Analyzing Software using Deep Learning

Reasoning about Types and Code Changes with Hierarchical Networks

Prof. Dr. Michael Pradel

Software Lab, University of Stuttgart

Summer 2022

Overview

- **Hierarchical neural networks** ←
- **Type prediction**

  Based on "TypeWriter: Neural Type Prediction with Search-based Validation" by Pradel et al., 2020
- **Representing code changes**

  Based on "CC2Vec: Distributed Representations of Code Changes" by Hoang et al., 2020

Motivation

- **What if the input to a predictive model consists of multiple parts that**
  - are too many to simply concatenate
  - are not a sequence
  - may each have a different structure?

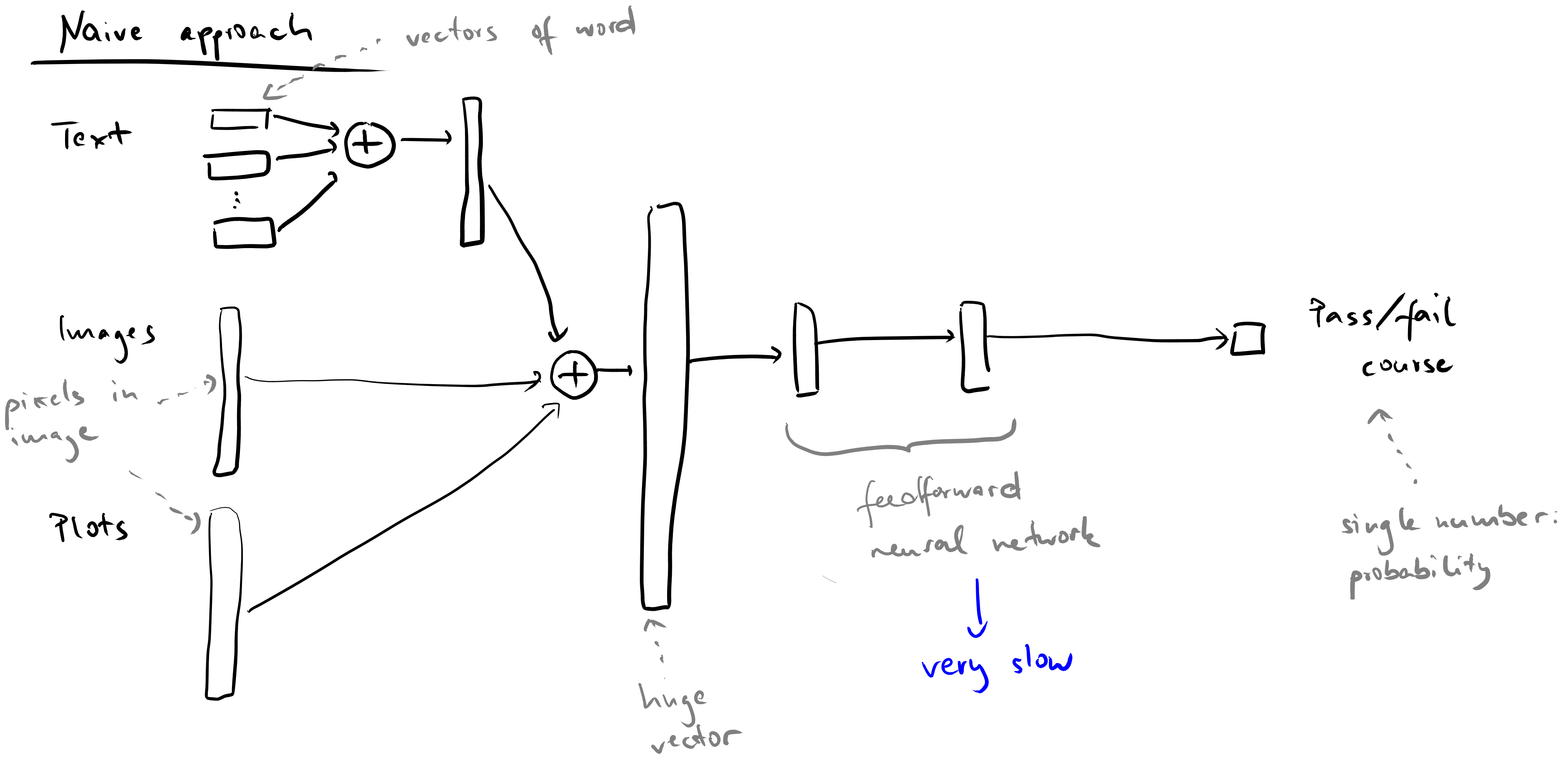
Examples

- **Document**
  - Lines of words
  - Images
  - Plots
- **Evidence of program crash**
  - Stack trace
  - Error message
  - Information about the user (key-value pairs)

Examples (2)

- **Program elements that have a type**
  - Code tokens
  - Identifier names
  - Comments associated with the function
- **Commits to a code repository**
  - Code change
    - Multiple code locations
  - Commit messages

