Georgetown University
Computational Linguistics

GEORGETOWN UNIVERSITY

AllenNLP Workshop

Luke Gessler
Georgetown University
October 30, 2021

Acknowledgements

- GUCL Social Chairs Yifu Mu and Lauren Levine for organizing
- GUCL for funding snacks

- AllenNLP Guide: https://guide.allennlp.org/
- Matt Gardner's 2019 slides: https://docs.google.com/presentation/d/19ER1y4EAeh3ZF2yn7-LiKE6dIvCb7YSiRFpqZ6sgQc8/edit#slide=id.p

Introduction

NLP in 2021

- State of the art algorithms use neural networks
- But neural networks are hard to use
  - Need for special hardware
  - Sophisticated architectures
  - Many hyperparameters
  - Special data processing
    - Only numerical inputs
    - NN's aren't scale invariant

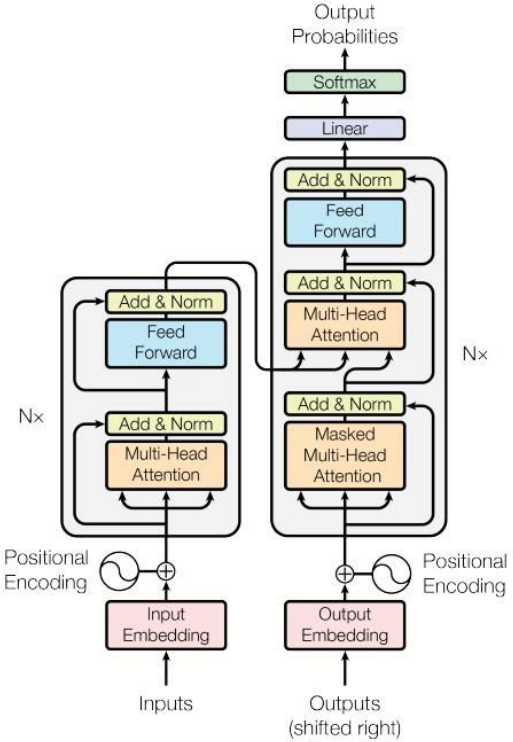
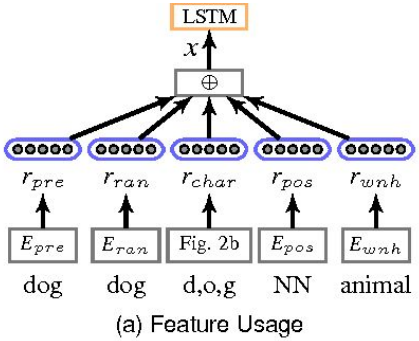
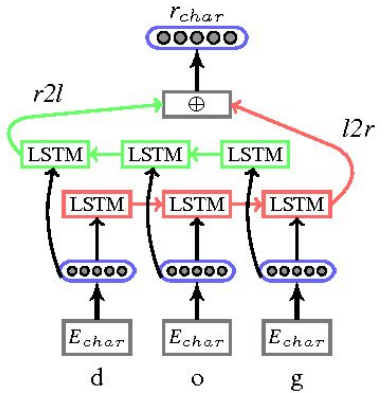


Figure 1: The Transformer - model architecture.



(a) Feature Usage



Introducing AllenNLP

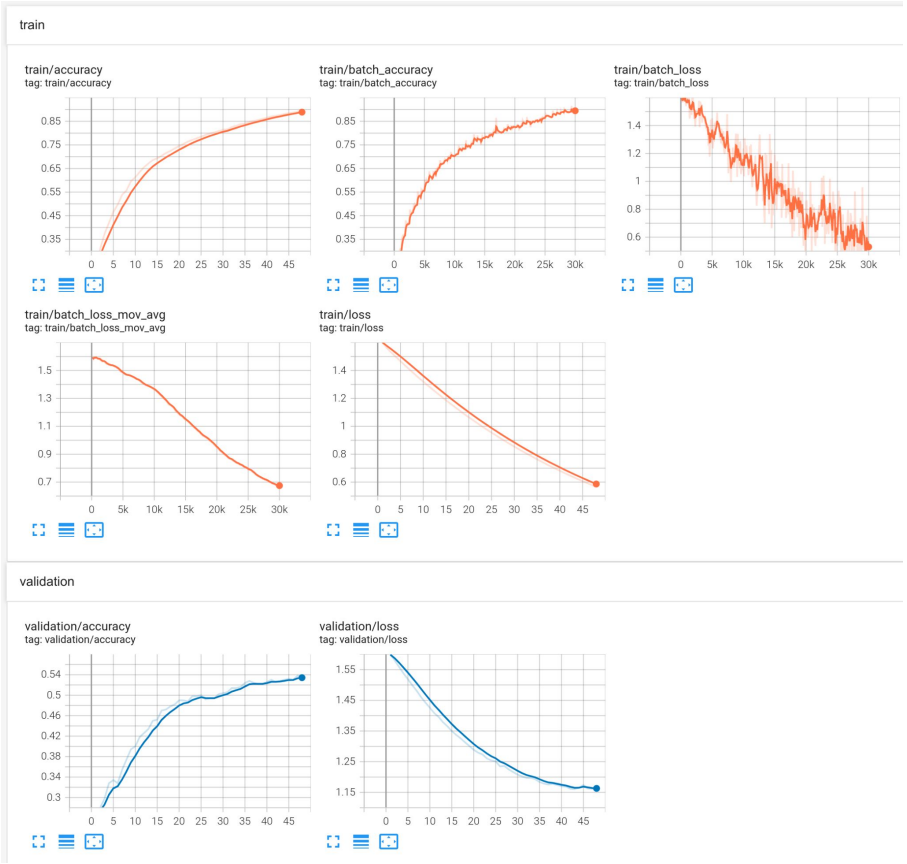
- Most NLP experiments look similar:
  - Take some text with annotations
  - Train a model to predict the annotations
  - Evaluate the model on held-out data
- AllenNLP provides **structure** and **sensible defaults**
  - Hard/tedious steps (batching, training with early stopping) are implemented for you, and are hyperparameterized
  - Object-oriented model of the experimental pipeline: abstract class (“interface”) for all components; use default impl. or your own

AllenNLP: Main Features

- **Modular, extensible** components
  - Model-related modules (e.g. Seq2SeqEncoder, TokenEmbedder)
  - Experiment-related modules (DataLoader, DatasetReader)
- **À la carte**: most utilities can be omitted or used on their own
- **Battle-tested code**: code for training, evaluation has been scrutinized
- **Centralized experiment management**: all hyperparameters, inputs, etc. in a single config file
- Very widely used by leading NLP institutions

Other Goodies

- Support for **distributed training**
  - Use more than one GPU node to accelerate training on large models/datasets
- Support for **multitask learning**
  - Often a huge pain to set up!
- **Tensorboard** integration
- “Free” **web demos**
  - Allow others to interact with your model on a webpage
  - No extra code



