GPT3, InstructGPT, PyTorch

Precept 6 (3/20)

Slides from COS597R: Deep Dive into LLMs

Encoder vs Decoder models

- **One pre-trained model for all tasks**
  - BERT (Devlin et al., 2018), RoBERTa (Liu et al., 2019) — encoder models
  - T5 (Raffel et al., 2019), BART (Lewis et al., 2019) — encoder-decoder models
  - GPT-1 (Radford et al., 2018), GPT-2 (Radford et al., 2019) — decoder models

- All based on **Transformers**
- They mainly differ in the pre-training objectives (slight difference in fine-tuning)
- Model sizes and pre-training data are also different!


The Annotated Transformer

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

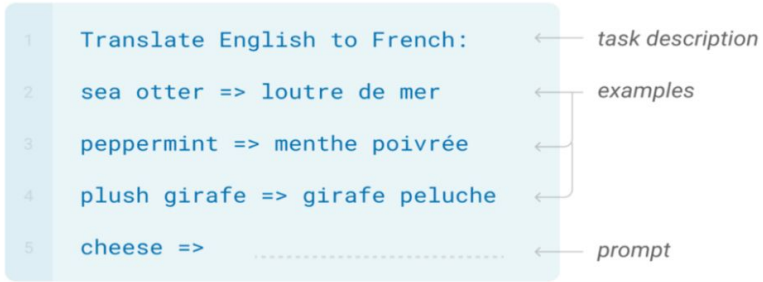
Illia Polosukhin* ‡ illia.polosukhin@gmail.com

- *v2022: Austin Huang, Suraj Subramanian, Jonathan Sum, Khalid Almubarak, and Stella Biderman.*
- *Original: Sasha Rush.*

GPT3: Main Contributions

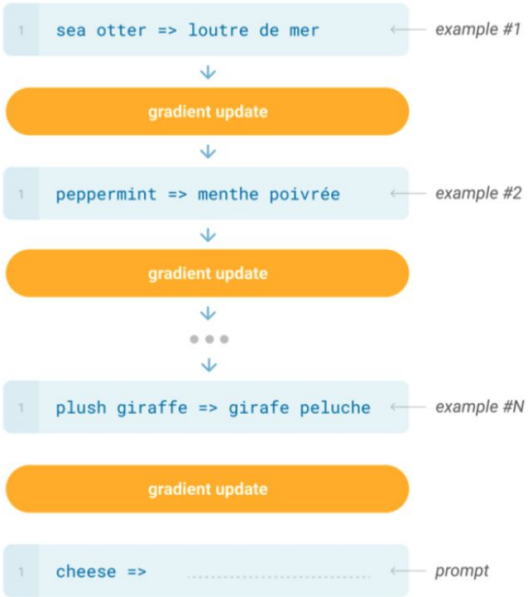
- An autoregressive language model of 175B parameters, 10x larger than any previous LMs
- Introduced the concept of “in-context learning”, and showed competitive performance

In-context learning: you can perform a task from only **a few examples** or **simple instructions** without any gradient updates or fine-tuning!



Fine-tuning

The model is trained via repeated gradient updates using a large corpus of example tasks.

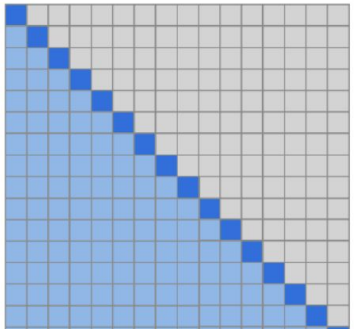
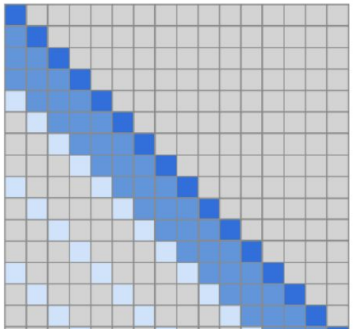
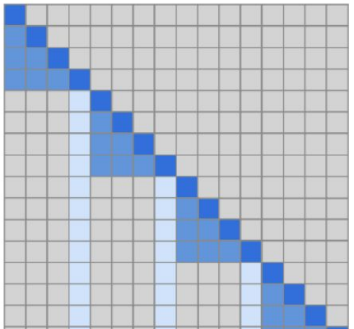


GPT3: Overview

- GPT-3 is a Transformer decoder only trained on large amounts of unlabeled text
- Training objective: next-token prediction

$$L_1(\mathcal{U}) = \sum_i \log P(u_i | u_{i-k}, \ldots, u_{i-1}; \Theta)$$

- Model architecture the same as GPT-2, including modified initialization, pre-normalization
  - Except that “we use alternating dense and locally banded sparse attention patterns in the layers of the Transformer”

GPT3: Overview

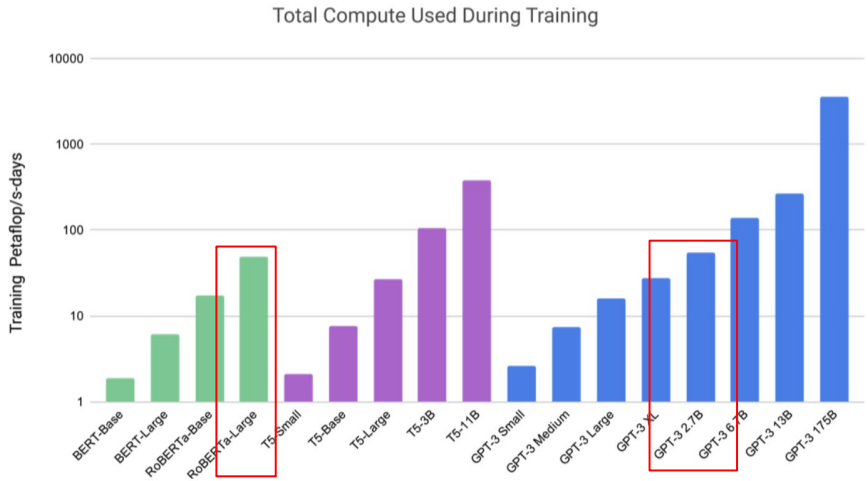
- GPT-3 is a Transformer decoder only trained on large amounts of unlabeled text
- All models were trained on 300B tokens

