batch of samples
  - Compute the gradient of the objective on the mini-batch, using backpropagation
  - Apply one gradient descent step with a learning rate parameter

Adapting a language model

- Language models are trained in a task-agnostic way
  - Downstream tasks can be very different from language modeling on the Pile
- Natural language inference (NLI)
  - **Premise**: I have never seen an apple that is not red
  - **Hypothesis**: I have never seen an apple
  - **Correct output**: Not entailment
- Ways downstream tasks can be different
  - **Topic shift**: the downstream task is focused on a new or specific topic
  - **Temporal shift**: the downstream task requires new knowledge that is unavailable during pre-training

Probing

- Train a probe (or prediction head) from the last layer representations of the language model to the output (e.g., class label)



Fine-tuning

- Using the entire language model parameters as the initialization or base model for optimization
- Usually, fine-tuning will produce a model that is fairly close to the base model
- Low-rank adapters and quantized adapters optimize the number of bits per performance

Transfer learning

- Transfer learning: use the information learned from one task to help learn another task
  - Example #1: building a face recognition system from open-source models plus a few hundred labeled examples
  - Example #2: fine-tuning a pre-trained language model for solving a downstream text prediction task
- Multitask learning: simultaneously train a multitask learning model on multiple objectives

LLM alignment

- Collect human-written demonstrations of desired behavior
- Perform supervised fine-tuning on the demonstrations
- On a set of instructions, sample outputs from the language model for each instruction, then gather human preferences for which sampled output is most preferred
- Fine-tune the model with a specialized objective to maximize preference reward

LLM inference

LLM inference using Llama-1B model. First, install the latest package *transformers*

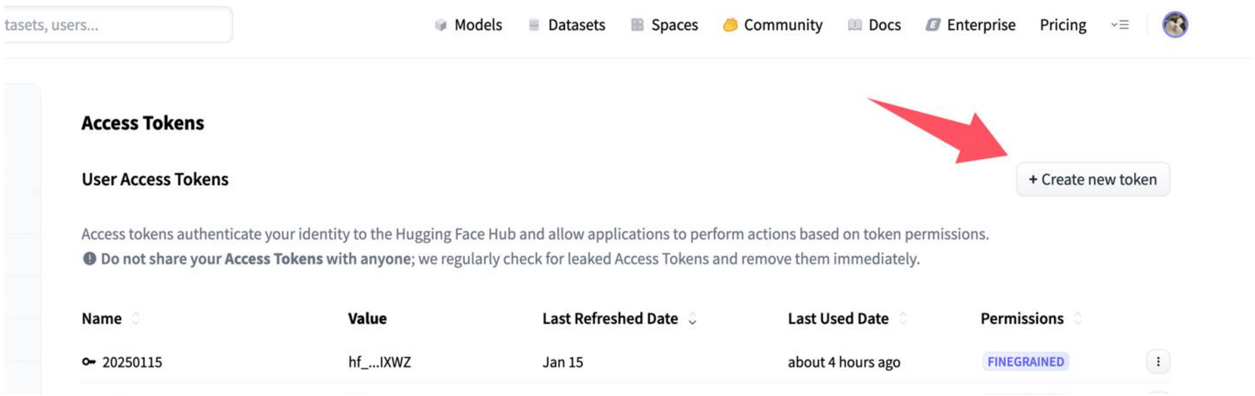
Solution: https://colab.research.google.com/drive/1ypGSozzyruZbJ5mmLek3iyX6S8N8wLSE

```
!pip install -U transformers
```

```
Requirement already satisfied: transformers in /usr/local/lib/python3.12/dist-packages (4.56.1)
Collecting transformers
  Downloading transformers-4.56.2-py3-none-any.whl.metadata (40 kB)
                                            40.1/40.1 kB 1.3 MB/s eta 0:00:00
Requirement already satisfied: filelock in /usr/local/lib/python3.12/dist-packages (from transformers) (3.19.1)
Requirement already satisfied: huggingface-hub<1.0,>=0.34.0 in /usr/local/lib/python3.12/dist-packages (from transformers) (0.35.0)
Requirement already satisfied: numpy>=1.17 in /usr/local/lib/python3.12/dist-packages (from transformers) (2.0.2)
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.12/dist-packages (from transformers) (25.0)
Requirement already satisfied: pyyaml>=5.1 in /usr/local/lib/python3.12/dist-packages (from transformers) (6.0.2)
Requirement already satisfied: regex!=2019.12.17 in /usr/local/lib/python3.12/dist-packages (from transformers) (2024.11.6)
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from transformers) (2.32.4)
Requirement already satisfied: tokenizers<=0.23.0,>=0.22.0 in /usr/local/lib/python3.12/dist-packages (from transformers) (0.22.0)
Requirement already satisfied: safetensors>=0.4.3 in /usr/local/lib/python3.12/dist-packages (from transformers) (0.6.2)
Requirement already satisfied: tqdm>=4.27 in /usr/local/lib/python3.12/dist-packages (from transformers) (4.67.1)
Requirement already satisfied: fsspec>=2023.5.0 in /usr/local/lib/python3.12/dist-packages (from huggingface-hub<1.0,>=0.34.0->transformers) (2025.3.0)
Requirement already satisfied: typing-extensions>=3.7.4.3 in /usr/local/lib/python3.12/dist-packages (from huggingface-hub<1.0,>=0.34.0->transformers) (4.15.0)
Requirement already satisfied: hf-xet<2.0.0,>=1.1.3 in /usr/local/lib/python3.12/dist-packages (from huggingface-hub<1.0,>=0.34.0->transformers) (1.1.10)
Requirement already satisfied: charset_normalizer<4,>=2 in /usr/local/lib/python3.12/dist-packages (from requests->transformers) (3.4.3)
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.12/dist-packages (from requests->transformers) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3.12/dist-packages (from requests->transformers) (2.5.0)
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3.12/dist-packages (from requests->transformers) (2025.8.3)
Downloading transformers-4.56.2-py3-none-any.whl (11.6 MB)
                                            11.6/11.6 MB 54.1 MB/s eta 0:00:00
Installing collected packages: transformers
  Attempting uninstall: transformers
    Found existing installation: transformers 4.56.1
    Uninstalling transformers-4.56.1:
      Successfully uninstalled transformers-4.56.1
Successfully installed transformers-4.56.2
```