Highlights: Configs

- Hyperparameters, model architecture live in a **single config file**
  - Uniform, single source of truth for settings
  - No more digging through code to find out what a particular hyperparameter is

```
{
    "dataset_reader" : {
        "type": "classification-tsv",
        "token_indexers": {
            "tokens": {
                "type": "single_id"
            }
        }
    },
    "train_data_path": "/path/to/your/training/data/here.tsv",
    "validation_data_path": "/path/to/your/validation/data/here.tsv",
    "model": {
        "type": "simple_classifier",
        "embedder": {
            "token_embedders": {
                "tokens": {
                    "type": "embedding",
                    "embedding_dim": 10
                }
            }
        },
        "encoder": {
            "type": "bag_of_embeddings",
            "embedding_dim": 10
        }
    },
    "data_loader": {
        "batch_size": 8,
        "shuffle": true
    },
    "trainer": {
        "optimizer": "adam",
        "num_epochs": 5
    }
}
```

Highlights: Declarative Training

- Very sophisticated **Trainer** code provides everything needed for a supervised training loop
  - Control over all hyperparameters (batch size, learning rate/learning rate scheduler, optimizer, …)
- Instead of writing code, just write a config
- **Callback** interface allows arbitrary extension

Vanilla PyTorch

```
def train(dataloader, model, loss_fn, optimizer):
    size = len(dataloader.dataset)
    model.train()
    for batch, (X, y) in enumerate(dataloader):
        X, y = X.to(device), y.to(device)

        # Compute prediction error
        pred = model(X)
        loss = loss_fn(pred, y)

        # Backpropagation
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if batch % 100 == 0:
            loss, current = loss.item(), batch * len(X)
            print(f"loss: {loss:>7f}  [{current:>5d}/{size:>5d}]")
```

AllenNLP

```
"trainer": {
    "optimizer": {
        "type": "huggingface_adamw",
        "lr": 5e-4,
        "parameter_groups": [
            [[".*transformer.*"], {"lr": 1e-5}]
        ]
    },
    "patience": 10,
    "num_epochs": 60,
    "validation_metric": "+span_f1"
}
```

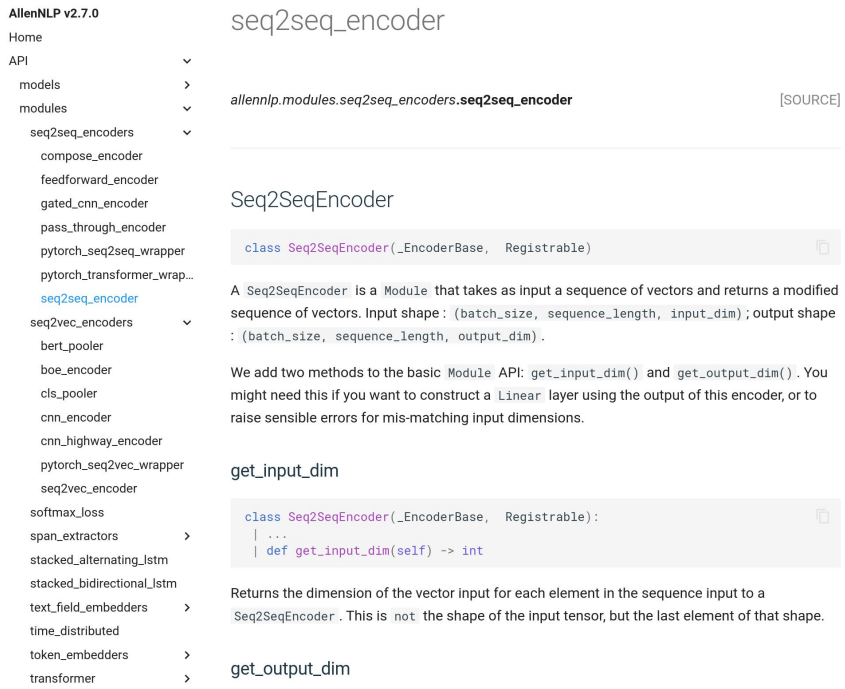
Highlights: Data ↔ Tensors

- **Instance**, **Field**, and **Vocabulary** abstractions make it easy to get data into and out of tensor representations
- This is the “missing piece”: PyTorch models turn tensors into tensors
- You don’t want to write this code yourself: it’s very error prone

```
>>> print(vocab)
Vocabulary with namespaces:
        Namespace: lemmas, Size: 14075
        Namespace: xpos_tags, Size: 61
        Namespace: tokens, Size: 6000
```

Highlights: NLP Module Bank

- NLP has a few very common kinds of modules:
  - Seq2Seq: turn a sequence of vectors into another sequence of vectors
  - Seq2Vec: turn a sequence of vectors into a single vector
  - Embedder: turn a sequence of tokens into a sequence of vectors
- AllenNLP provides multiple implementations of each, allowing you to **swap them without writing any code** (just modify the config)



Highlights: Ecosystem

- Seamless integration with the popular **`transformers`** package: use any popular contextualized word embedding model **just by supplying the name:** `bert-base-cased`, `SpanBERT/spanbert-large-cased`

- Large bank of AllenNLP models for most common NLP tasks: **`allennlp-models`**
  - Syntactic/semantic parsers
  - Coreference resolution models
  - Classification
  - Multimodal (text+vision)

Tasks and components

This is an overview of the tasks supported by the AllenNLP Models library along with the corresponding components provided, organized by category. For a more comprehensive overview, see the AllenNLP Models documentation or the Paperswithcode page.

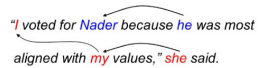
- **Classification**

  Classification tasks involve predicting one or more labels from a predefined set to assign to each input. Examples include Sentiment Analysis, where the labels might be `{"positive", "negative", "neutral"}`, and Binary Question Answering, where the labels are `{True, False}`.

  🛠 **Components provided:** Dataset readers for various datasets, including BoolQ and SST, as well as a Biattentive Classification Network model.

- **Coreference Resolution**

  Coreference resolution tasks require finding all of the expressions in a text that refer to common entities.

  See nlp.stanford.edu/projects/coref for more details.

  🛠 **Components provided:** A general Coref model and several dataset readers.

- **Generation**

  This is a broad category for tasks such as Summarization that involve generating unstructered and often variable-length text.

  🛠 **Components provided:** Several Seq2Seq models such a Bart, CopyNet, and a general Composed Seq2Seq, along with corresponding dataset readers.

- **Language Modeling**

  Language modeling tasks involve learning a probability distribution over sequences of tokens.

  🛠 **Components provided:** Several language model implementations, such as a Masked LM and a Next Token LM.

- **Multiple Choice**

  Multiple choice tasks require selecting a correct choice among alternatives, where the set of choices may be different for each input. This differs from classification where the set of choices is predefined and fixed across all inputs.