Model Name	n_{params}	n_{layers}	d_{model}	n_{heads}	d_{head}	Batch Size	Learning Rate
GPT-3 Small	125M	12	768	12	64	0.5M	6.0×10^{-4}
GPT-3 Medium	350M	24	1024	16	64	0.5M	3.0×10^{-4}
GPT-3 Large	760M	24	1536	16	96	0.5M	2.5×10^{-4}
GPT-3 XL	1.3B	24	2048	24	128	1M	2.0×10^{-4}
GPT-3 2.7B	2.7B	32	2560	32	80	1M	1.6×10^{-4}
GPT-3 6.7B	6.7B	32	4096	32	128	2M	1.2×10^{-4}
GPT-3 13B	13.0B	40	5140	40	128	2M	1.0×10^{-4}
GPT-3 175B or “GPT-3”	175.0B	96	12288	96	128	3.2M	0.6×10^{-4}

  - Larger models typically use a larger batch size but require a smaller learning rate
  - **Context window size = 2048**
  - Use a lot of “model parallelism” during training
  - Use Adam optimizer $\beta_1 = 0.9, \beta_2 = 0.95, \text{and } \epsilon = 10^{-8}$

GPT3: Compute

Model	Total train compute (PF-days)	Total train compute (flops)	Params (M)	Training tokens (billions)
T5-Small	2.08E+00	1.80E+20	60	1,000
T5-Base	7.64E+00	6.60E+20	220	1,000
T5-Large	2.67E+01	2.31E+21	770	1,000
T5-3B	1.04E+02	9.00E+21	3,000	1,000
T5-11B	3.82E+02	3.30E+22	11,000	1,000
BERT-Base	1.89E+00	1.64E+20	109	250
BERT-Large	6.16E+00	5.33E+20	355	250
RoBERTa-Base	1.74E+01	1.50E+21	125	2,000
RoBERTa-Large	4.93E+01	4.26E+21	355	2,000
GPT-3 Small	2.60E+00	2.25E+20	125	300
GPT-3 Medium	7.42E+00	6.41E+20	356	300
GPT-3 Large	1.58E+01	1.37E+21	760	300
GPT-3 XL	2.75E+01	2.38E+21	1,320	300
GPT-3 2.7B	5.52E+01	4.77E+21	2,650	300
GPT-3 6.7B	1.39E+02	1.20E+22	6,660	300
GPT-3 13B	2.68E+02	2.31E+22	12,850	300
GPT-3 175B	3.64E+03	3.14E+23	174,600	300



Why few-shot learning?

- Collecting large supervised training sets is expensive

Corpus	|Train|	|Test|	Task	Metrics	Domain
			Single-Sentence Tasks		
CoLA	8.5k	**1k**	acceptability	Matthews corr.	misc.
SST-2	67k	1.8k	sentiment	acc.	movie reviews
			Similarity and Paraphrase Tasks		
MRPC	3.7k	1.7k	paraphrase	acc./F1	news
STS-B	7k	1.4k	sentence similarity	Pearson/Spearman corr.	misc.
QQP	364k	**391k**	paraphrase	acc./F1	social QA questions
			Inference Tasks		
MNLI	393k	**20k**	NLI	matched acc./mismatched acc.	misc.
QNLI	105k	5.4k	QA/NLI	acc.	Wikipedia
RTE	2.5k	3k	NLI	acc.	news, Wikipedia
WNLI	634	**146**	coreference/NLI	acc.	fiction books

GLUE (Devlin et al. , 2018)

- Humans do not require large supervised datasets to learn most language tasks
- It allows humans to seamlessly **mix together** or **switch** between many tasks and tasks when interacting with NLP systems
  - Fluidity
  - Generality