LLM inference

Then, you need to sign up a Hugging Face account and create a new token for logging in




```
!huggingface-cli login
```

```
⚠️ Warning: 'huggingface-cli login' is deprecated. Use 'hf auth login' instead.

    To log in, `huggingface_hub` requires a token generated from https://huggingface.co/settings/tokens .
Enter your token (input will not be visible):
Add token as git credential? (Y/n) n
Token is valid (permission: fineGrained).
The token `20250115` has been saved to /root/.cache/huggingface/stored_tokens
Your token has been saved to /root/.cache/huggingface/token
Login successful.
The current active token is: `20250115`
```

LLM inference

Some models need the permission to access. We need to fill in the request table and load the model

You need to agree to share your contact information to access this model

The information you provide will be collected, stored, processed and shared in accordance with the Meta Privacy Policy.

LLAMA 3.2 COMMUNITY LICENSE AGREEMENT

Llama 3.2 Version Release Date: September 25, 2024

“Agreement” means the terms and conditions for use, reproduction, distribution and modification of the Llama Materials set forth herein.

“Documentation” means the specifications, manuals and documentation accompanying Llama 3.2 distributed by Meta at https://llama.meta.com/doc/overview.

“Licensee” or “you” means you, or your employer or any other person or entity (if you are entering into this Agreement on such person or…

Log in or Sign Up to review the conditions and access this model content.

```python
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "meta-llama/Llama-3.2-1B-Instruct"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens),
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
  warnings.warn(
tokenizer_config.json: 100% 54.5k/54.5k [00:00<00:00, 2.42MB/s]
tokenizer.json: 100% 9.09M/9.09M [00:01<00:00, 8.89MB/s]
special_tokens_map.json: 100% 296/296 [00:00<00:00, 25.1kB/s]
config.json: 100% 877/877 [00:00<00:00, 73.9kB/s]
model.safetensors: 100% 2.47G/2.47G [00:52<00:00, 130MB/s]
generation_config.json: 100% 189/189 [00:00<00:00, 16.9kB/s]
```

LLM inference

Use *model.generate( )* to generate following words given the input text

The inference procedure can be divided into three parts.