Naive approach



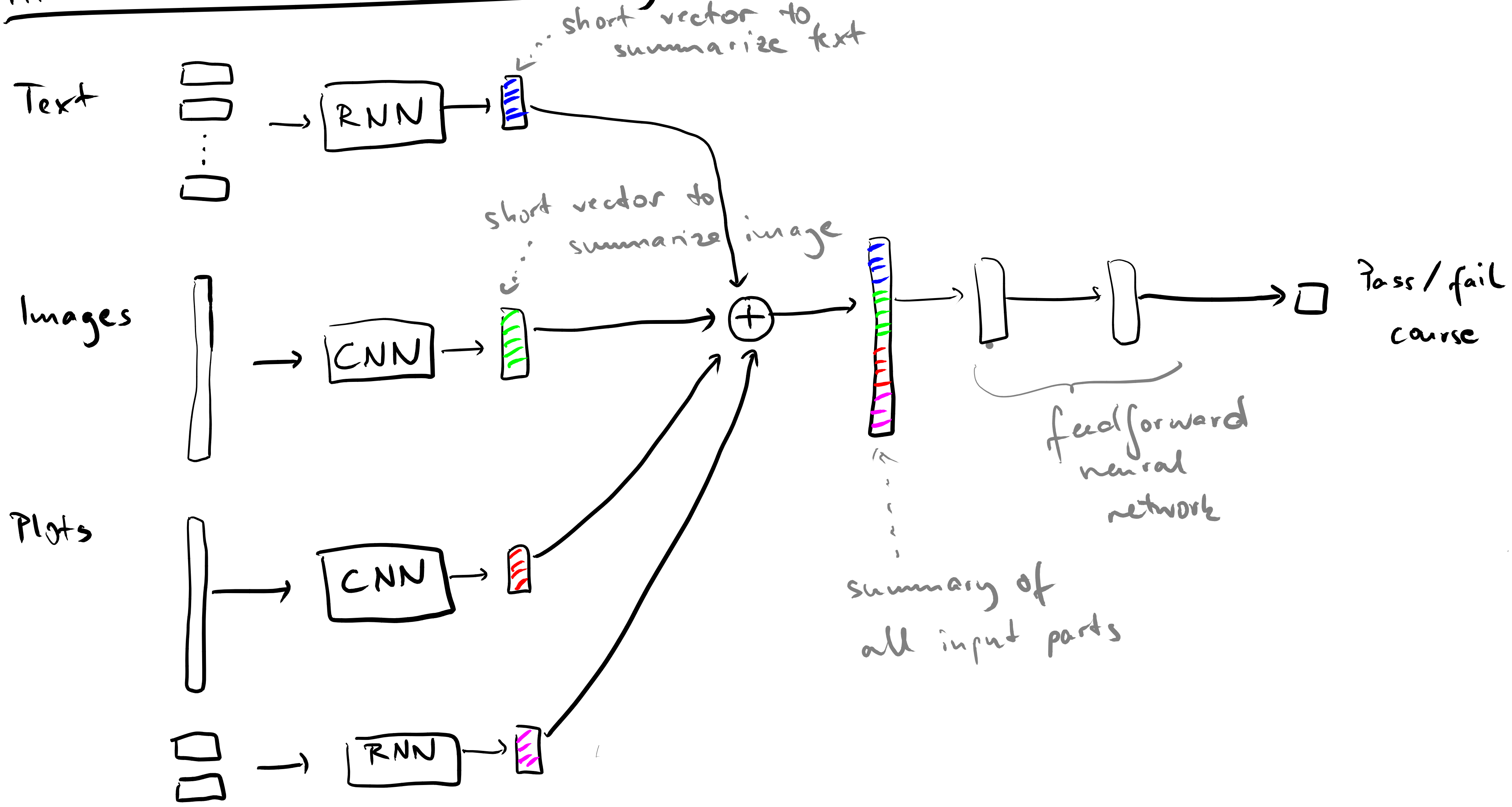
Hierarchical Neural Networks

- **Neural model composed of submodels**
- **Aligned into a hierarchy**
    - E.g., a tree where inputs arrive at leaves
    - Information propagates from leaves to the root
- **Prediction based on summarized information at root**

Submodels

- **Each submodel: Encode specific part of input**
- **Different submodels may be different kinds of neural networks**
    - E.g., feedforward network for some input, RNN for some other input

Hierarchical Model to Classify Documents



Jointly Training the Model

How to train a hierarchical neural network?

Option 1: Train each submodel separately

- ✓ Training focuses on specific model and its input
- ✗ Need training data for each submodel
- ✗ Submodel isn't aware of the overall task