  🛠 **Components provided:** A transformer-based multiple choice model and a handful of dataset readers for specific datasets.

- **Pair Classification**

  Pair classification is another broad category that contains tasks such as Textual Entailment, which is to

Recap

- Highlights:
  - All experimental parameters in a single config file
  - No need to write training/eval code
  - Swappable implementations of common neural modules
  - Integration with the larger PyTorch NLP ecosystem
- Additionally, all of these pieces can be used independently
  - Some people don’t like configs--it’s possible to write code instead
  - Need an untraditional training loop? Just write your own

AllenNLP vs. Other Frameworks

- SpaCy: optimized for real-world, production use
- fairseq: only for sequence tasks
- Flair: optimized for ease of use, not as easy to extend
- AllenNLP: optimized for research use and modularity

Warning: Windows Support

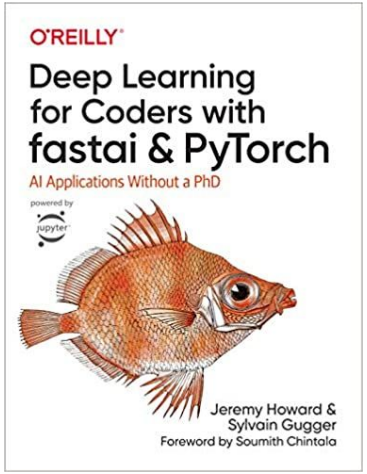
- AllenNLP works on Windows, but a little extra work is required
  - Devs are working on it
  - Biggest issue: .jsonnet configs not fully supported, must use .json instead
  - Issue: https://github.com/allenai/allennlp/issues/612
- Only an issue on local machines
  - You will probably use remote GPU machines or Google Colab anyway

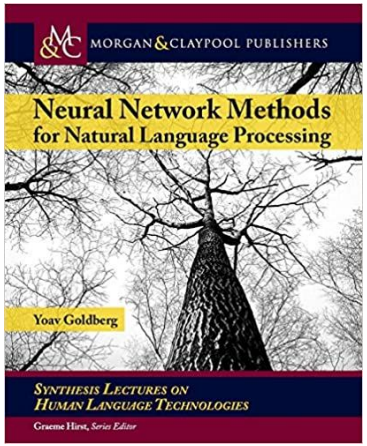
Workshop Scope

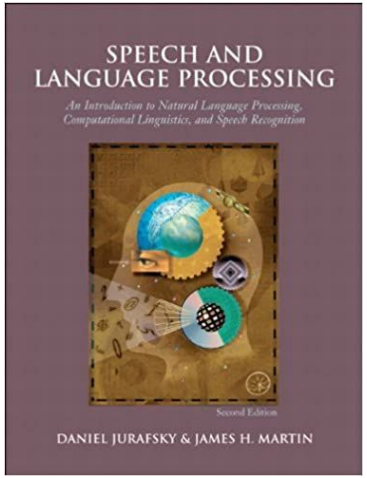
- Goal: learn core AllenNLP concepts, the nuts and bolts of good experiments
  - Read data
  - Make or re-use a model
  - Make an experimental configuration
  - Iterate quickly on experiments
- We will **not** cover (but you will be better equipped to learn about):
  - How to program your **forward()** functions, i.e. tensor operations and torch.nn modules
  - How to use SOTA models to get predictions
  - Hyperparameter optimization
  - Setting up and using computational environments (e.g. GPU clusters)
  - Advanced model architectures

Recommended further reading

- PyTorch: Howard and Gugger, *Deep Learning for Coders with Fastai and PyTorch*
- NLP, intro: Jurafsky and Martin, *Speech and Language Processing*
- NLP advanced: Goldberg and Hirst: *Neural Network Methods in Natural Language Processing*



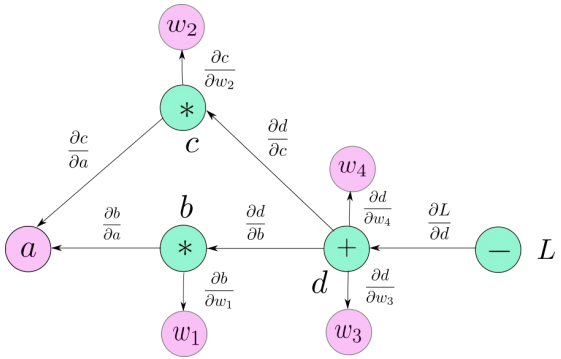




Recap: PyTorch

PyTorch

- PyTorch is a library for models that can efficiently learn how to turn tensors into tensors
- Support for GPU acceleration
- Automatic differentiation engine
  - Supports learning model parameters via backpropagation



Tensors

- Neural networks deal with **tensors**
- Essentially, a tensor is a collection of **numbers** with a certain **shape**
  - A specified number of dimensions, each with a number of elements
- Nicknames for tensors:
  - 0 axes: scalar
  - 1 axis: vector
  - 2 axes: matrix

Scalar

```
scalar = torch.tensor(3.14)
print(scalar)
print(scalar.shape)
```

```
tensor(3.1400)
torch.Size([])
```

Vector

```
vector = torch.tensor([3.1,-0.2,5.5])
print(vector)
print(vector.shape)
```

```
tensor([ 3.1000, -0.2000,  5.5000])
torch.Size([3])
```

Matrix

```
matrix = torch.tensor([[1.0, 0.0, 3.0],
                       [0.0, 1.0, 0.0],
                       [-3.0, 0.0, 1.0]])
print(matrix)
print(matrix.shape)
```

```
tensor([[ 1.,  0.,  3.],
        [ 0.,  1.,  0.],
        [-3.,  0.,  1.]])
torch.Size([3, 3])
```

Modules

- **Modules** are the building blocks of neural networks
- Workhorse for turning tensors into other tensors
- Modules have **parameters**
  - Special tensors which are updated during training
- Modules may also contain modules
- Invoked like a function