GPT3: Approach

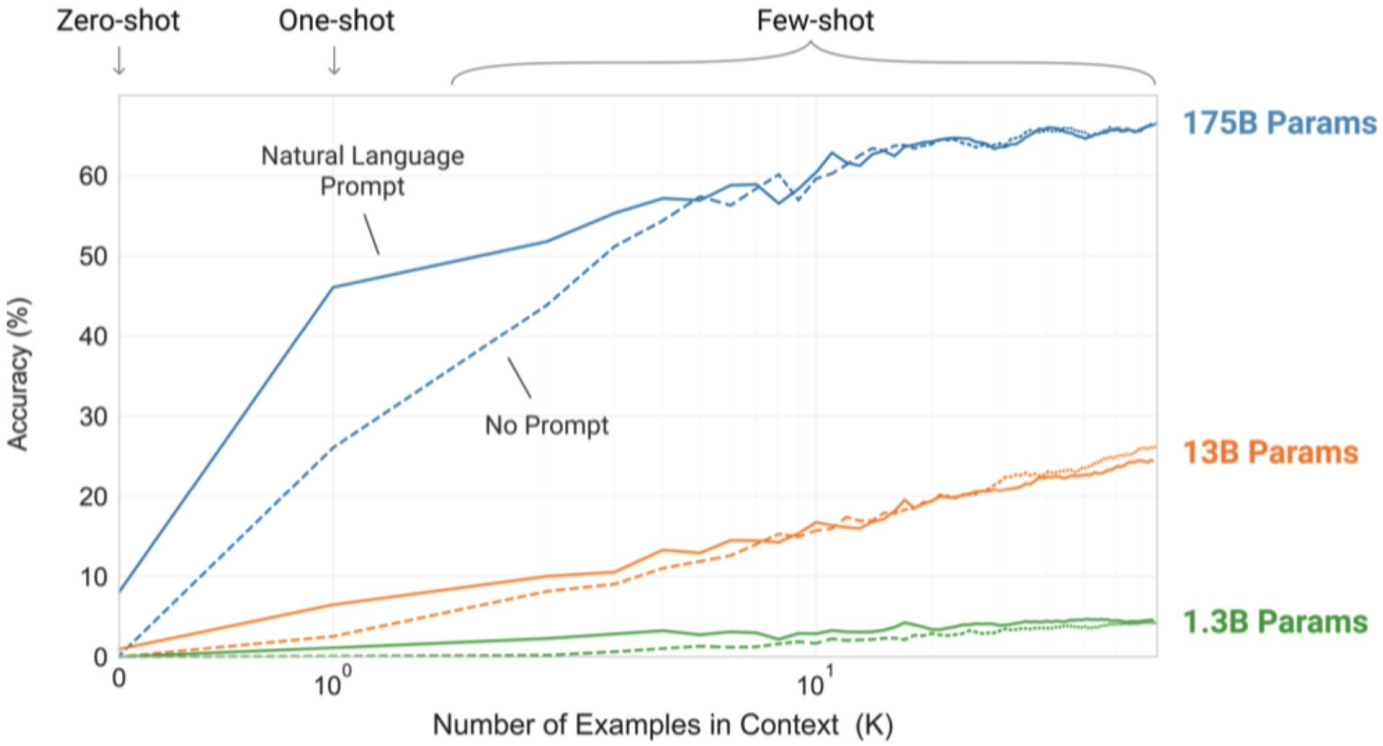
- **Few-shot:** a few demonstrations are prepended in the context (no weights updated allowed)
  - The demonstrations are randomly sampled from training set
  - K: typically 10-100, depending on how many examples can fit in context (2048)
  - Not always “the larger K, the better” => use a development set to decide K
  - Optionally add a natural language prompt
- **One-shot:** special case when K = 1.

  “it most closely matches the way in which some tasks are communicated to humans”

  “it is sometimes difficult to communicate the content or format of a task if no examples are given”
- **Zero-shot:** avoidance of spurious correlation, “unfairly hard”

  “at least some settings zero-shot is closest to how humans perform tasks”

Few-shot Learning



Chinchilla Scaling Law

"Compute optimal" : Best heldout cross-entropy given total FLOPs budget

- No constraints on # of GPUs and # Tokens
- Ignores communication latencies

$$N_{opt}(C), D_{opt}(C) = \underset{N,D \text{ s.t. } \mathrm{FLOPs}(N,D)=C}{\operatorname{argmin}} L(N, D).$$

Caveat: Minimization only over architectures, training, and datasets that were popular in '22

Experiments: 400 models, sizes 70M to 16B

Dataset size 5B to 500B

(other hyper-parameters such as batch size, dimension, etc taken from earlier studies)

Chinchilla Scaling Law

Parameters	FLOPs	FLOPs (in *Gopher* unit)	Tokens
400 Million	1.92e+19	1/29,968	8.0 Billion
1 Billion	1.21e+20	1/4,761	20.2 Billion
10 Billion	1.23e+22	1/46	205.1 Billion
67 Billion	5.76e+23	1	1.5 Trillion
175 Billion	3.85e+24	6.7	3.7 Trillion
280 Billion	9.90e+24	17.2	5.9 Trillion
520 Billion	3.43e+25	59.5	11.0 Trillion
1 Trillion	1.27e+26	221.3	21.2 Trillion
10 Trillion	1.30e+28	22515.9	216.2 Trillion

"Chinchilla Scaling Law"

($D \approx 20N$ is compute-optimal choice)

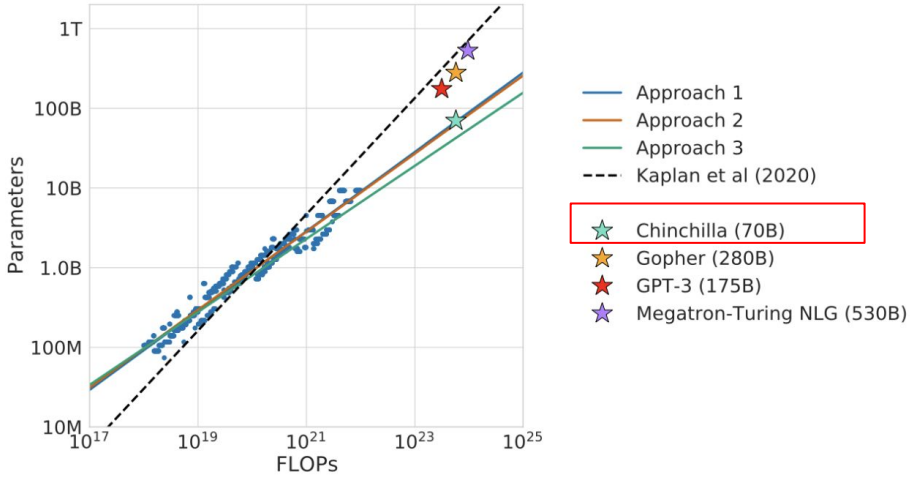


Figure 1 | **Overlaid predictions.** We overlay the predictions from our three different approaches, along with projections from Kaplan et al. (2020). We find that all three methods predict that current large models should be substantially smaller and therefore trained much longer than is currently done. In Figure A3, we show the results with the predicted optimal tokens plotted against the optimal number of parameters for fixed FLOP budgets. ***Chinchilla* outperforms *Gopher* and the other large models (see Section 4.2).**