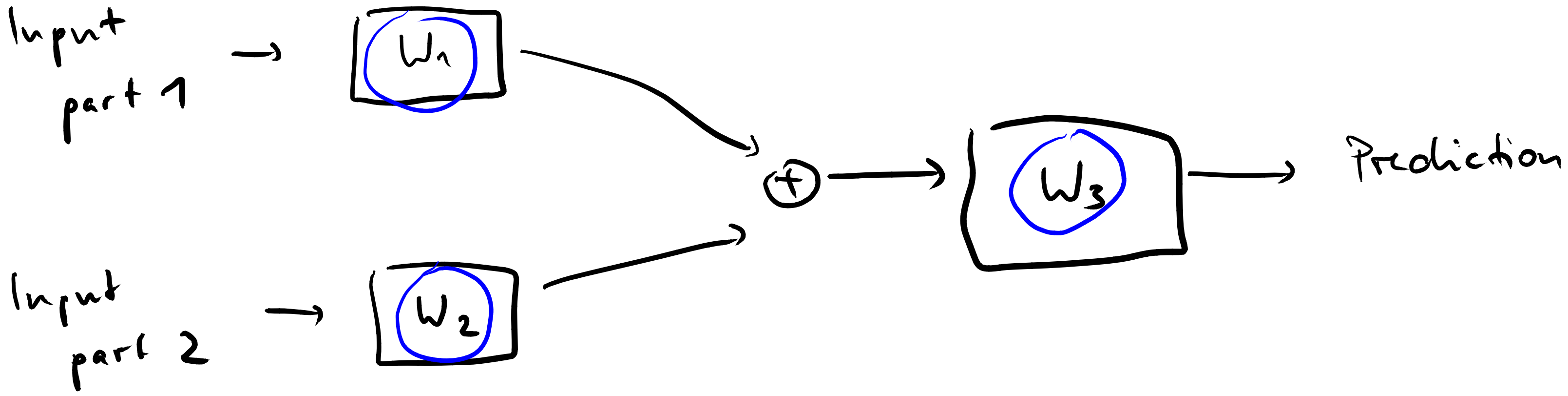
Jointly Training the Model

How to train a hierarchical neural network?

Option 2: Train entire model jointly

- ✓ Need training data only for the overall task
- ✓ Submodels get optimized for the overall task
- ✗ For large models, feedback from final prediction may get lost (vanishing gradient problem)

Example: Joint Training



◯ ... all optimized together

Overview

- **Hierarchical neural networks**
- **Type prediction** ←

  Based on “TypeWriter: Neural Type Prediction with Search-based Validation” by Pradel et al., 2020
- **Representing code changes**

  Based on “CC2Vec: Distributed Representations of Code Changes” by Hoang et al., 2020

Types in Dynamic Progr. Langs.

- **Dynamically typed languages:**
  Extremely popular
- **Lack of type annotations:**
  - Type errors
  - Hard-to-understand APIs
  - Poor IDE support

Example

```
def find_match(color):
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

Gradual Typing

- **Annotate some code locations with types**
  - E.g., parameter types and return types of some functions only
- **Gradual type checker**
  - Warn about inconsistencies
  - Ignores missing information

Gradual Typing

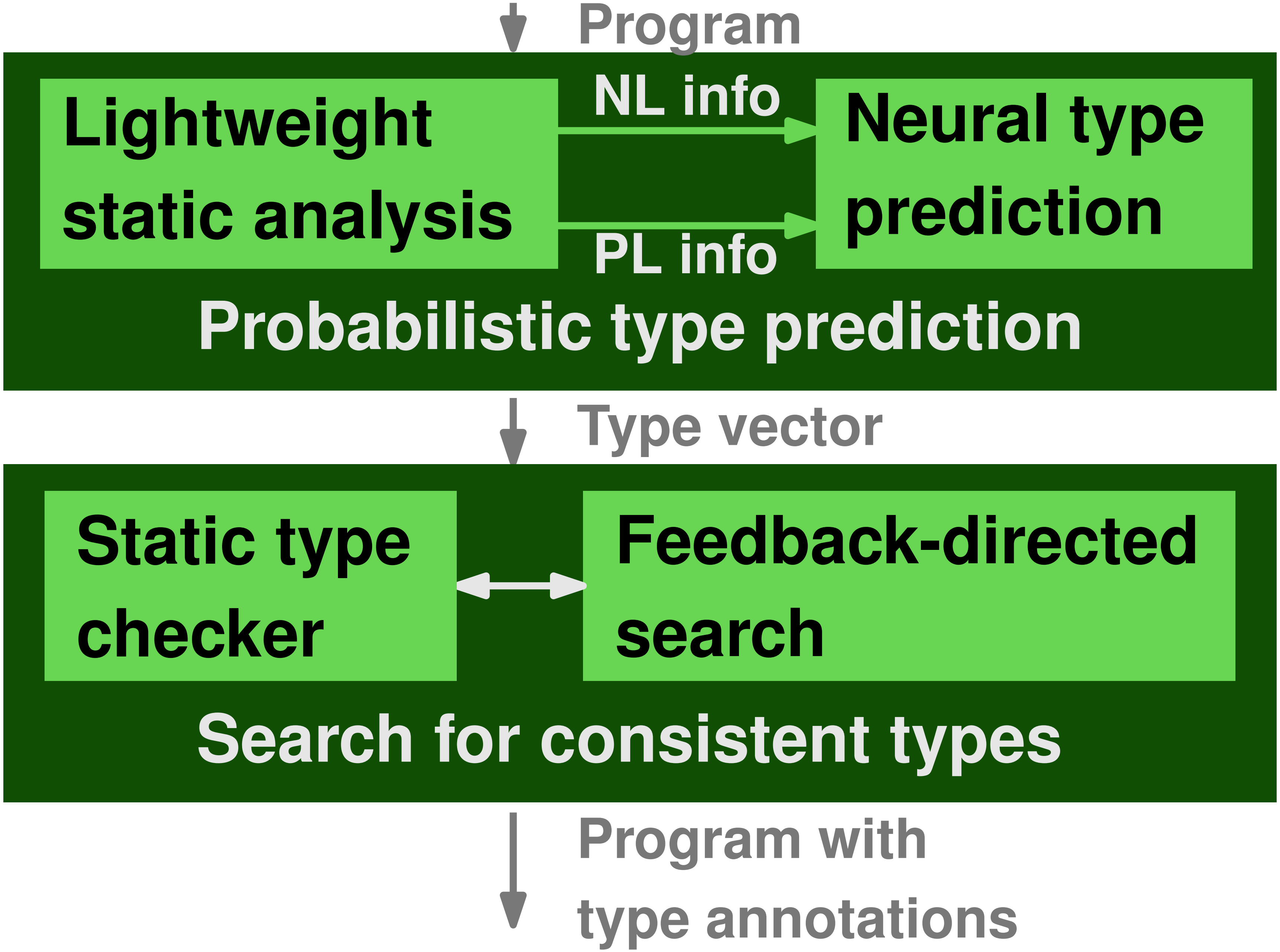
- **Annotate some code locations with types**
  - E.g., parameter types and return types of some functions only
- **Gradual type checker**
  - Warn about inconsistencies
  - Ignores missing information

But: Annotating types is painful

How to Add Type Annotations?

- **Option 1: Static type inference**
  - Guarantees type correctness, but very limited
- **Option 2: Dynamic type inference**
  - Depends on inputs and misses types
- **Option 3: Probabilistic type prediction**
  - Models learned from existing type annotations

Overview of TypeWriter



Program

Lightweight static analysis

NL info

PL info

Neural type prediction

Probabilistic type prediction

Type vector

Static type checker

Feedback-directed search

Search for consistent types

Program with type annotations

Extracting NL and PL Info

- **NL information**
  - Names of functions and arguments
  - Function-level comments
- **PL information**
  - Occurrences of the to-be-typed code element
  - Types made available via imports

Example