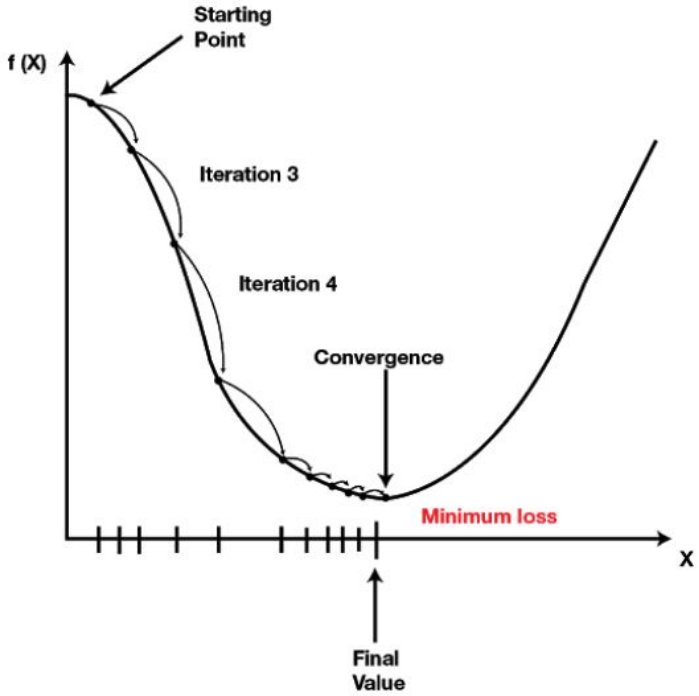
```
import torch.nn as nn
linear = nn.Linear(5,2)
for name, param in linear.named_parameters():
    print(name, param)
print()
print(linear(torch.ones(5,)))
```

```
weight Parameter containing:
tensor([[-0.0485, -0.0929,  0.3418, -0.3951,  0.3206],
        [ 0.1991,  0.1089, -0.2115,  0.1325,  0.1051]], requires_grad=True)
bias Parameter containing:
tensor([0.3587, 0.4160], requires_grad=True)

tensor([0.4845, 0.7500], grad_fn=<AddBackward0>)
```

Training

- Parameters need to be learned after being randomly initialized
- A **loss function** compares model outputs against expected outputs, produces a numerical **loss** value
- Loss is **minimized** by using a **backpropagation** algorithm to update model parameters



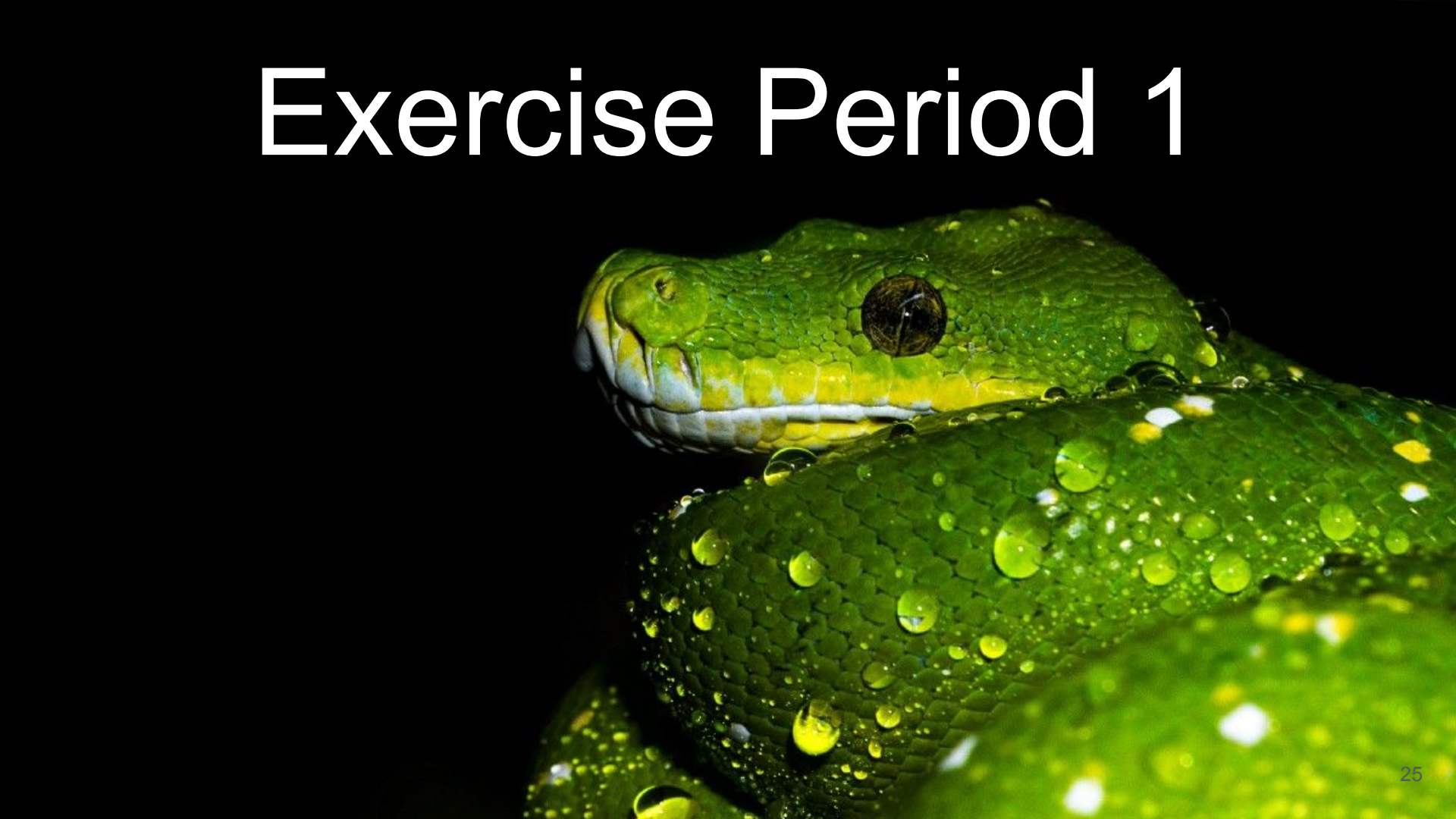
Recap

- PyTorch has two basic abstractions: **tensor**s and **module**s
  - Tensors:
    - collection of numbers
    - defined shape
  - Modules:
    - turn tensors into tensors
    - may have trainable parameters
    - may contain other modules
- Models are trained by **minimizing** a **loss function**

Exercise period

- See the exercise booklet
- **Make sure you `git pull` the latest version!**

Exercise Period 1



First Experiment

First Experiment

- It’s time to run your first AllenNLP experiment!
- Note: **small datasets** and **simple models** for today
  - You need to be able to run experiments in <5m!
  - Model performance will not be very good
- Quick overview of concepts before deeper dive in exercises
  - Concepts will transfer to real settings with large datasets and models
- Four essential ingredients supplied by you:
  - A dataset
  - A dataset reader
  - A model
  - An experiment configuration