Instruction Tuning: Motivation

- Language modeling objective is **misaligned**
  - "Predicting the next token on a web page from the internet" is different from "follow the user's instructions helpfully and safely"
- What are user's intention?
  - Explicit: instruction following
  - Implicit: stay truthful, not being biased, toxic or otherwise harmful
- The three H principle:

Helpful

Honest

Harmless

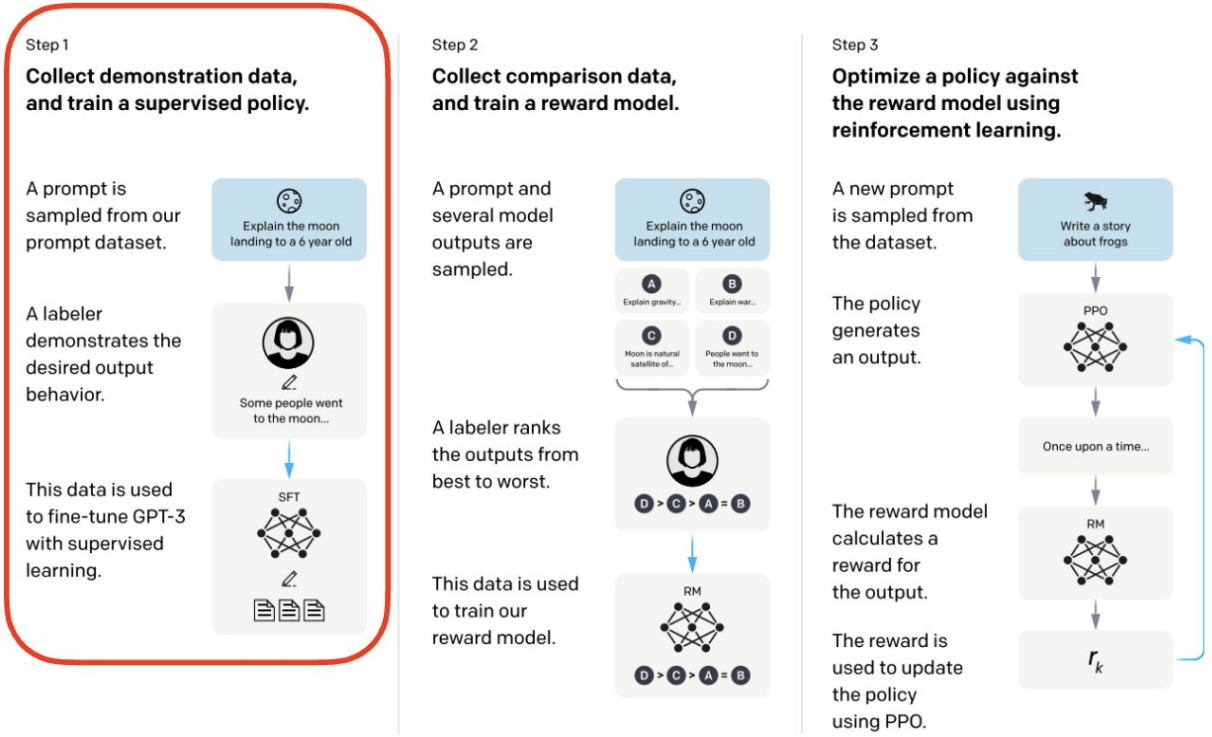
- **Helpful**: we want the model to solve the tasks for us
- **Honest**: we want the model to give us accurate information and express uncertainty when they don't know the answer
- **Harmless**: we don't want models to cause any harm to people or environment.

- Related keywords: post-training, instruction fine-tuning, supervised fine-tuning (SFT)
- Instruction tuning = supervised fine-tuning on instruction data

Instruction Tuning

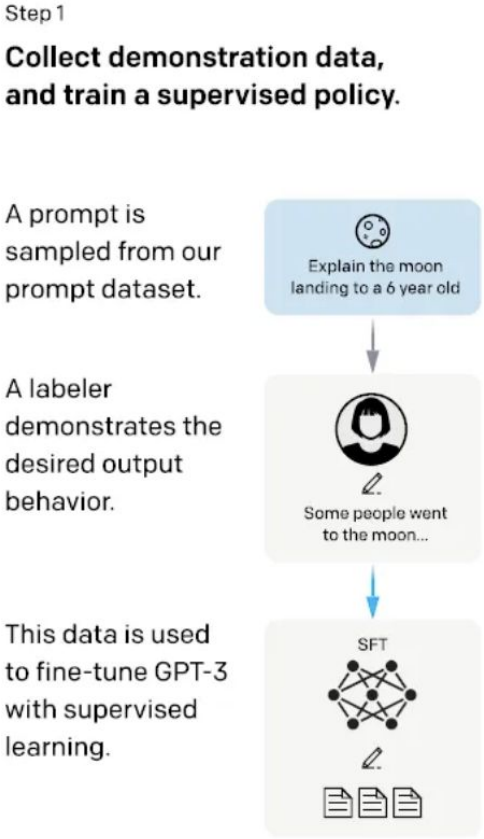
- **First wave (2021-2022):** instruction tuning on massive (NLP) tasks can generalize to unseen tasks
    - Cross-task generalization
    - Limited to standard tasks - easier to evaluate!
- **Second wave (2022-??):** “open-ended” instruction tuning, popularized by InstructGPT/ChatGPT
    - Anything can be a task - infinite possibilities!
    - Evaluation is hard: human evaluation, LLM as judge..