```
def find_match(color):
  """
  Args:
    color (str): color to match on and return
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

Example

Names: find_match, color

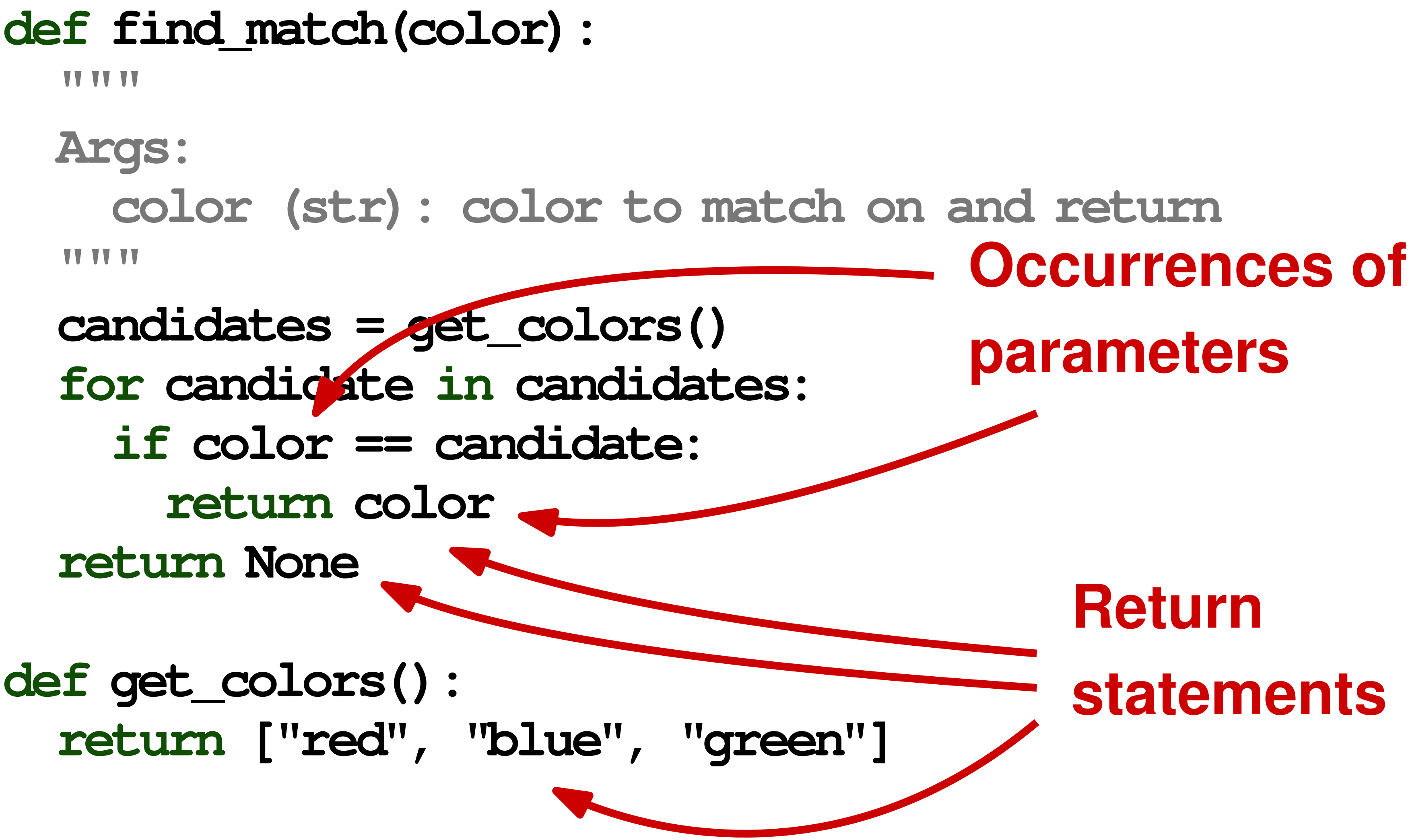
```
def find_match(color):
  """
  Args:
    color (str): color to match on and return
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None
```

