Neural code generation: course overview

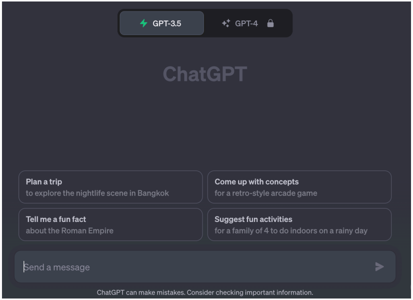
Instructor: Daniel Fried
TAs: Yiqing Xie and Atharva Naik

LTI 11-891, Carnegie Mellon University, Fall 2025
`https://cmu-codegen.github.io/f2025`

Sequence-to-sequence generation

General-purpose sequence generation

- Summarize documents
- Have a conversation
- ...



Code generation

Code generation

- Write software
- Automatically fix bugs
- Help prove that code is correct
- Tool for reasoning
- Interact with an environment
- ...

Code generation - applications

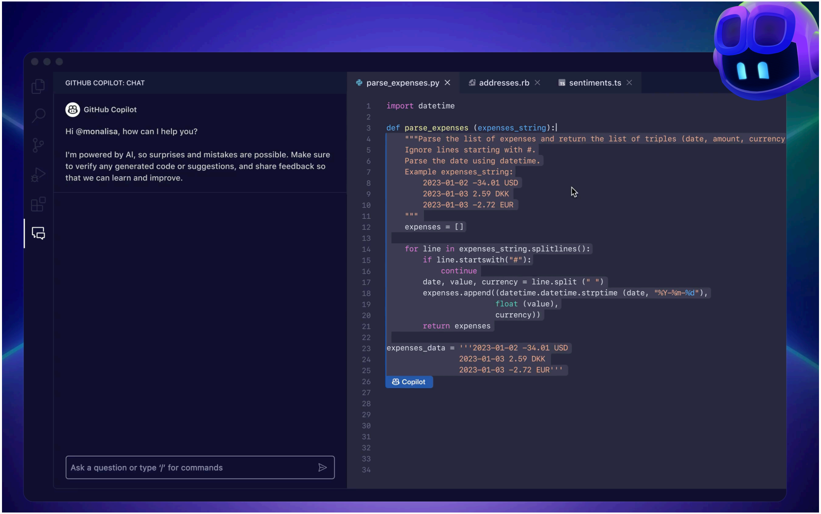


Figure 1: GitHub Copilot (12.2023)

Code generation - applications

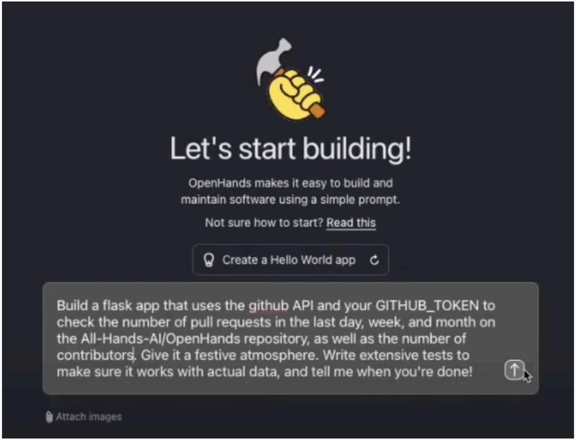


Figure 2: OpenHands Agent (08.2025)

Code generation - applications

RESEARCH

FunSearch: Making new discoveries in mathematical sciences using Large Language Models

14 DECEMBER 2023

Alhussein Fawzi and Bernardino Romera Paredes

Figure 3: FunSearch by Deepmind (12.2023)

Code generation - applications

```python
def priority(el: tuple[int, ...],
↪   n: int) -> float:
  score = n
  in_el = 0
  el_count = el.count(0)

  if el_count == 0:
    score += n ** 2
    if el[1] == el[-1]:
      score *= 1.5
    if el[2] == el[-2]:
      score *= 1.5
    if el[3] == el[-3]:
      score *= 1.5
  else:
    if el[1] == el[-1]:
      score *= 0.5
    if el[2] == el[-2]:
      score *= 0.5

  for e in el:
    if e == 0:
      if in_el == 0:
        score *= n * 0.5
      elif in_el == el_count - 1:
        score *= 0.5
      else:
        score *= n * 0.5 ** in_el
      in_el += 1
    else:
      score += 1

  if el[1] == el[-1]:
    score *= 1.5
  if el[2] == el[-2]:
    score *= 1.5

  return score
```

Figure 4: The function discovered by FunSearch that results in the largest known cap set (size 512) in 8 dimensions.