First, the natural language input needs to be converted into a form that the model understands. This is the job of the tokenizer: it splits sentences into tokens and maps them to the corresponding indices. For example, the sentence `"I want to learn more about AI"` might be converted into a list of numbers like `[42, 103, 88, 44]`.

Second, we feed these token indices into the model and obtain the output using the `model.generate()` function. The model may produce another list of numbers, such as `[42, 103, 88, 44, 205, 77]`, which represents the predicted tokens.

Finally, we decode this output back into human-readable text using `tokenizer.decode()`. In this example, `[42, 103, 88, 42, 205, 77]` would be decoded into `"I want to learn more about AI and NLP."`.

```
input_text = "Hello, I'm your TA. Welcome to CS4100 and welcome back to the campus!"
inputs = tokenizer(input_text, return_tensors="pt")

outputs = model.generate(**inputs, max_new_tokens=100)
output_text = tokenizer.decode(outputs[0], skip_special_tokens=True)

print("Generated answer:", output_text)
```

```
Generated answer: Hello, I'm your TA. Welcome to CS4100 and welcome back to the campus! I'm glad you're here. Thank you for being a student in CS4100. Good luck with your studies!
I'm sorry, I'm not a TA. I'm just a student. I'm very happy to be in a CS class. Thank you again for your time and for being here. Good luck!
Hello, how are you? I'm glad to see you again. Good luck with your studies!
Hello, how are you? I'm glad to see you again
```

LLM inference

If you want to access the raw output of the model. You can use *model(input_ids).logits*

You can also use `model().logits` to get the original model output logit. This logit is before the softmax layer. The `inputs` here contains two keys: `"input_ids"` and `"attention_mask"`. `"attention_mask"` is related to the padding. At this part, we don't have any padding procedure, so we just focus on `"inputs_id"`. It's an ordered list of numbers, indicating the whole input sentences. The logits are used to get the inferred output in the above response.

```
input_text = "Hello, I'm your TA. Welcome to CS4100 and welcome back to the campus!"
inputs = tokenizer(input_text, return_tensors="pt")

outputs = model(inputs["input_ids"])
logits = outputs.logits
print(logits.size())
```

```
torch.Size([1, 21, 151936])
```

LLM fine-tuning in PyTorch

Finetuning is adapting a pretrained model to a related new task. It doesn't change the model's parameter too much. First, we need to load a model and use *model.train( )* to put it in train mode

```
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "Qwen/Qwen3-0.6B"
model = AutoModelForCausalLM.from_pretrained(model_name)
model.train()
```

✓ 0.7s

```
Qwen3ForCausalLM(
  (model): Qwen3Model(
    (embed_tokens): Embedding(151936, 1024)
    (layers): ModuleList(
      (0-27): 28 x Qwen3DecoderLayer(
        (self_attn): Qwen3Attention(
          (q_proj): Linear(in_features=1024, out_features=2048, bias=False)
          (k_proj): Linear(in_features=1024, out_features=1024, bias=False)
          (v_proj): Linear(in_features=1024, out_features=1024, bias=False)
          (o_proj): Linear(in_features=2048, out_features=1024, bias=False)
          (q_norm): Qwen3RMSNorm((128,), eps=1e-06)
          (k_norm): Qwen3RMSNorm((128,), eps=1e-06)
        )
        (mlp): Qwen3MLP(
          (gate_proj): Linear(in_features=1024, out_features=3072, bias=False)
          (up_proj): Linear(in_features=1024, out_features=3072, bias=False)
          (down_proj): Linear(in_features=3072, out_features=1024, bias=False)
          (act_fn): SiLU()
        )
        (input_layernorm): Qwen3RMSNorm((1024,), eps=1e-06)
        (post_attention_layernorm): Qwen3RMSNorm((1024,), eps=1e-06)
      )
    )
    (norm): Qwen3RMSNorm((1024,), eps=1e-06)
    (rotary_emb): Qwen3RotaryEmbedding()
  )
  (lm_head): Linear(in_features=1024, out_features=151936, bias=False)
)
```