Function-level comment

Names: get_colors

```
def get_colors():
  return ["red", "blue", "green"]
```

Example



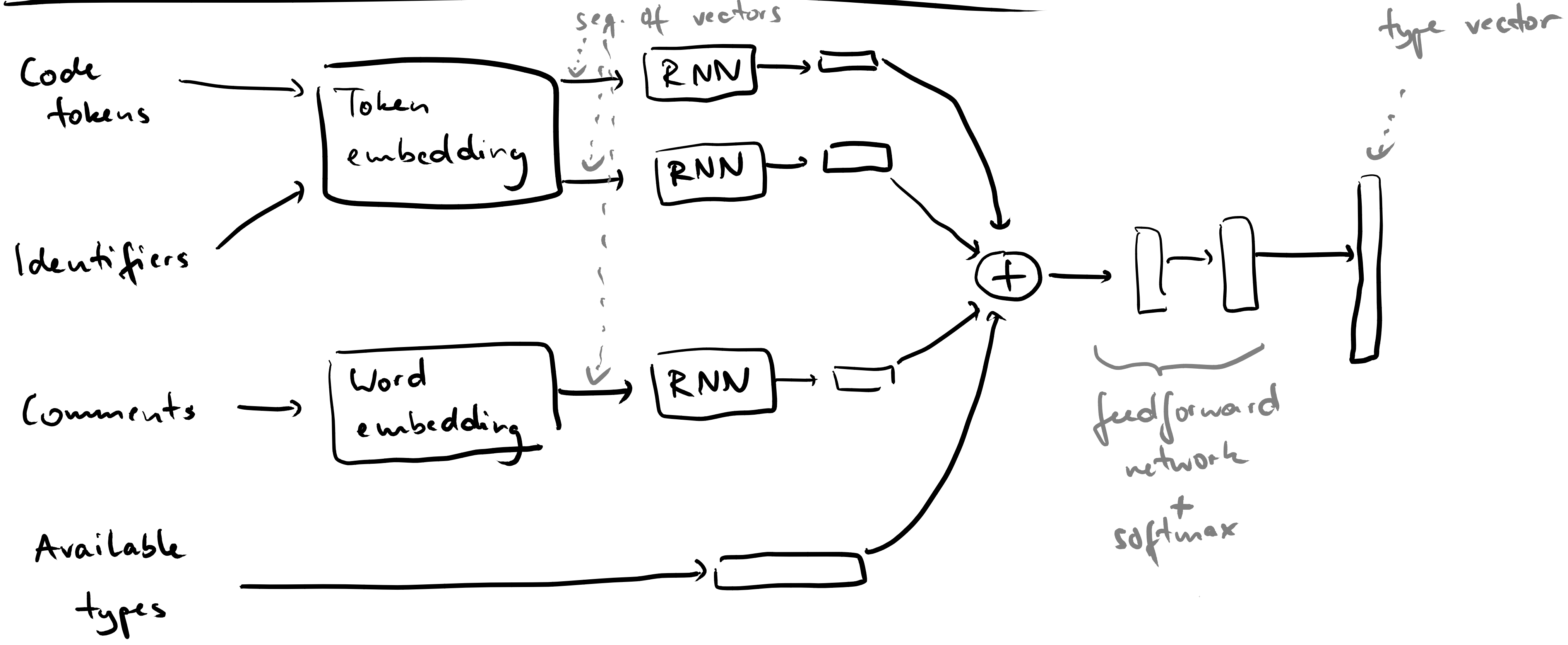
```
def find_match(color):
  """
  Args:
    color (str): color to match on and return
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

Occurrences of parameters

Return statements

Hierarchical neural network for type prediction



Output: Type Vector

- **Type prediction as a classification problem**
- **Output of the model: Type vector**
  - One element for each of top-1000 types
  - During training:

    All zero, except for the correct type
  - During prediction:

    Interpreted as probability distribution over types

Training the Model

- **Training data: Existing type annotations**
  - Multi-million line code base
  - Some types (≈ 20-50%) already annotated
- **Learns to predict missing types from existing annotations**

Example of Predictions

```
def find_match(color):
  """
  Args:
    color (str): color to match on and return
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