InstructGPT: Overview



InstructGPT (Ouyang et al., 2022)

InstructGPT: SFT

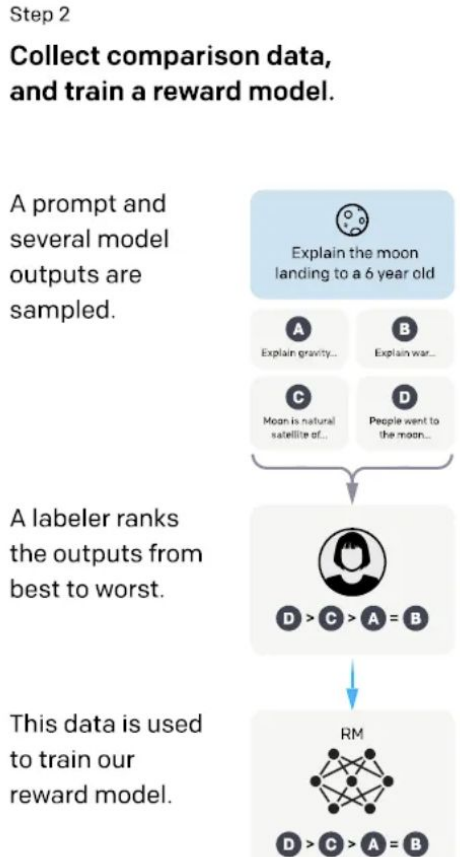


- 13k prompts are written by labelers/collected from API
- Responses are written by labelers
- Training on SFT data for 16 epochs

Use-case	Prompt
Brainstorming	List five ideas for how to regain enthusiasm for my career
Generation	Write a short story where a bear goes to the beach, makes friends with a seal, and then returns home.
Rewrite	This is the summary of a Broadway play: """ {summary} """ This is the outline of the commercial for that play: """

SFT Data		
split	source	size
train	labeler	11,295
train	customer	1,430
valid	labeler	1,550
valid	customer	103

InstructGPT: Reward Modeling



- 33k prompts are written by labelers/collected from API
- Labelers need to rank K responses (sampled from model; K=4~9)

> "most of our comparison data comes from our supervised policies, with some coming from our PPO policies"

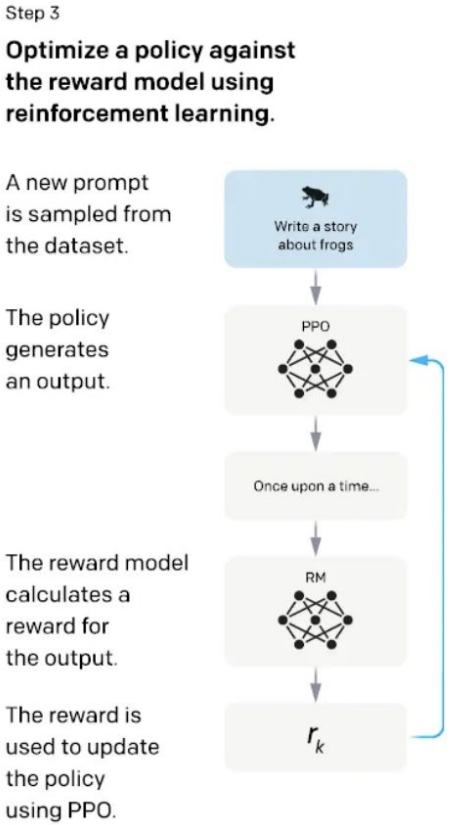
- The RM is only 6B parameters: $R : (x, y) \rightarrow \mathbb{R}$

$$\text{loss}(\theta) = -\frac{1}{\binom{K}{2}} E_{(x,y_w,y_l)\sim D}\left[\log\left(\sigma\left(r_\theta(x, y_w) - r_\theta(x, y_l)\right)\right)\right]$$

RM Data

split	source	size
train	labeler	6,623
train	customer	26,584
valid	labeler	3,488
valid	customer	14,399

InstructGPT: RL



- Key idea: fine-tuning supervised policy to optimize reward (output of the RM) using PPO
- 31k prompts only collected from API

$$\text{objective}\,(\phi) = E_{(x,y)\sim D_{\pi_\phi^{\text{RL}}}}\left[r_\theta(x,y)\right]$$