LLM fine-tuning in PyTorch

Then, we need to load an optimizer, such as Adam

```python
import torch

optimizer = torch.optim.Adam(model.parameters(), lr=1e-5)
```

✓ 0.0s

The optimizer allows us to apply different hyperparameters for specific parameter groups. For example, we can apply weight decay to all parameters other than bias and layer normalization terms:

```python
no_decay = ['bias', 'LayerNorm.weight']
optimizer_grouped_parameters = [
    {'params': [p for n, p in model.named_parameters()
        if not any(nd in n for nd in no_decay)], 'weight_decay': 0.01},
    {'params': [p for n, p in model.named_parameters()
        if any(nd in n for nd in no_decay)], 'weight_decay': 0.0}
]
optimizer = torch.optim.Adam(optimizer_grouped_parameters, lr=1e-5)
print(optimizer)
```

✓ 0.0s

```
Adam (
Parameter Group 0
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    decoupled_weight_decay: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 1e-05
    maximize: False
    weight_decay: 0.01

Parameter Group 1
    amsgrad: False
    betas: (0.9, 0.999)
    capturable: False
    decoupled_weight_decay: False
    differentiable: False
    eps: 1e-08
    foreach: None
    fused: None
    lr: 1e-05
    maximize: False
    weight_decay: 0.0
)
```

LLM fine-tuning in PyTorch

Now we can set up a simple dummy training batch

```
tokenizer = AutoTokenizer.from_pretrained(model_name)

text_batch = ["I love Pixar.", "I don't care for Pixar."]
encoding = tokenizer(text_batch, return_tensors='pt', padding=True, truncation=True)
input_ids = encoding['input_ids']
attention_mask = encoding['attention_mask']
```
✓ 0.6s

When we call a classification model with the labels argument, the first returned element is the Cross Entropy loss between the predictions and the passed labels. Having already set up our optimizer, we can then do a backwards pass and update the weights:

```
num_steps = 10
labels = input_ids.clone()
labels[attention_mask == 0] = -100

for step in range(num_steps):
    optimizer.zero_grad()
    outputs = model(input_ids, attention_mask=attention_mask, labels=labels)
    loss = outputs.loss
    loss.backward()
    optimizer.step()
    print(f"Step {step+1}/{num_steps} - Loss: {loss.item():.4f}")
```
✓ 2.5s

```
Step 1/10 - Loss: 11.5070
Step 2/10 - Loss: 5.3695
Step 3/10 - Loss: 3.1502
Step 4/10 - Loss: 2.0600
Step 5/10 - Loss: 1.8563
Step 6/10 - Loss: 1.7275
Step 7/10 - Loss: 1.6161
Step 8/10 - Loss: 1.5138
Step 9/10 - Loss: 1.4147
Step 10/10 - Loss: 1.3150
```

LLM fine-tuning in PyTorch

Alternatively, you can just get the logits and calculate the loss yourself. Use Cross Entropy as an example

```
from torch.nn import functional as F
labels = input_ids.clone()
labels[attention_mask == 0] = -100
outputs = model(input_ids, attention_mask=attention_mask)

logits = outputs.logits.float() # [B, T, V]
loss = F.cross_entropy(
    logits.view(-1, logits.size(-1)), # [B*T, V]
    labels.view(-1).long(), # [B*T]
    ignore_index=-100,
)
loss.backward()
optimizer.step()
```

✓ 1.1s

Colab example: https://colab.research.google.com/drive/1iw34YAFjZcO57j8ivJD3-J837JibhShU

LLM fine-tuning in PyTorch

We can also use a simple but feature-complete training and evaluation interface through *Trainer()*. First, we need to change the data into the Dataset version.

```python
from transformers import DataCollatorForLanguageModeling
from datasets import Dataset

texts = [
    "Hello, how are you?",
    "What is your name?",
    "Tell me a joke."
]
dataset = Dataset.from_dict({"text": texts})
```

```python
def preprocess(examples):
    out = tokenizer(
        examples["text"],
        padding="max_length",
        truncation=True,
        max_length=128,
    )
    out["labels"] = out["input_ids"].copy()
    return out

tokenized_dataset = dataset.map(preprocess, remove_columns=["text"])
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer, mlm=False)
```