Example of Predictions

```
def find_match(color):
  """
  Args:
    color (str): color to match on and return
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

Predictions:
int, str, bool

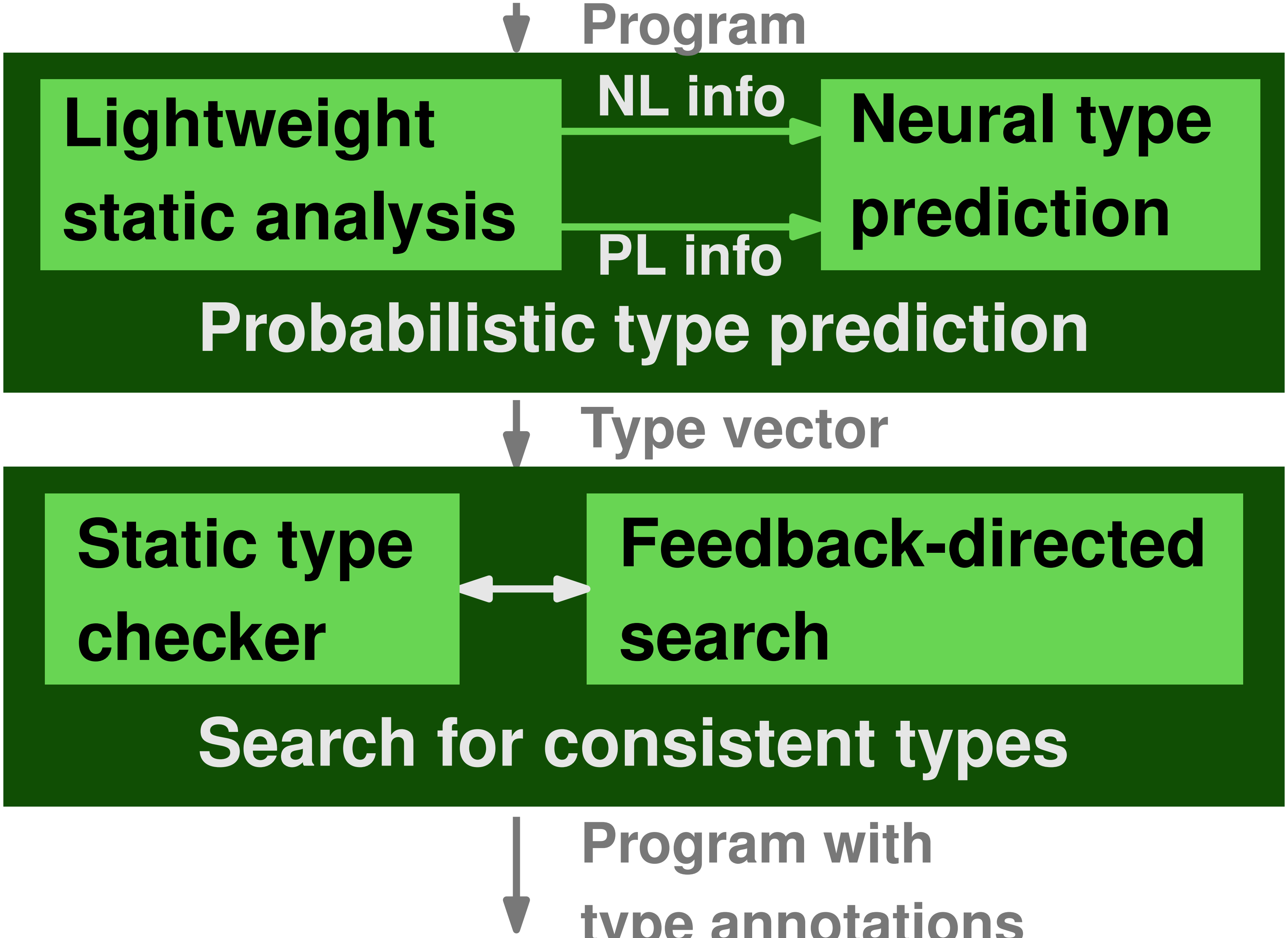
Predictions: str,
Optional[str],
None

Predictions:
List[str],
List[Any], str

Challenges

- **Imprecision**
  - Some predictions are wrong
  - Developers must decide which suggestions to follow
- **Combinatorial explosion**
  - For each missing type: One or more suggestions
  - Exploring all combinations: Practically impossible

Overview of TypeWriter



Program

Lightweight static analysis

NL info

PL info

Neural type prediction

Probabilistic type prediction

Type vector

Static type checker

Feedback-directed search

Search for consistent types

Program with type annotations

Searching for Consistent Types

- **Top-k predictions for each missing type**
  - Filter predictions using gradual type checker
  - E.g., pyre and mypy for Python, flow for JavaScript
- **Combinatorial search problem**
  - For type slots S and k predictions per slot: $(k+1)^{|S|}$ possible type assignments

Searching for Consistent Types

- **Top-k predictions for each missing type**
  - Filter predictions using gradual type checker
  - E.g., pyre and mypy for Python, flow for JavaScript
- **Combinatorial search problem**
  - For type slots S and k predictions per slot: $(k+1)^{|S|}$ possible type assignments

Too large to explore exhaustively!

Feedback Function

- **Goal: Minimize missing types without introducing type errors**

- **Feedback score (lower is better):**

$$v \cdot n_{missing} + w \cdot n_{errors}$$

Feedback Function

- **Goal: Minimize missing types without introducing type errors**

- **Feedback score (lower is better):**

$$v \cdot n_{missing} + w \cdot n_{errors}$$

Default: $v = 1$**,** $w = 2$**,**
i.e., higher weight for errors

Search Strategies

- **Optimistic** vs. **pessimistic**
  - Add top-most predicted type everywhere and then remove types
  - Add one type at a time
- **Greedy** vs. **non-greedy**
  - If score decreases, keep the type
  - Backtrack to avoid local minima

Example