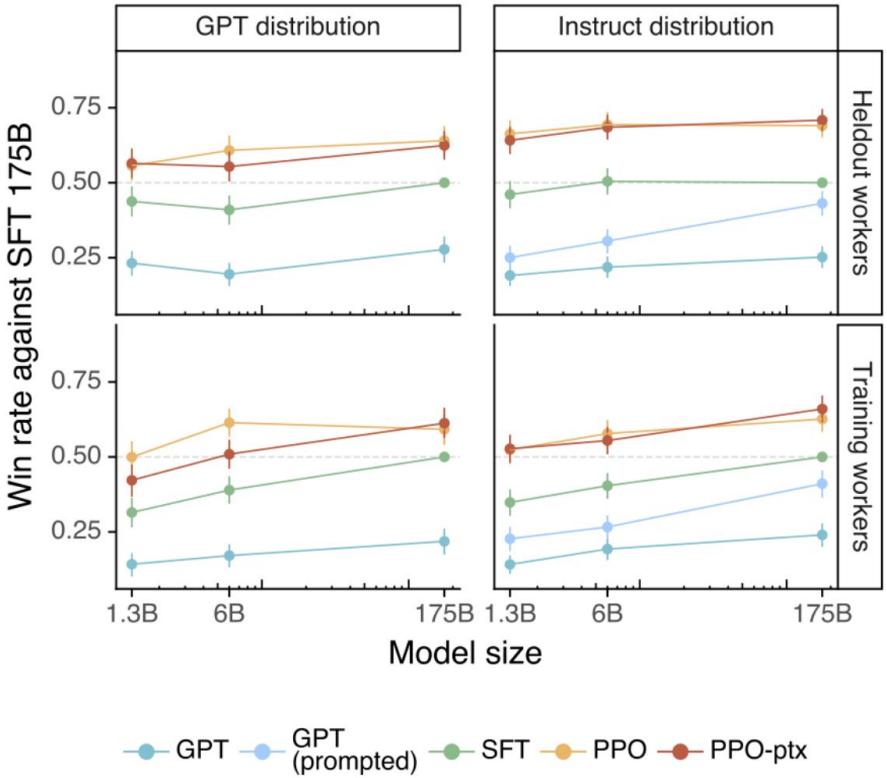
- Tweak #1: add a per-token KL penalty from the SFT model at each token to mitigate overoptimization of the reward model
- Tweak #2: add pre-training loss to "fix the performance regressions on public NLP datasets" (**PPO-ptx**)

$$\text{objective}\,(\phi) = E_{(x,y)\sim D_{\pi_\phi^{\text{RL}}}}\left[r_\theta(x,y) - \beta\log\left(\pi_\phi^{\text{RL}}(y\mid x)/\pi^{\text{SFT}}(y\mid x)\right)\right] + \gamma E_{x\sim D_{\text{pretrain}}}\left[\log(\pi_\phi^{\text{RL}}(x))\right]$$

PPO Data

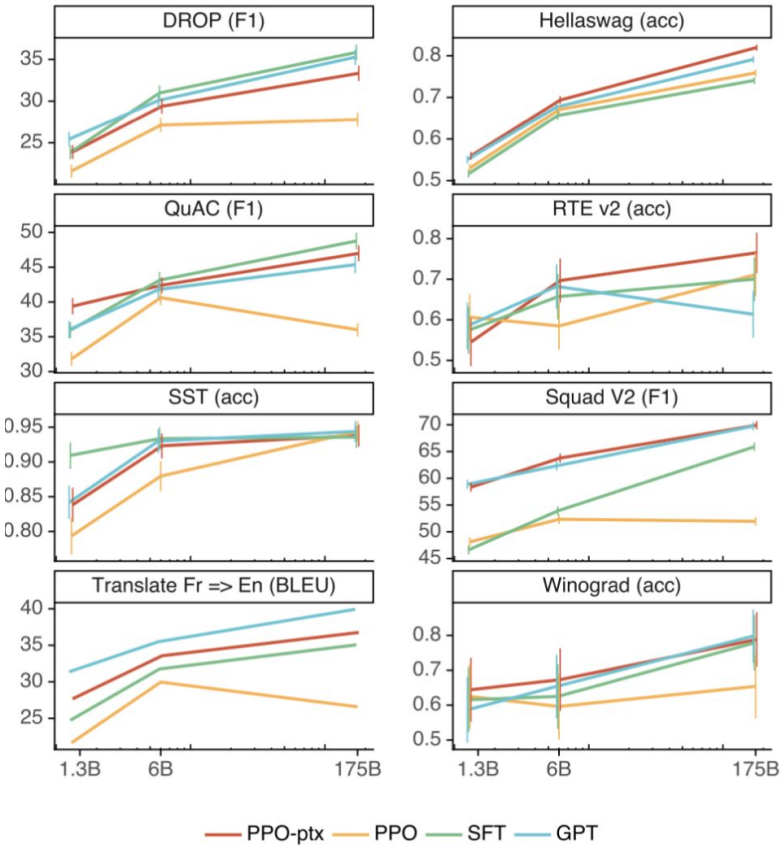
split	source	size
train	customer	31,144
valid	customer	16,185

InstructGPT vs GPT3



- 1.3B PPO model is more preferred to 175 B SFT/GPT

InstructGPT



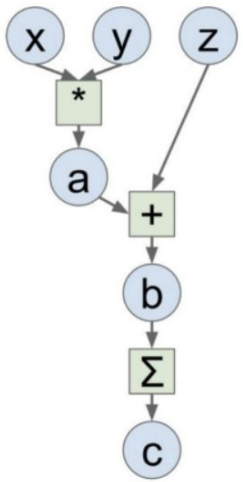
- "Alignment tax"
- PPO-ppx mitigates performance regression on most tasks

Other results:

- Improvements on TruthfulQA
- Small improvements on RealToxicityPrompts
- No improvements on bias evaluation

PyTorch Preview

- API is very clean and code is very readable
- No need to compute gradients, just use `.backward()`!