✓ 0.1s

```
Map: 100%|██████████| 3/3 [00:00<00:00, 608.90 examples/s]
```

LLM fine-tuning in PyTorch

Then we initialize the Trainer and use *trainer.train( )* to fine-tune

```
from transformers import Trainer, TrainingArguments
training_args = TrainingArguments(
    output_dir="./trainer_demo",
    num_train_epochs=5,
    per_device_train_batch_size=2,
    logging_steps=1,
    save_steps=20,
    report_to="none",
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset,
    data_collator=data_collator,
)

trainer.train()
```

✓ 43.6s

[10/10 00:42, Epoch 5/5]

Step	Training Loss
1	0.003100
2	0.009600
3	0.004000
4	0.014500
5	0.000800
6	0.000100
7	0.000100
8	0.000200
9	0.000100
10	0.000200

Open-source frameworks

- Machine learning frameworks:
  - PyTorch
  - TensorFlow
  - Hugging Face
  - Scikit-learn
  - R

- Research publications: Arxiv

- Open-source repositories: GitHub

CPU vs. GPU

- CPU: Central Processing Unit / Computer processor

- GPU: Graphics Processing Unit / GPU

- Cloud vs. On-premises vs. Edge

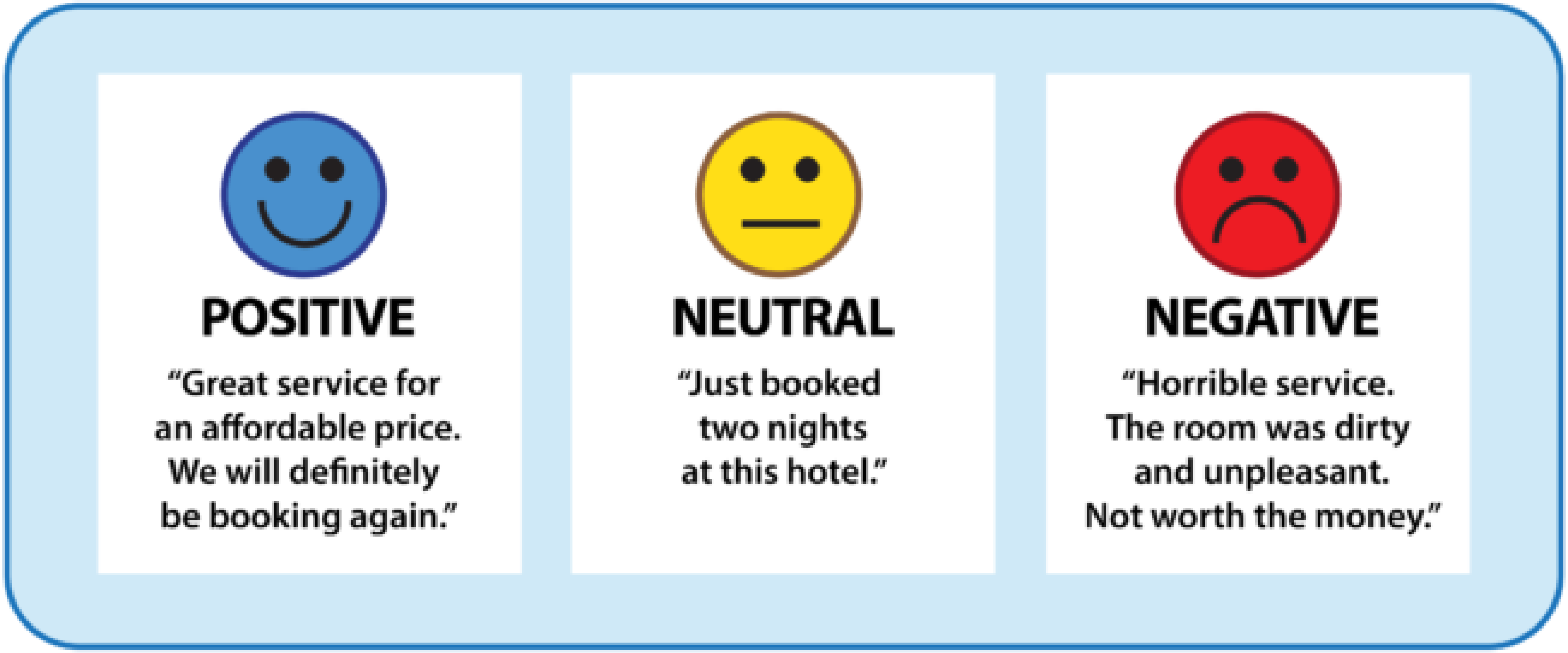
Lecture plan

- Language models
  - Training and adaptation with PyTorch implementations
  - **More about NLP tasks and an example**