Code generation - applications

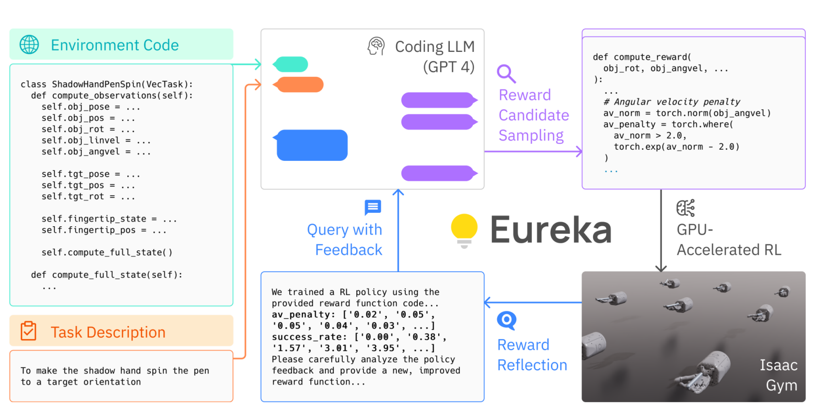


Figure 5: Eureka by NVIDIA Research (ICLR 2024)

Code generation – a brief history

Classical methods for program synthesis (specification $\rightarrow$ program)

Code generation – a brief history

Classical methods for program synthesis (specification $\rightarrow$ program)

- *Sketch* [Solar-Lezama 2008] :

```
#define LHS {| tmp | (l | nl).(h | t)(.next)? |}
#define LOC {| LHS | null |}
#define COMP {| LOC ( == | != ) LOC |}

list reverseEfficient(list l){
    list nl = new list();
    node tmp = null;
    bit c = COMP;
    while(c){
        repeat(??)
            if( COMP ){ LHS = LOC; }
        c = COMP;
    }
}
```

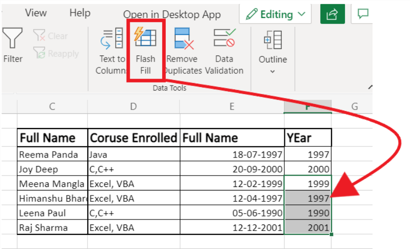
```
// test harness
void main(int n){
  if(n >= N){ n = N-1; }
  node[N] nodes = null;
  list l = newList();
  //Populate the list, and
  //write its elements
  //to the nodes array
  populateList(n, l, nodes);

  l = reverseSK(l);

  //Check that node i in
  //the reversed list is
  //equal to nodes[n-i-1]
  check(n, l, nodes);
}
```

- Specification: code with holes and test cases
- Output: fills in holes
- SAT-based search procedure

Code generation – a brief history

Classical methods for program synthesis (specification $\rightarrow$ program)

- *FlashFill* [Gulwani 2011] :



  - Specification: (input, output) examples
  - Output: Excel string transformation
  - Domain-specific language and exhaustive search

Neural code generation – a brief history

Early language models for code

- N-gram language models [Hindle et al 2012, Allamanis & Sutton 2013]

> *Programming languages, in theory, are complex, flexible and powerful, but, "natural" programs, the ones that real people actually write, are mostly simple and rather repetitive; thus they have usefully predictable statistical properties that can be captured in statistical language models and leveraged for software engineering tasks.*

Figure 6: Hindle et al 2012

Neural code generation – a brief history

Early language models for code

- N-gram language models [Hindle et al 2012, Allamanis & Sutton 2013]

$$p(a_4|a_1a_2a_3) = \frac{count(a_1a_2a_3a_4)}{count(a_1a_2a_3*)}$$

Figure 7: Hindle et al 2012

Neural code generation – a brief history

Early language models for code

- N-gram language models [Hindle et al 2012, Allamanis & Sutton 2013]