Dataset

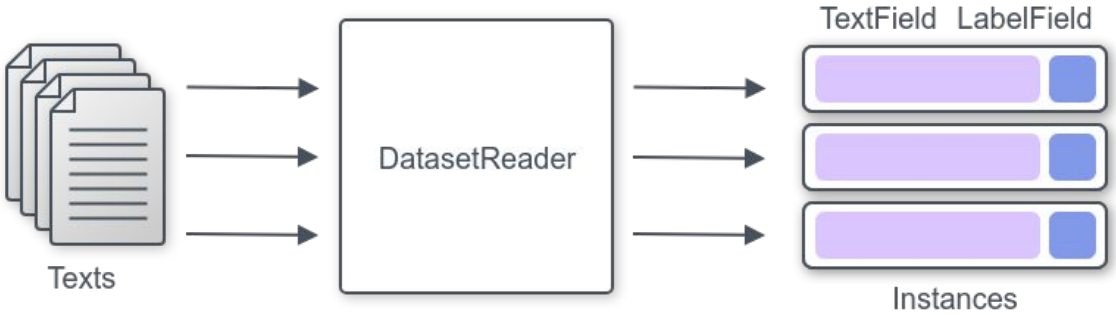
- A subset of the Yelp 2015 dataset: https://huggingface.co/datasets/yelp_review_full
- 700,000 reviews from Yelp with their star rating (1 to 5)
  - We'll only use 6,000 for our exercises
- Sample review (note that "label" is 0-indexed (0 means 1 star, etc.):

```
{
    "label": 0,
    "text": "This is the greasiest pizza
        I've ever had in Pittsburgh, I
        couldn't eat it."
}
```

- Goal: use text to predict label

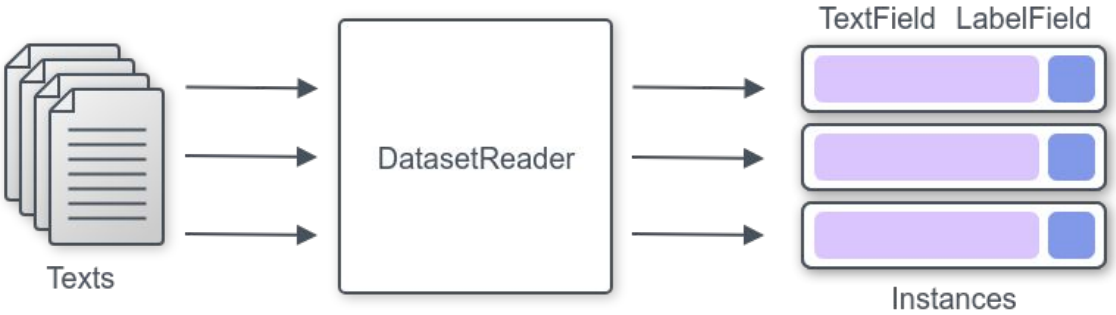
Dataset Reader

- **DatasetReader**: an AllenNLP abstract class for reading data
- Need to implement **_read(file_path)**, which reads a data file and turns it into a list of **Instance**s, which have **Field**s
- Tokenization happens at this step



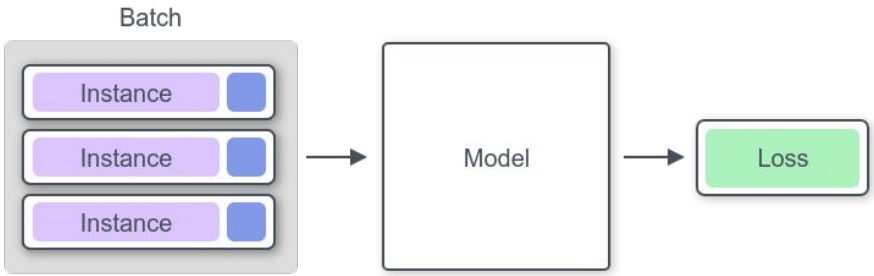
Dataset Reader

- **Instance**: multiple named **Field**s
- Different **Field** types: **TextField**s, **LabelField**s, ...
- **Field** has human-readable data, knows how to turn itself into a tensor
  - Text (i.e., list of tokens) needs to be turned into a tensor with word IDs



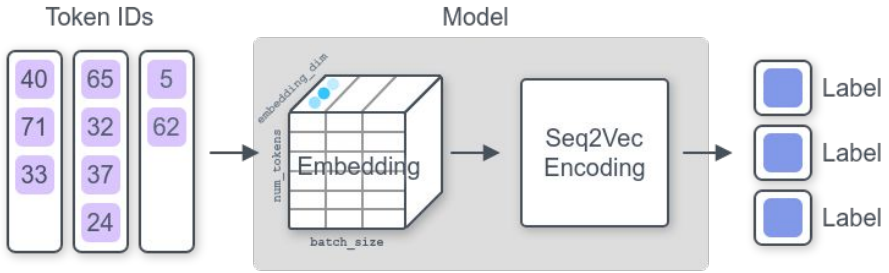
Models

- A **Model** takes **Instance**s (as batched tensors) in its **forward()** method and produces a dictionary with **loss**
  - This is different from a vanilla PyTorch model, which only outputs tensors
  - Other outputs may be included in the dictionary, e.g. predictions



Models

- Our model does the following:
  - Take token IDs and the label as input
  - Use a **TextFieldEmbedder** module to turn **words** into 50-dimensional **vectors**
  - Use a **Seq2VecEncoder** to get a single 50-dimensional vector for the entire *sentence*
  - Use a **Linear** layer to turn each sentence's 50-dimensional vector into a 5-dimensional vector
  - Record loss and accuracy



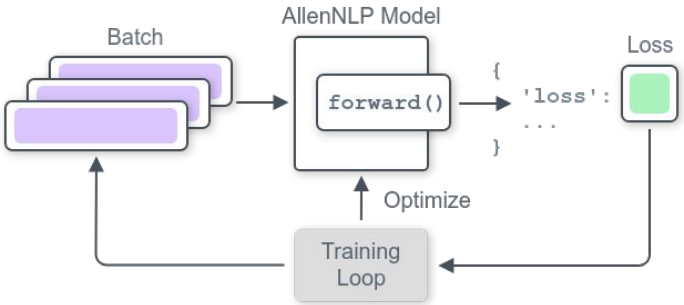
Configuration

- All parts discussed so far (dataset, dataset reader, model) “don’t know about each other”
  - Implicitly depend on each other: model is expecting fields called **text** and **label**, dataset reader produces fields called **text** and **label**
  - But the the components do not explicitly depend on each other
- Need to tie them together somehow