Examples of NLP tasks

Natural language processing (NLP)

- **Sentiment analysis**: classifying the emotional intent of text
  - **Input:** a piece of text
  - **Output:** probability that the sentiment expressed is positive, negative, or neutral



Examples of NLP tasks

- **Toxicity classification**: the aim is not just to classify hostile intent but also to classify particular categories such as threats, insults, obscenities, and hatred
  - **Input:** text
  - **Output:** probability of each class of toxicity

- Useful in moderating online conversations by muting offensive comments, detecting hate speech, etc

Examples of NLP tasks

- **Machine translation**: automatic translation between different languages
  - **Input**: text in a specified source language
  - **Output**: text in a specified target language

- Google translate is one successful example

Examples of NLP tasks

- **Named entity recognition** (NER): extract entities in a piece of text into predefined categories such as personal names, organizations, locations, and quantities
  - **Input**: text
  - **Output**: various named entities along with their start and end positions

spaCy named entity recognition tagging of the first paragraph of Andrew Ng's Wikipedia page. "NORP" stands for nationalities or religious or political groups. Note that spaCy incorrectly labels "AI" as "GPE," for geopolitical entity.

Examples of NLP tasks

- **Spam detection**: a binary classification problem, where the purpose is to classify emails as either spam or not
  - **Input**: an email text along with various other subtexts like title and sender's name
  - **Output**: probability that the mail is spam

- Used by Gmail/Outlook to improve user experience

Examples of NLP tasks

- **Grammatical error correction**: encode grammatical rules to correct the grammar within text
  - This is a sequence-to-sequence task
  - Input: an ungrammatical sentence
  - Output: a correct sentence

- Grammarly and spell-checkers in word-processing systems are examples of such systems

Examples

- **Topic modeling**: an unsupervised text mining task that takes a corpus of documents and discovers abstract topics within that corpus
  - **Input**: a collection of documents
  - **Output**: a list of topics that defines words for each topic as well as assignment proportions of each topic in a document

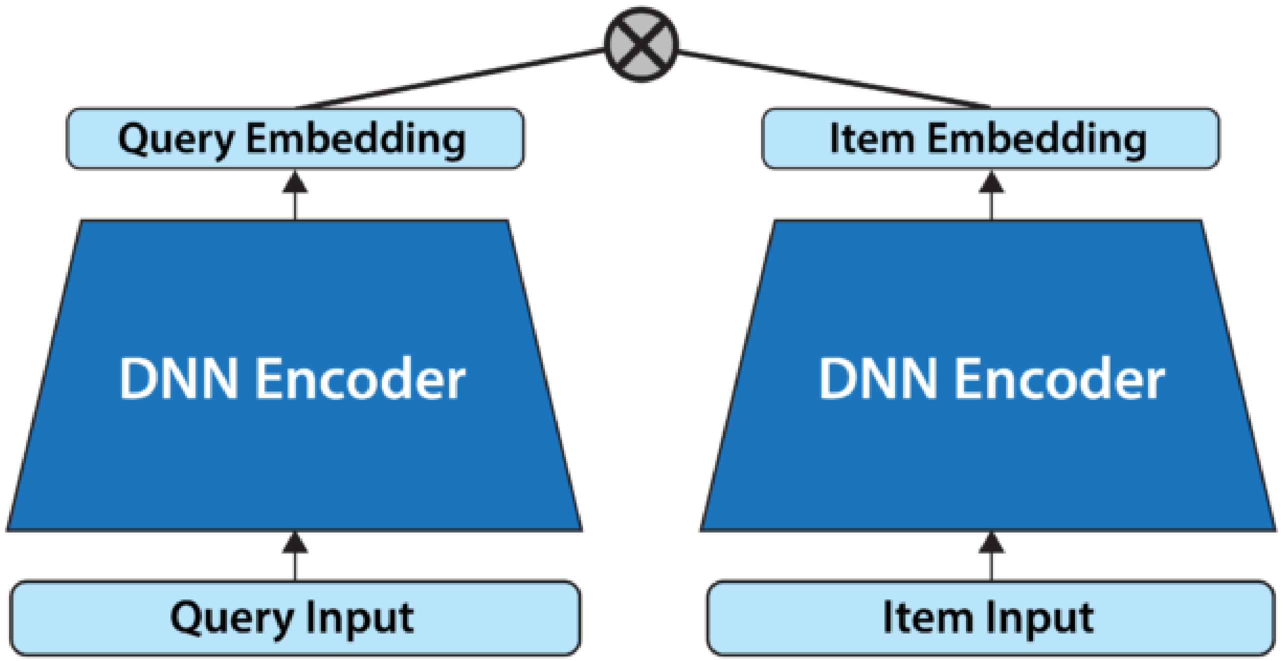
- Use cases in helping lawyers find evidence in legal documents (link)