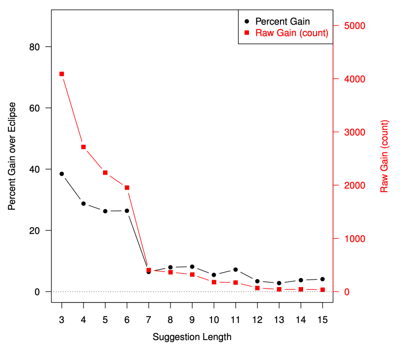


Figure 8: Hindle et al 2012; language-model suggestions in Eclipse

Neural code generation – a brief history

Early language models for code

- N-gram language models [Hindle et al 2012, Allamanis & Sutton 2013]

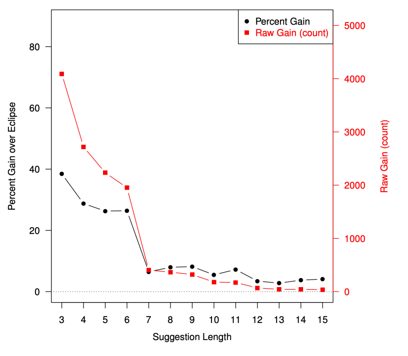


Figure 8: Hindle et al 2012; language-model suggestions in Eclipse

Restrictive n-gram model; limited generation capability

Neural code generation – a brief history

Early neural models for code

- Latent predictor network [Ling et al 2016] : seq2seq architecture for code generation

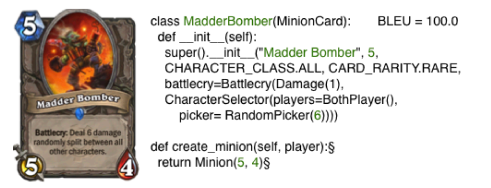


Figure 9: Generate code from a description of a card

Neural code generation – a brief history

Early neural models for code

- Abstract syntax network [Rabinovich, Stern, Klein 2017] : hierarchical neural architecture for code generation

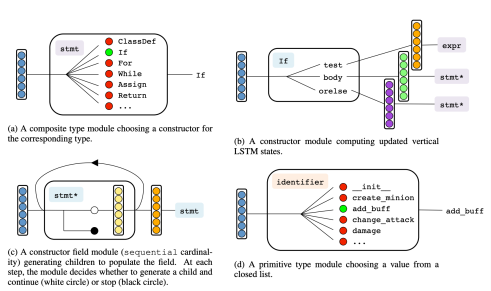


Figure 10: *Hierarchically* generate code from a description of a card

Early neural models for code

- Abstract syntax network [Rabinovich, Stern, Klein 2017] : hierarchical neural architecture for code generation

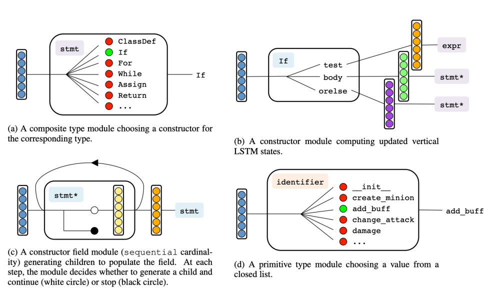


Figure 10: *Hierarchically* generate code from a description of a card

Specialized architectures, trained for a specific dataset

Neural code generation – a brief history

In 2021, GPT-3 had just come out. LLMs were starting to work.