```
def find_match(color):
  """
  Args:
    color (str): color to match on and return
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

Example

```
def find_match(color):
  """
  Args:
    color (str): color to match on and return
  """
  candidates = get_colors()
  for candidate in candidates:
    if color == candidate:
      return color
  return None

def get_colors():
  return ["red", "blue", "green"]
```

Predictions:
int, str, bool

Predictions: str,
Optional[str],
None

Predictions:
List[str],
List[Any], str

Results

- **Neural model**
  - Precision: 58-73% in top-1, 50-92% in top-5
  - Recall: 50-58
- **Model and search together**
  - Best strategy adds 72% of type-correct types and completely annotates 44% of files
- **In use at Facebook**
  - Thousands of suggested types accepted by developers with minimal effort

Overview

- **Hierarchical neural networks**
- **Type prediction**

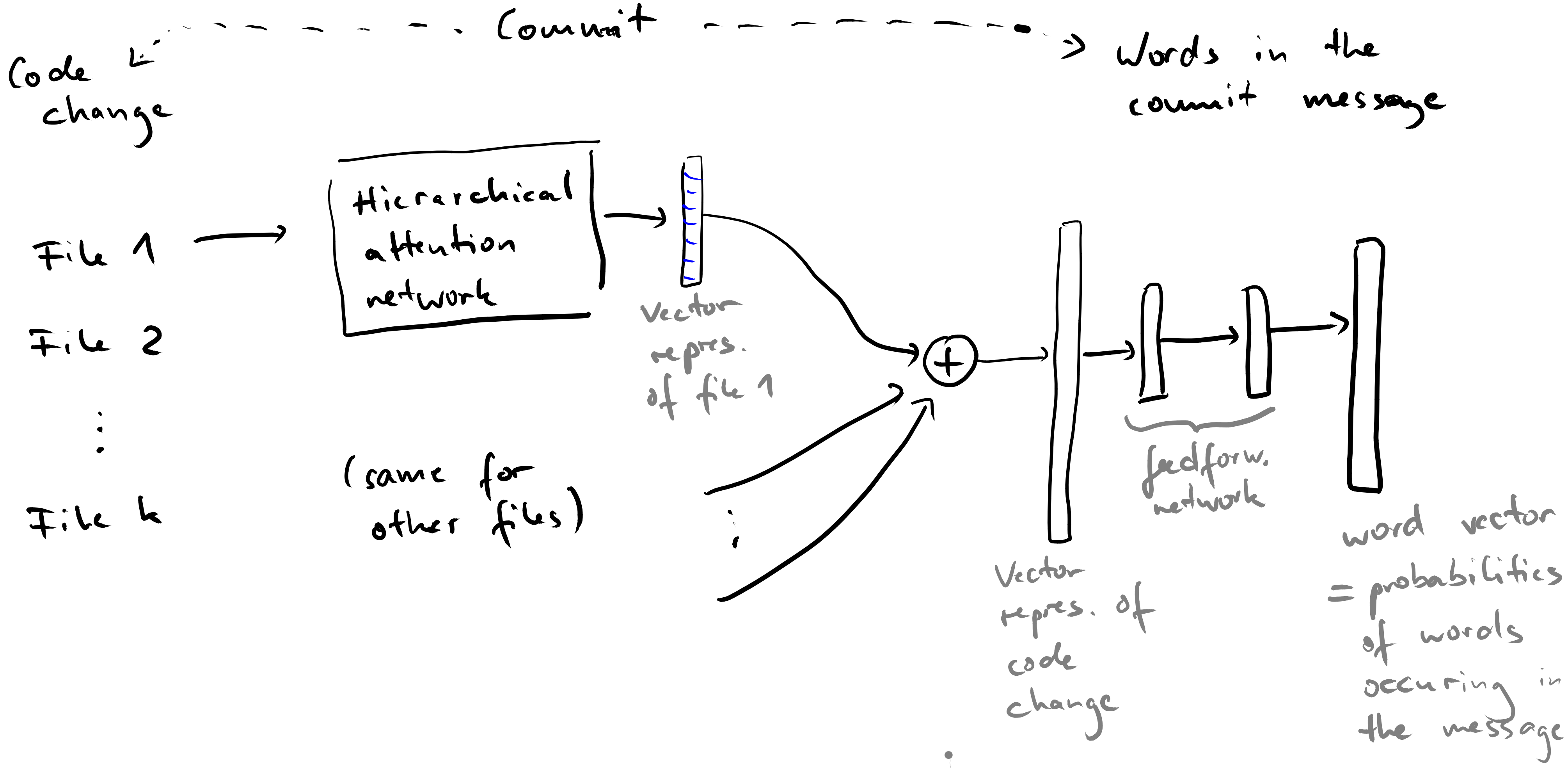
  Based on “TypeWriter: Neural Type Prediction with Search-based Validation” by Pradel et al., 2020
- **Representing code changes** ←

  Based on “CC2Vec: Distributed Representations of Code Changes” by Hoang et al., 2020

Representing Code Changes

- **Source code evolves all the time**
- **Goal: Represent code changes to make predictions**
  - What should be the commit message?
  - Does the change fix a bug?
  - Does the change introduce a bug?