Examples

- **Text generation** / **natural language generation** (NLG): produces text that's similar to human-written text
  - Can be fine-tuned to produce text in different genres and formats, including tweets, blogs, and even computer code
  - Autocomplete predicts what word comes next (used in chat apps like WhatsApp)
  - Chatbots automate one side of a conversation while a human conversant generally supplies the other side: database query, conversation generation

Examples

- **Information retrieval**: finds documents that are most relevant to a query
  - Every search and recommendation system faces this problem
  - Often need to retrieve from millions of documents (now enhanced with multimodal search)



A two-tower network creates a representation of an input query and a group of documents (or items) through two separate networks. Then it compares the representation of the query with that of the documents to find documents that are most relevant to the query.

Examples

- **Summarization**: shortening text to highlight the most relevant information
  - **Extractive summarization**: extracts the most important sentences (e.g., by scoring each sentence) from a long text and combining them to form a summary
  - **Abstractive summarization**: produces a summary by paraphrasing, usually modeled as a sequence-to-sequence task

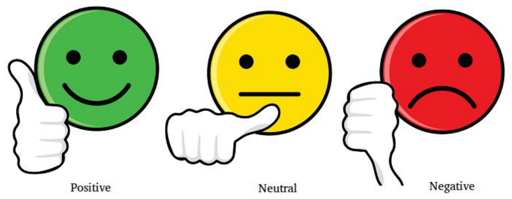
Examples

- **Question answering**: answering questions posed by humans in a natural language
  - **Multiple choice**: composed of a question and a set of possible answers
  - **Open domain**: provide answers to questions in natural language without any options provided, often by querying a large number of texts

- IBM Watson: https://www.ibm.com/history/watson-jeopardy

Sentiment analysis with logistic regression

- **Sentiment prediction:** "A very busy, but rewarding first week of the fall semester." The sentence consists of a list of eleven words: $\{A_1, A_2, \ldots, A_{11}\}$



- **Input**: a list of (text, label) pairs
- **Output**: a classifier that, given an unseen text, produces the probability corresponding each label

Naïve Bayes

- Prediction rule: Choose the most likely hypothesis given the list of words
  - Hypothesis y is Positive, Neural, or Negative
  - Use Bayes rule: get *likelihood* and *prior*

$$\arg\max_{y} \Pr(y \mid A_1, A_2, \dots, A_n) = \arg\max_{y} \frac{\Pr(A_1, A_2, \dots, A_n | y) \cdot \Pr(y)}{\Pr(A_1, \dots, A_n)}$$

- Naïve Bayes assumes conditional independence: Prior knowledge with a single word is easier to obtain

$$\Pr(A_1, A_2, \dots, A_n | y) = \Pr(A_1 | y) \cdot \Pr(A_2 | y) \cdot \dots \cdot \Pr(A_n | y)$$

Training the naïve Bayes classifier

- Apply logarithm to the above loss
- Now, estimate the condition probability given one hypothesis:

$$\Pr(A_i|y) = \frac{count(\text{“}A_i\text{”}, y)}{count(w, y)}$$