Evaluating Large Language Models Trained on Code

Mark Chen [*1] **Jerry Tworek** [*1] **Heewoo Jun** [*1] **Qiming Yuan** [*1] **Henrique Ponde de Oliveira Pinto** [*1]
Jared Kaplan [*2] **Harri Edwards** [1] **Yuri Burda** [1] **Nicholas Joseph** [2] **Greg Brockman** [1] **Alex Ray** [1] **Raul Puri** [1]
Gretchen Krueger [1] **Michael Petrov** [1] **Heidy Khlaaf** [3] **Girish Sastry** [1] **Pamela Mishkin** [1] **Brooke Chan** [1]
Scott Gray [1] **Nick Ryder** [1] **Mikhail Pavlov** [1] **Alethea Power** [1] **Lukasz Kaiser** [1] **Mohammad Bavarian** [1]
Clemens Winter [1] **Philippe Tillet** [1] **Felipe Petroski Such** [1] **Dave Cummings** [1] **Matthias Plappert** [1]
Fotios Chantzis [1] **Elizabeth Barnes** [1] **Ariel Herbert-Voss** [1] **William Hebgen Guss** [1] **Alex Nichol** [1] **Alex Paino** [1]
Nikolas Tezak [1] **Jie Tang** [1] **Igor Babuschkin** [1] **Suchir Balaji** [1] **Shantanu Jain** [1] **William Saunders** [1]
Christopher Hesse [1] **Andrew N. Carr** [1] **Jan Leike** [1] **Josh Achiam** [1] **Vedant Misra** [1] **Evan Morikawa** [1]
Alec Radford [1] **Matthew Knight** [1] **Miles Brundage** [1] **Mira Murati** [1] **Katie Mayer** [1] **Peter Welinder** [1]
Bob McGrew [1] **Dario Amodei** [2] **Sam McCandlish** [2] **Ilya Sutskever** [1] **Wojciech Zaremba** [1]

Abstract

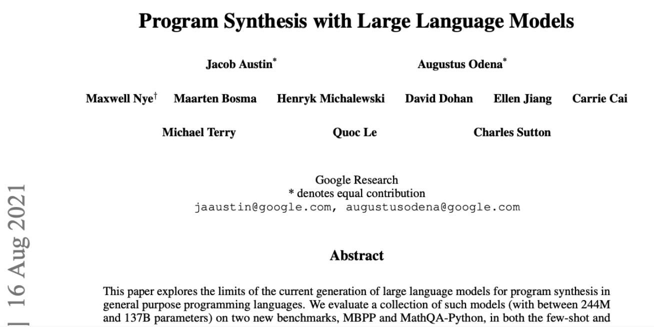
We introduce Codex, a GPT language model fine-tuned on publicly available code from GitHub, and study its Python code-writing capabilities.

1. Introduction

Scalable sequence prediction models (Graves, 2014; Vaswani et al., 2017; Child et al., 2019) have become a general-purpose method for generation and representation learning in many domains, including natural language processing (Mikolov et al., 2013; Sutskever et al., 2014; Dai &

Neural code generation – a brief history

In 2021, GPT-3 had just come out. LLMs were starting to work.



16 Aug 2021

Program Synthesis with Large Language Models

Jacob Austin* **Augustus Odena***

Maxwell Nye† **Maarten Bosma** **Henryk Michalewski** **David Dohan** **Ellen Jiang** **Carrie Cai**

Michael Terry **Quoc Le** **Charles Sutton**

Google Research
* denotes equal contribution
jaaustin@google.com, augustusodena@google.com

Abstract

This paper explores the limits of the current generation of large language models for program synthesis in general purpose programming languages. We evaluate a collection of such models (with between 244M and 137B parameters) on two new benchmarks, MBPP and MathQA-Python, in both the few-shot and

Neural code generation – a brief history

In 2021, GPT-3 had just come out. LLMs were starting to work.

- Language models
- + general purpose architecture
- + diverse data

Code generation with large language models (LLMs)

prompt

Write a function to find the smallest missing element in a sorted array. Your code should satisfy these tests:

```
assert smallest_missing([0, 1, 2, 3, 4, 5, 6], 0, 6) == 7
assert smallest_missing([0, 1, 2, 6, 9, 11, 15], 0, 6) == 3
assert smallest_missing([1, 2, 3, 4, 6, 9, 11, 15], 0, 7) == 0
```

model

```
def smallest_missing(arr, n, m):
    smallest = min(n, m)
    for i in range(n, m + 1):
        if arr[i] <= smallest:
            smallest += 1
    return smallest
```