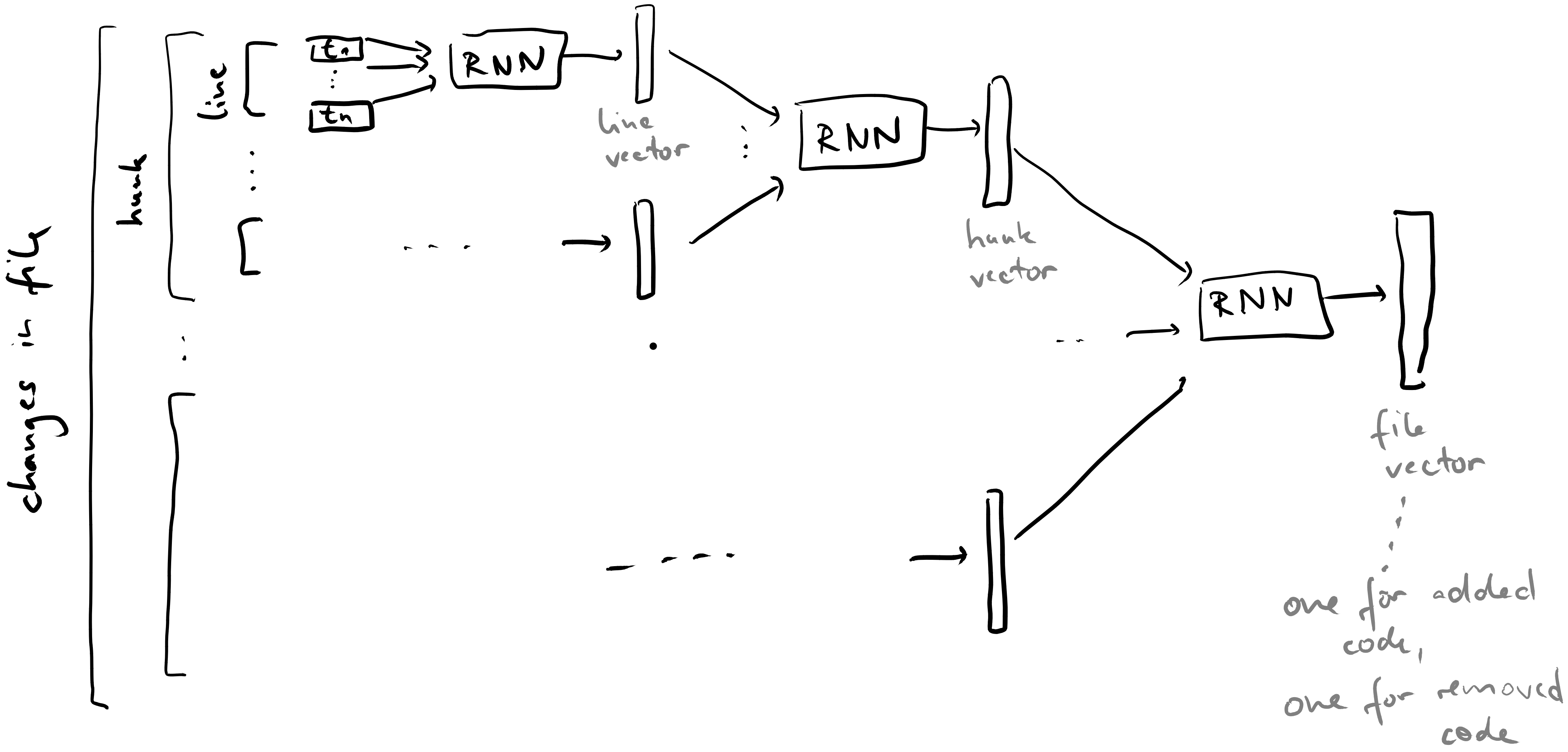
CC2Vec: Overview



Data Extraction

- **Each code change: Set of affected files**
- **Each affected file: Set of hunks**
  - Hunk = consecutive lines of modified code
- **Each hunk: Added and removed lines**
- **Each line: Sequence of code tokens**

Hierarchical Model



Comparison Layers

- **Goal: Focus on changes in a file**
- **Given: Vector representation of**
    - Added code: e_a
    - Removed code: e_r
- **Set of comparison functions**
    - E.g., element-wise subtraction
- **Result: One vector that summarizes all changes in a file**

Training the Model

- **Gather from version control system of project**
  - Pairs of code change and commit message
  - Evaluation with tens of thousands of pairs
- **Train entire model jointly**
- **Once trained, use embeddings of code changes for specific applications**

Applications

- **Predict commit message**
  - Search for nearest neighbor of code change and reuse it message
- **Predict: Is a code change a bug fix?**
  - Relevant, e.g., to decide which code changes to backport to older Linux kernel versions
- **Just-in-time defect prediction**
  - Useful to allocate quality assurance resources (e.g., code reviews) to code changes