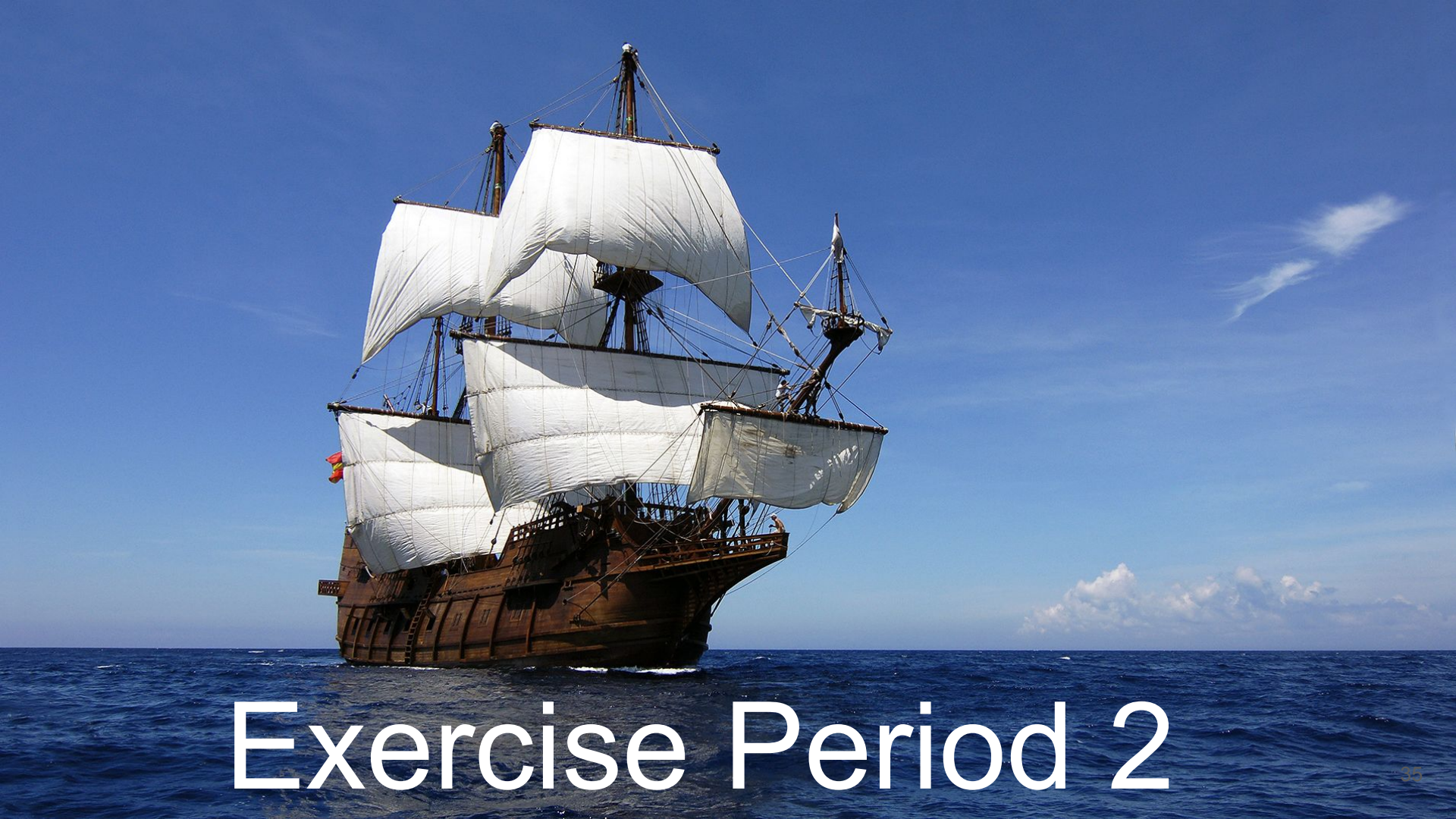
```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
"train_data_path": "data/train_5000.jsonl",
"validation_data_path": "data/dev_500.jsonl",
"model": {
    "type": "simple_classifier",
    "embedder": {
        "token_embedders": {
            "tokens": {
                "type": "embedding",
                "embedding_dim": 50
            }
        }
    },
    "encoder": {
        "type": "bag_of_embeddings",
        "embedding_dim": 50
    }
},
"data_loader": {
    "batch_size": 8,
    "shuffle": true
},
"trainer": {
    "optimizer": "huggingface_adamw",
    "num_epochs": 10,
    "patience": 2,
    "cuda_device": -1,
    "callbacks": ["tensorboard"]
}
```

Configuration

- Solution: a configuration file
- Declares all required models and data utilities, and details about the training loop
- AllenNLP reads this “recipe”, constructs objects, runs code




```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
"train_data_path": "data/train_5000.jsonl",
"validation_data_path": "data/dev_500.jsonl",
"model": {
    "type": "simple_classifier",
    "embedder": {
        "token_embedders": {
            "tokens": {
                "type": "embedding",
                "embedding_dim": 50
            }
        }
    },
    "encoder": {
        "type": "bag_of_embeddings",
        "embedding_dim": 50
    }
},
"data_loader": {
    "batch_size": 8,
    "shuffle": true
},
"trainer": {
    "optimizer": "huggingface_adamw",
    "num_epochs": 10,
    "patience": 2,
    "cuda_device": -1,
    "callbacks": ["tensorboard"]
}
```


Exercise Period 2

The Experimental Cycle

Model walkthrough

- You’ve now run your first model--but what happened?
- We’ll now take a closer look
- Many details, but focus just on the big picture
  - All you need to know and write code for is what we covered in the previous section
  - This overview is showing you how they’re put together, under the hood

Overview

- Setup
- Training loop
- Cleanup

Setup

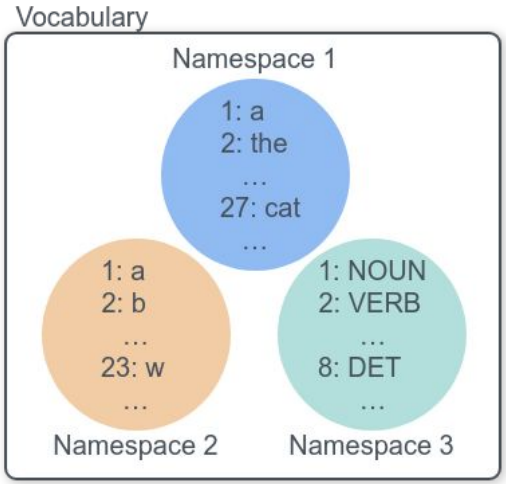
- Read config file
  - All following steps use the config as a “recipe sheet”

Setup

- Read config file
  - All following steps use the config as a “recipe sheet”
- Read data
  - The **DatasetReader** we provided reads **Instance**s
  - **DataLoader** does the batching and tensor-ifying

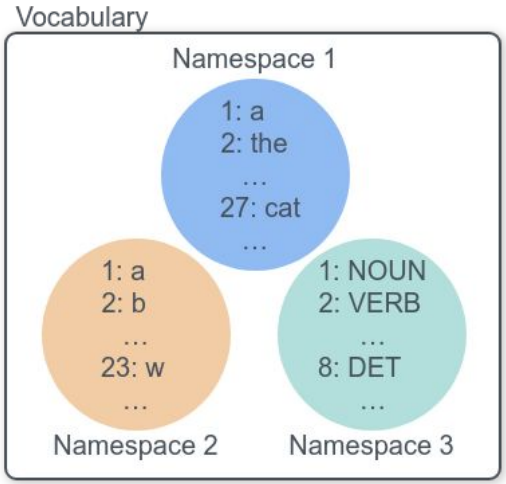
Setup

- Read config file
  - All following steps use the config as a “recipe sheet”
- Read data
  - The **DatasetReader** we provided reads **Instance**s
  - **DataLoader** does the batching and tensor-ifying
- **Vocabulary** is constructed
  - strings ↔ numbers within a **namespace**
    - Keep POS tags separate from tokens, etc.
  - To construct: find all unique values of every **Field**