Figure 11: Allows for natural language specifications [Austin et al 2021]

Neural code generation – a brief history

Code generation with large language models (LLMs)

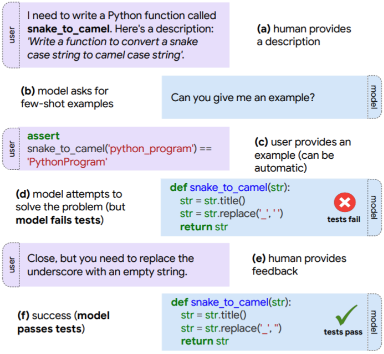


Figure 12: An overview of the "flow" of the human-model collaboration experiments. The human gives a description of the desired program and then guides the model toward the correct solution via dialog.

Figure 12: Key property: **flexibility** to perform many tasks [Austin et al 2021]

Neural code generation – a brief history

Code generation with large language models (LLMs)

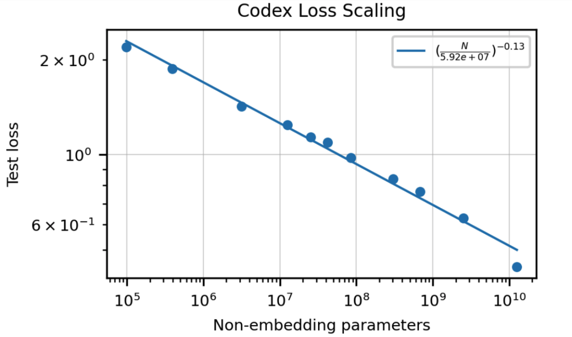


Figure 13: Key property: improves by **increasing scale** [Chen et al 2021]

Neural code generation – after Codex

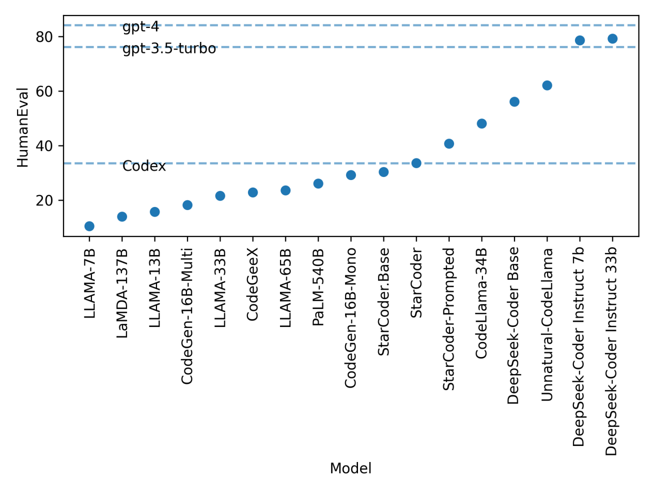


Figure 14: A lot of interest and development!

Why neural code generation?

- Many applications , with real-world impact
- Large amount of data
- Structured, compositional
- Combines informal (e.g., intent) and formal (e.g. testable code)
- Rich tooling (e.g., static analysis, compilers, ...)
- An agentive setting
- ...

Neural code generation

- Part I: *Foundations*
- Part II: *Frontiers*

Part I: Foundations

Principles of neural language models as applied to code.

- Model: $p_\theta(\mathrm{y}|\mathrm{x};\mathcal{D})$
    - $\mathrm{x}, \mathrm{y}:$ input, output sequences
    - $\theta:$ parameters (e.g., transformer)
    - $\mathcal{D}:$ dataset

Principles of neural language models as applied to code.

- Model: $p_\theta(\mathrm{y}|\mathrm{x}; \mathcal{D})$
  - x, y : input, output sequences
  - θ : parameters (e.g., transformer)
  - $\mathcal{D}$: dataset
- Learning:
  - $\arg\max_\theta \sum_{\mathrm{y}\in\mathcal{D}} \log p_\theta(\mathrm{y})$

Part I: Foundations

Principles of neural language models as applied to code.