Computational Graph

```numpy
import numpy as np
np.random.seed(0)

N, D = 3, 4

x = np.random.randn(N, D)
y = np.random.randn(N, D)
z = np.random.randn(N, D)

a = x * y
b = a + z
c = np.sum(b)

grad_c = 1.0
grad_b = grad_c * np.ones((N, D))
grad_a = grad_b.copy()
grad_z = grad_b.copy()
grad_x = grad_a * y
grad_y = grad_a * x
```

NumPy

```pytorch
import torch
torch.manual_seed(0)

N, D = 3, 4

x = torch.rand((N, D), requires_grad=True)
y = torch.rand((N, D), requires_grad=True)
z = torch.rand((N, D), requires_grad=True)

a = x * y
b = a + z
c = torch.sum(b)

c.backward()
```

PyTorch

Slides from COS484 (Sp2024)

torch.Tensor: Introduction

- Exactly like NumPy arrays, but can be run efficiently on GPUs
- Supports the same operations like indexing, slicing, reshaping, transpose, cross product, matrix product, element-wise multiplication, …

```numpy
import numpy
# create a tensor
new_numpy = numpy.array([[1, 2], [3, 4]])
# create a 2 x 3 tensor with random values
empty_numpy = numpy.random.rand(2,3)
# create a 2 x 3 tensor with random values between -1 and 1
uniform_numpy = numpy.reshape(np.random.uniform(-1,1,6),[2,3])
# create a 2 x 3 tensor with random values from a uniform distribution on the interval [0, 1)
rand_numpy = numpy.random.rand(2, 3)
# create a 2 x 3 tensor of zeros
zero_numpy = numpy.zeros([2, 3])
```

NumPy

```pytorch
import torch
# create a tensor
new_tensor = torch.Tensor([[1, 2], [3, 4]])
# create a 2 x 3 tensor with random values
empty_tensor = torch.Tensor(2, 3)
# create a 2 x 3 tensor with random values between -1and 1
uniform_tensor = torch.Tensor(2, 3).uniform_(-1, 1)
# create a 2 x 3 tensor with random values from a uniform distribution on the interval
rand_tensor = torch.rand(2, 3)
# create a 2 x 3 tensor of zeros
zero_tensor = torch.zeros(2, 3)
```

PyTorch

torch.Tensor: Gradients

`requires_grad` - Makes this a trainable parameter

- *False* by default
- Turn on:

```
t.requires_grad_()
t = torch.randn(1, requires_grad=True)
```

- Tensor value = `t.data`
- Gradient value = `t.grad`
- History of autograd operations = `t.grad_fn`

```
1  import torch
2
3  N, D = 3, 4
4
5  x = torch.rand((N, D),requires_grad=True)
6  y = torch.rand((N, D),requires_grad=True)
7  z = torch.rand((N, D),requires_grad=True)
8
9  a = x * y
10 b = a + z
11 c=torch.sum(b)
12
13 c.backward()
14
15 print(c.grad_fn)
16 print(x.data)
17 print(x.grad)
```

```
<SumBackward0 object at 0x7fd0cb970cc0>
tensor([[0.4118, 0.2576, 0.3470, 0.0240],
        [0.7797, 0.1519, 0.7513, 0.7269],
        [0.8572, 0.1165, 0.8596, 0.2636]])
tensor([[0.6855, 0.9696, 0.4295, 0.4961],
        [0.3849, 0.0825, 0.7400, 0.0036],
        [0.8104, 0.8741, 0.9729, 0.3821]])
```

Note: Functions that end with an underscore _ modify the tensor in-place!

Tensors: Devices

Check if a GPU is available:

`torch.cuda.is_available()`

Convert a `numpy.array` to `torch.Tensor`:

`torch.from_numpy(x_train)`

`# this returns a cpu tensor`

Convert back to `numpy`:

`t.numpy()`

Move tensor to a device:

`t.to('cuda')` or `t.to('cpu')`

Check tensor or array type:

`type(t)` or `t.type()`

```
import torch
import torch.optim as optim
import torch.nn as nn
from torchviz import make_dot

device = 'cuda' if torch.cuda.is_available() else 'cpu'

# Our data was in Numpy arrays, but we need to transform them into PyTorch's Tensors
# and then we send them to the chosen device
x_train_tensor = torch.from_numpy(x_train).float().to(device)
y_train_tensor = torch.from_numpy(y_train).float().to(device)

# Here we can see the difference - notice that .type() is more useful
# since it also tells us WHERE the tensor is (device)
print(type(x_train), type(x_train_tensor), x_train_tensor.type())
```

Autograd: Introduction

- Automatic differentiation package
- `t.backward()` calculates gradients along the computational graph
- Gradients are accumulated by default
  - Need to zero them out after each update
  - t.grad.zero_()
  - (Typically done with Optimizer!)

Autograd: Introduction

Automatic differentiation package

We can easily compute gradients with:

```
t.backward()  # computes along the computation graph
```

Gradients are accumulated by default

We can update the parameter weight using the gradient as:

```
w -= lr * w.grad
```

After updating, you need to reset the gradients by clearing their values:

```
w.grad_.zero()
```

Autograd: Manual Update Example

- Definition
- Forward pass
- Backward pass
- Weight update
- Reset grads

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    with torch.no_grad():
        a -= lr * a.grad
        b -= lr * b.grad

    a.grad.zero_()
    b.grad.zero_()

print(a, b)
```

Optimizer

The `torch.optim` library makes updating the parameters easier:

We can use different optimization algorithms, like Adam, SGD, RMSprop.

We compute the gradients like before: `t.backward()`

But now we use the optimizer to apply param updates: `optimizer.step()`

Zero gradients with: `optimizer.zero_grad()`

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)


# Defines a SGD optimizer to update the parameters
optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor
    error = y_train_tensor - yhat
    loss = (error ** 2).mean()

    loss.backward()

    optimizer.step()

    optimizer.zero_grad()
```

Loss Functions

- Use pre-implemented loss functions too!
- L1, MSE, Cross-Entropy, …

```
a = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)
b = torch.randn(1, requires_grad=True, dtype=torch.float, device=device)

# Defines a MSE loss function
loss_fn = nn.MSELoss(reduction='mean')

optimizer = optim.SGD([a, b], lr=lr)

for epoch in range(n_epochs):
    yhat = a + b * x_train_tensor

    loss = loss_fn(y_train_tensor, yhat)

    loss.backward()
    optimizer.step()
    optimizer.zero_grad()
```