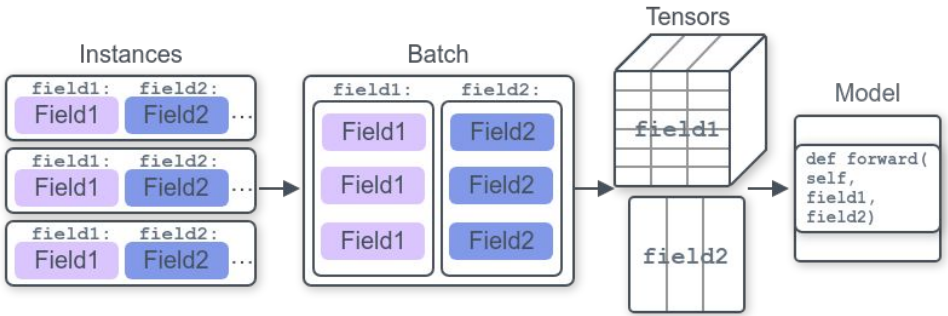
Setup

- Read config file
  - All following steps use the config as a “recipe sheet”
- Read data
  - The **DatasetReader** we provided reads **Instance**s
  - **DataLoader** does the batching and tensor-ifying
- **Vocabulary** is constructed
  - strings ↔ numbers within a **namespace**
    - Keep POS tags separate from tokens, etc.
  - To construct: find all unique values of every **Field**
- **Model** (the one we wrote) is constructed



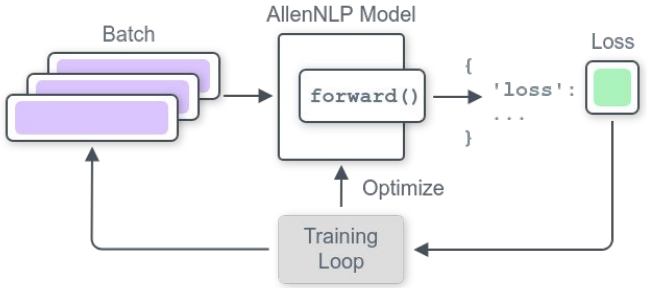
Training loop: terms

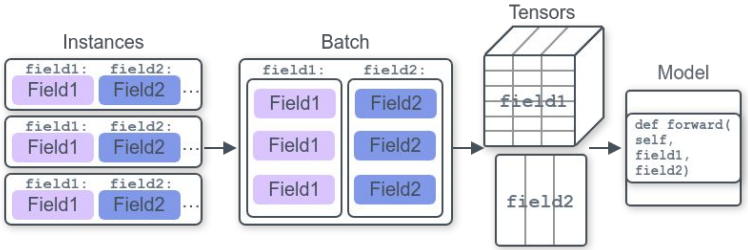
- **batch**: a collection of **Instance**s packed together
  - A practical method: GPUs can process larger, combined tensors more efficiently
  - Suppose we have a batch with 3 instances
  - If one instance has label 0, and two instances have label 1, the batched label field will be `torch.tensor([0, 1, 1])`
- **epoch**: a full run-through of the data during training



Training loop

- Loop through data for # epochs
  - Batch fed into the model
  - Loss is produced by the model
  - Optimizer updates model parameters
- Save metrics (e.g. accuracy) on a dev set at the end of each epoch
- If **patience** is supplied, stop early if validation performance fails to improve for that number of epochs






```
"trainer": {
    "optimizer": "huggingface_adamw",
    "num_epochs": 5,
    "patience": 2,
}
```