- Model: $p_\theta(\mathsf{y}|\mathsf{x};\mathcal{D})$
  - $\mathsf{x},\mathsf{y}:$ input, output sequences
  - $\theta:$ parameters (e.g., transformer)
  - $\mathcal{D}:$ dataset
- Learning:
  - $\arg\max_\theta \sum_{\mathsf{y}\in\mathcal{D}} \log p_\theta(\mathsf{y})$

Part I: Foundations

Principles of neural language models as applied to code.

- Model: $p_\theta(\mathsf{y}|\mathsf{x};\mathcal{D})$
    - x, y : input, output sequences
    - θ : parameters (e.g., transformer)
    - $\mathcal{D}$: dataset
- Learning:
    - $\arg\max_\theta \sum_{\mathsf{y}\in\mathcal{D}} \log p_\theta(\mathsf{y})$
- Inference:
    - $\mathsf{y} = f(p_\theta(\cdot|x))$
    - f: e.g., sampling

Part I: Foundations

Principles of neural language models as applied to code.

- Model: $p_\theta(\mathsf{y}|\mathsf{x};\mathcal{D})$
  - x, y : input, output sequences
  - θ : parameters (e.g., transformer)
  - $\mathcal{D}$: dataset
- Learning:
  - $\arg\max_\theta \sum_{\mathsf{y}\in\mathcal{D}} \log p_\theta(\mathsf{y})$
- Inference:
  - $\mathsf{y} = f(p_\theta(\cdot|x))$
  - f: e.g., sampling
- Evaluation

Learning: how do we *train* language models for code generation?

- **Pretraining**: large-scale initial training based on *scaling laws* (8/28) and *code objectives* (9/2)

Part I: Foundations – *Learning*

Learning: how do we *train* language models for code generation?

- **Pretraining**: large-scale initial training based on *scaling laws* (8/28) and *code objectives* (9/2)
- **Finetuning**: specializing the model to follow instructions (9/4)

Learning: how do we *train* language models for code generation?

- **Pretraining**: large-scale initial training based on *scaling laws* (8/28) and *code objectives* (9/2)
- **Finetuning**: specializing the model to follow instructions (9/4)
- **Learning from feedback**: improving the model with feedback on its outputs, such as execution results (9/25)

Evaluation: how good is our neural code generator?

- Code metrics and benchmarks (9/9, 9/11)

Part I: Foundations – *Data*

Data: what data should we train with? (9/16, 9/18)

- Data for *pretraining* and *domain-adaptation*
- *Synthetic* data
- Impact of data *quality*

Inference: how do we *generate* code with a trained language model?

- Algorithms that leverage *execution*, *verification*, and *feedback* (9/23, 9/25, 10/2)

Neural code generation

- Part I: *Foundations*
  - Learning, Inference, Data, Evaluation
- Part II: *Frontiers*

- Part I: *Foundations*
- Part II: *Frontiers*

Part II: Frontiers – *Human Interaction and Software Engineering*

Code is **communicative** and code generators are **used by real people**

- **Pragmatic** aspects of code generation (11/13)
- **Programming with AI** (11/18)
- **Applications to Software Engineering** (10/7 and 10/9)

Part II: Frontiers – *Adaptability*

Real-world code is long, exists in repositories unseen during training, and evolves over time. How do we adapt to these conditions?

- Methods for **long-context** generation and **retrieval** in code (10/28 and 10/30)

Part II: Frontiers – *Agents*

LLM-based systems that can use tools to write, edit, and debug complex code.

- Agent benchmarks (10/21)
- Agent frameworks (10/23)
- Creating agent training data (11/11)

Code as a medium for **reasoning** and **control** (11/20)

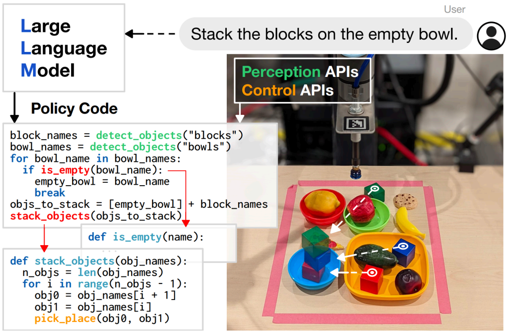


Fig. 1: Given examples (via few-shot prompting), robots can use code-writing large language models (LLMs) to translate natural language commands into robot policy code which process perception outputs, parameterize control primitives, recursively generate code for undefined functions, and generalize to new tasks.