Models: Introduction

All models inherit from the `nn.Module` class

Models need to implement at least these 2 methods:

1. `__init__(self)`

   Initializer will local variables and parameters

2. `forward(self, inputs)`

   Performs the actual prediction computation using parameters

Useful methods that we inherit from nn.Module

`model.state_dict()` reports the model parameters and values

`model.parameters()` reports trainable parameters

`model.train()` sets model to train mode so that gradients can update properly

`model.eval()` sets model to eval mode so that parameters don't change

Models: Example

Inherit from `nn.Module`

Model definition

```
class ManualLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # To make "a" and "b" real parameters of the model, we need to wrap them with nn.Parameter
        self.a = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))
        self.b = nn.Parameter(torch.randn(1, requires_grad=True, dtype=torch.float))

    def forward(self, x):
        # Computes the outputs / predictions
        return self.a + self.b * x
```

Prediction computation

Models: More Complex Components

- Can use pre-defined “layers”
  E.g. `nn.Linear(input_dim, output_dim)`
- Package layers together with `nn.Sequential`

  `Sequential(x, y, z) = z( y( x(input) ) )`
- Tons of pre-defined layers/models

  `nn.Embedding, nn.Linear, nn.LSTM, nn.ReLU, …`

```
class LayerLinearRegression(nn.Module):
    def __init__(self):
        super().__init__()
        # Instead of our custom parameters, we use a Linear layer with single input and single output
        self.seq_linear = nn.Sequential(nn.Linear(1, 2),nn.RELU(),nn.Linear(2,1))

    def forward(self, x):
        # Now it only takes a call to the layer to make predictions
        return self.seq_linear(x)
```

Datasets

Explicit parent class that models typical datasets

Kind of like a list of tuples where each entry is a (feature, label) pair

To make a custom dataset, you typically have 3 components:

1. `__init__(self)`
2. `__get_item__(self, idx)`
3. `__len__(self)`

Unless this is very large (can't fit in memory), you can probably just use `TensorDataset`

```python
from torch.utils.data import Dataset, TensorDataset

class CustomDataset(Dataset):
    def __init__(self, x_tensor, y_tensor):
        self.x = x_tensor
        self.y = y_tensor

    def __getitem__(self, index):
        return (self.x[index], self.y[index])

    def __len__(self):
        return len(self.x)

x_train_tensor = torch.from_numpy(x_train).float()
y_train_tensor = torch.from_numpy(y_train).float()

train_data = CustomDataset(x_train_tensor, y_train_tensor)
print(train_data[0])

train_data = TensorDataset(x_train_tensor, y_train_tensor)
print(train_data[0])
```

DataLoaders

Usually want to train in *batches*

DataLoaders take in a dataset (or TensorDataset), a desired mini-batch size, and whether to shuffle

The loader will behave like an iterator, so we can just use a normal for-loop!

Fetches a new mini-batch each iteration

`DataLoader`

```
from torch.utils.data import DataLoader

train_loader = DataLoader(dataset=train_data, batch_size=16, shuffle=True)
```

Train/Validation/Test Splits

`random_split( )`

Produces train, validation, and test splits from a Dataset

```
from torch.utils.data.dataset import random_split

x_tensor = torch.from_numpy(x).float()
y_tensor = torch.from_numpy(y).float()

dataset = TensorDataset(x_tensor, y_tensor)

train_dataset, val_dataset, test_dataset = random_split(dataset, [60, 20, 20])

train_loader = DataLoader(dataset=train_dataset, batch_size=16)
val_loader = DataLoader(dataset=val_dataset, batch_size=20)
test_loader = DataLoader(dataset=test_dataset, batch_size=20)
```

Saving / Loading Models: Inference only

Only save the `state_dict`

Save :

```
torch.save(model.state_dict(), PATH)
```

Load :

```
model = ModelClass(*args, **kwargs)
model.load_state_dict(torch.load(PATH))
model.eval()
```

Note: this only saves the model for inference! Cannot continue training this model

PATH typically ends in .pt

Saving / Loading models: Checkpointing

Save the `state_dict`, `optimizer_dict` and any other relevant training information (num iters, etc.)

Save:

```
torch.save({'model_state_dict': model.state_dict(),
            'optimizer_state_dict': optimizer.state_dict(),
            'Epoch': epoch,
            'Loss': loss, … })
```

Load:

```
model = ModelClass(*args, **kwargs)
optimizer = OptimizerClass(*args, **kwargs)
checkpoint = torch.load(PATH)
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
…
```

Evaluation

`torch.no_grad`

Don't compute/store gradients during evaluation (like validation set, test set, etc.)

`eval()`

Tell compiler which mode to run

Training
(want gradients)

Validation
(no gradients)

```
losses = []
val_losses = []
model = ManualLinearRegression().to(device)
loss_fn = nn.MSELoss(reduction='mean')
optimizer = optim.SGD(model.parameters(), lr=lr)

for epoch in range(n_epochs):
    for x_batch, y_batch in train_loader:
        model.train()

        x_batch = x_batch.to(device)
        y_batch = y_batch.to(device)
        yhat = model(x_train_tensor)

        loss = loss_fn(y_batch, yhat)
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()

        losses.append(loss)

    with torch.no_grad():
        for x_val, y_val in val_loader:
            x_val = x_val.to(device)
            y_val = y_val.to(device)

            model.eval()

            yhat = model(x_val)
            val_loss = loss_fn(y_val, yhat)
            val_losses.append(val_loss.item())
```

Additional Materials

Stanford CS224N PyTorch Tutorial

PyTorch documentation

Hugging Face Transformers Quickstart (hosts your favorite pretrained models)