Figure 15: Code generation for robotics

Some programming languages allow for **proving** that code is **correct**[1]

- **Neural theorem proving** (11/25)
  - Use LLMs to make it easier to verify things
  - Use verifiable code for mathematical reasoning

[1] E.g., Coq, Dafny, F*, Isabelle, Lean

Neural code generation

- Part I: *Foundations*
  - Learning, inference, data, evaluation
- Part II: *Frontiers*
  - Interaction, adaptability, reasoning, agents, formal methods

Course structure, projects, and logistics

Course structure

- 6-unit version of the course
  - Attend lectures (with pre- and post-assignments)
  - Attend discussions (with pre- and post-assignments)
  - Lead a discussion with a team (via a presentation)

Course structure

- 6-unit version of the course
    - Attend lectures (with pre- and post-assignments)
    - Attend discussions (with pre- and post-assignments)
    - Lead a discussion with a team (via a presentation)
- 12-unit version of the course: all the above, plus:
    - A high-quality research project, in teams of 3–4.
        - Two checkpoint reports
        - Two structured project hours
        - Final presentation
        - Final report

6-Unit course structure: discussions

In a student-led discussion, 4 students present a (set of) papers on a theme. Choose how much to focus on each paper, but cover the following topics:

- **Content:** motivation, setting, methods, findings. What was surprising?
- **Reviewer:** role-play a conference reviewer. Score the paper, and justify.
- **Future:** Brainstorm future work ideas for discussion.
- **Reproducibility:** What code and data would you use to dig deeper?

Use slides, but a main goal is to facilitate a discussion! Ask questions to the class.

6-Unit course structure: discussions

For presenters:

- Submit your slides before the day you present.
- We'll grade based on the presentation and slides.
- It's ok if you spark a long discussion and don't get through all slides.
- Present one time during the course, for 33% of the 6-unit grade, or 16% of the 12-unit grade.

6-Unit course structure: discussions

For presenters:

- Submit your slides before the day you present.
- We'll grade based on the presentation and slides.
- It's ok if you spark a long discussion and don't get through all slides.
- Present one time during the course, for 33% of the 6-unit grade, or 16% of the 12-unit grade.

Sign-ups:

- Sign-up spreadsheet coming after class.
- **Please sign up by Thursday end-of-day**. You can swap later if you find someone willing to.
- Extra credit (+2 out of 20 presentation points) for any team that presents on Thursday next week (9/4), on *finetuning for code*.

6-Unit course structure

On days you're not presenting (**both lectures and discussions**):

- Pre-assignment (33% of grade):
  - Short summary and ≥ 1 discussion questions for a paper.
  - Submit by 11:59pm the day **before** class.
  - 23 days, but we'll grade out of 20.

6-Unit course structure

On days you're not presenting (**both lectures and discussions**):

- Pre-assignment (33% of grade):
  - Short summary and ≥ 1 discussion questions for a paper.
  - Submit by 11:59pm the day **before** class.
  - 23 days, but we'll grade out of 20.
- Post-assignment (33% of grade):
  - 2-3 sentences on what you found interesting.
  - Submit by 11:59pm the day **of** class.
  - 23 days, but we'll grade out of 20.

12-Unit: Course project

- For students taking the class for 12 units, all of the 6 unit requirements, and also a course project.
- **Simulates doing a research project** on a topic related to the course.
- Teams of 3-4 members
- *Propose your own topic* or *pick a topic from our list*
- Ends in a report and presentation that should be in the style of a workshop paper or the first draft of a conference paper.

12-Unit: Project timeline (tentative)

- **Team formation**: Sept 9th

Your team will have a total of 5 late days which you can budget across any of the written reports (Report 1, Report 2, or the Final Report).

12-Unit: Project timeline (tentative)

- **Team formation**: Sept 9th
- **Project hours 1 (5%)**: Sept 30th
  Meet with course staff for 10 minutes, with a few slides.

Your team will have a total of 5 late days which you can budget across any of the written reports (Report 1, Report 2, or the Final Report).

12-Unit: Project timeline (tentative)

- **Team formation**: Sept 9th
- **Project hours 1 (5%)**: Sept 30th
  Meet with course staff for 10 minutes, with a few slides.
- **Report 1 (25%)**: Oct 10th
  Task proposal and data analysis; related work; baseline proposal.

Your team will have a total of 5 late days which you can budget across any of the written reports (Report 1, Report 2, or the Final Report).

12-Unit: Project timeline (tentative)

- **Team formation**: Sept 9th
- **Project hours 1 (5%)**: Sept 30th
  Meet with course staff for 10 minutes, with a few slides.
- **Report 1 (25%)**: Oct 10th
  Task proposal and data analysis; related work; baseline proposal.
- **Fall break**: Oct 14th and 16th – Take time off!

Your team will have a total of 5 late days which you can budget across any of the written reports (Report 1, Report 2, or the Final Report).

12-Unit: Project timeline (tentative)

- **Team formation**: Sept 9th
- **Project hours 1 (5%)**: Sept 30th
  Meet with course staff for 10 minutes, with a few slides.
- **Report 1 (25%)**: Oct 10th
  Task proposal and data analysis; related work; baseline proposal.
- **Fall break**: Oct 14th and 16th – Take time off!
- **Project hours 2 (5%)**: Nov 6th

Your team will have a total of 5 late days which you can budget across any of the written reports (Report 1, Report 2, or the Final Report).

12-Unit: Project timeline (tentative)

- **Team formation**: Sept 9th
- **Project hours 1 (5%)**: Sept 30th
  Meet with course staff for 10 minutes, with a few slides.
- **Report 1 (25%)**: Oct 10th
  Task proposal and data analysis; related work; baseline proposal.
- **Fall break**: Oct 14th and 16th – Take time off!
- **Project hours 2 (5%)**: Nov 6th
- **Report 2 (25%)**: Nov 11th
  Baseline results and analysis, and a technique proposal.

Your team will have a total of 5 late days which you can budget across any of the written reports (Report 1, Report 2, or the Final Report).

12-Unit: Project timeline (tentative)

- **Team formation**: Sept 9th
- **Project hours 1 (5%)**: Sept 30th
  Meet with course staff for 10 minutes, with a few slides.
- **Report 1 (25%)**: Oct 10th
  Task proposal and data analysis; related work; baseline proposal.
- **Fall break**: Oct 14th and 16th – Take time off!
- **Project hours 2 (5%)**: Nov 6th
- **Report 2 (25%)**: Nov 11th
  Baseline results and analysis, and a technique proposal.
- **Final presentations (10%)**: Dec 2nd and 4th
  In-class 15-20 minute presentations.

Your team will have a total of 5 late days which you can budget across any of the written reports (Report 1, Report 2, or the Final Report).

12-Unit: Project timeline (tentative)

- **Team formation**: Sept 9th
- **Project hours 1 (5%)**: Sept 30th
  Meet with course staff for 10 minutes, with a few slides.
- **Report 1 (25%)**: Oct 10th
  Task proposal and data analysis; related work; baseline proposal.
- **Fall break**: Oct 14th and 16th – Take time off!
- **Project hours 2 (5%)**: Nov 6th
- **Report 2 (25%)**: Nov 11th
  Baseline results and analysis, and a technique proposal.
- **Final presentations (10%)**: Dec 2nd and 4th
  In-class 15-20 minute presentations.
- **Final report (30%)**: Dec 8th
  Results and analysis of your technique; future work proposal.

Your team will have a total of 5 late days which you can budget across any of the written reports (Report 1, Report 2, or the Final Report).

Discussion

- Introduce yourself! Name and program.
- What brings you to this class?

Neural code generation

- Part I: *Foundations*
  - ***Learning***, inference, data, evaluation
- Part II: *Frontiers*
  - Interaction, adaptability, reasoning, formal methods, science

Next meeting: lecture on pretraining and scaling laws for code

References i