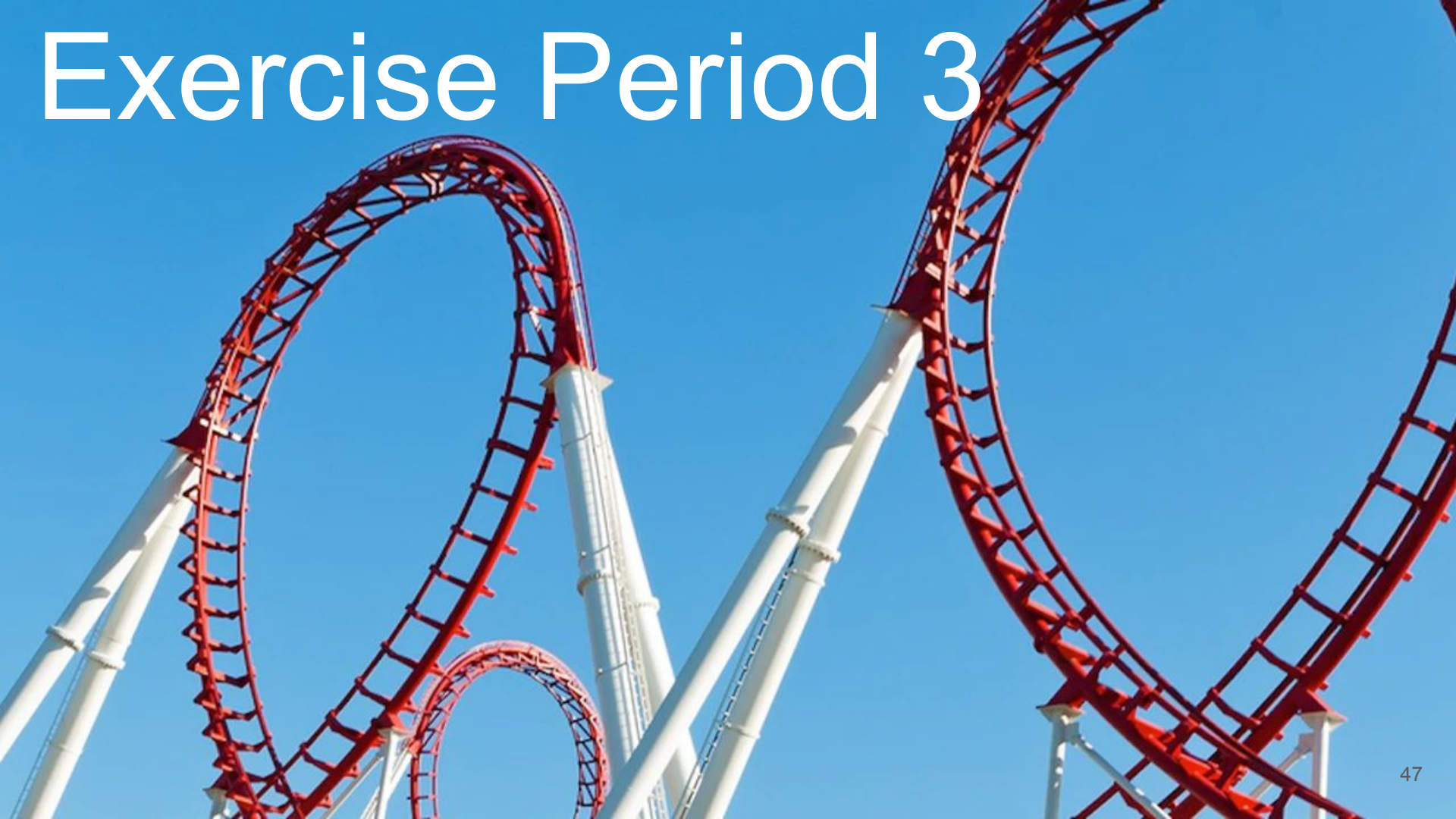
Cleanup

- Save the model
- Note final values of performance metrics

Remarks

- Finding more detail
  - Use a debugger in your favorite Python IDE to trace execution
  - See debug.py
  - Most of the work happens in **allennlp.commands.train**
- What if I want to change the training process?
  - Almost all of the time, you can do so by just modifying the configuration
  - If you really want to, you can implement your own **Trainer** or write your own code to manipulate **Model**s, **DatasetReader**s, etc. the way you'd like

Exercise Period 3




Fields

Fields

- Recall: a data point is just an **Instance** with **Field**s
- Basic types:
  - **TextField**: a sequence of tokens, for natural language text
  - **LabelField**: a single string value, e.g. for sentiment classification
- Sequential types, rely on a “parent” field:
  - **SpanField**: start and begin indexes identifying a subsequence, e.g. for named entities
  - **SequenceLabelField**: one string value per item, e.g. for POS tags
- More types: https://docs.allennlp.org/main/api/data/fields/field/

```
fields = {
    "text": TextField(tokens, self.token_indexers),
    "label": LabelField(label),
}
return Instance(fields)
```

TextField

- Most fields are easy to turn into tensors
- Biggest exception: **TextField**
- Successful handling of a **TextField** requires cooperation between the **tokenizer**, the **indexer**, and the **embedder**

Turning text into tensors

- Begin with a string: `"Hello, world"`
- Desired output: a tensor like `torch.tensor([101, 2, 729])`
  - And within our model, embeddings like `torch.tensor([[0.6645, ..., 0.0155], ...])`

Turning text into tensors

- Begin with a string: `"Hello, world"`
- Desired output: a tensor like `torch.tensor([101, 2, 729])`
  - And within our model, embeddings like `torch.tensor([[0.6645, ..., 0.0155], ...])`

- Broken into three steps:
  - Tokenization: `"Hello, world"` → `["Hello", ",", "world"]`
  - Indexing: `["Hello", ",", "world"]` → `torch.tensor([101, 2, 729])`
  - Embedding: `torch.tensor([101, 2, 729])` → `torch.tensor([[0.6645, ..., 0.0155], ...])`

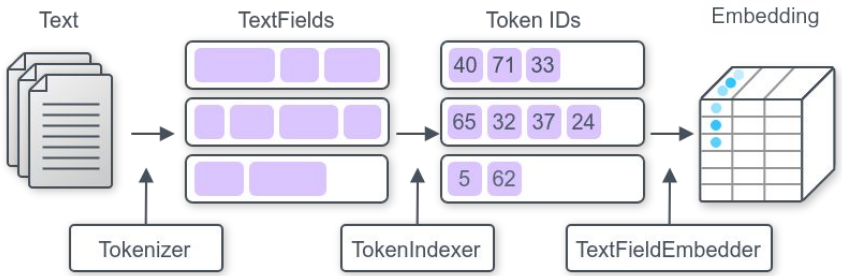
Turning text into tensors

- Begin with a string: `"Hello, world"`
- Desired output: a tensor like `torch.tensor([101, 2, 729])`
  - And within our model, embeddings like `torch.tensor([[0.6645, ..., 0.0155], ...])`

- Broken into three steps:
  - Tokenization: `"Hello, world"` → `["Hello", ",", "world"]`
  - Indexing: `["Hello", ",", "world"]` → `torch.tensor([101, 2, 729])`
  - Embedding: `torch.tensor([101, 2, 729])` → `torch.tensor([[0.6645, ..., 0.0155], ...])`
- Tokenization happens in **DatasetReader**, **TextField** initially stores tokens
- Indexing is performed by the **DataLoader** when batches are created
- Embedding happens during a model's **forward()** pass

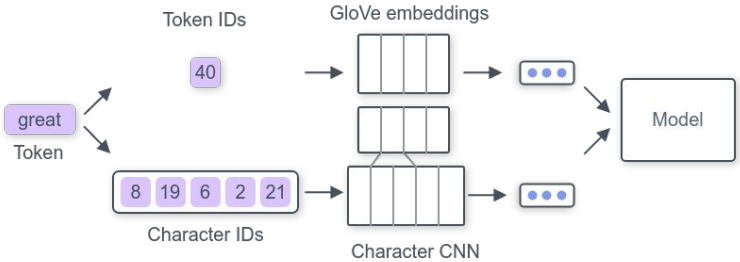
Text modules

- Three steps of turning text into tensors correspond to these abstractions:
  - **Tokenizer**
  - **TokenIndexer**
  - **TokenEmbedder**
- We could have stopped there, **but**: people want multiple embeddings for the same text
  - Character embeddings and static embeddings



Handling Multiple Embeddings

- Solution:
  - Give each embedding a unique key, e.g. `"tokens"` or `"chars"`
  - Allow **TextField** to keep track of all indexers: **token_indexers** is a dict with the key and the corresponding **TokenIndexer** object for each kind of embedding
  - **TextField** input to a model is a dict with each **TokenIndexer**'s output, which is another dict
  - Model has a **TextFieldEmbedder** with a keyed dict of **TokenEmbedder**s
- Uncommon that you'll want this, but may be useful
- Benefit: adding e.g. character embeddings only requires editing your config!



Text handling in the config

Tokenize: keep letters and numbers together, separate otherwise

```
"Hello, world"
→ ["Hello", ",", "world"]
```

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
"train_data_path": "data/train_5000.jsonl",
"validation_data_path": "data/dev_500.jsonl",
"model": {
    "type": "simple_classifier",
    "embedder": {
        "token_embedders": {
            "tokens": {
                "type": "embedding",
                "embedding_dim": 50
            }
        }
    },
    "encoder": {
        "type": "bag_of_embeddings",
        "embedding_dim": 50
    }
},
```

Text handling in the config

Tokenize: keep letters and numbers together, separate otherwise

```
"Hello, world"
→ ["Hello", ",", "world"]
```

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
"train_data_path": "data/train_5000.jsonl",
"validation_data_path": "data/dev_500.jsonl",
"model": {
    "type": "simple_classifier",
    "embedder": {
        "token_embedders": {
            "tokens": {
                "type": "embedding",
                "embedding_dim": 50
            }
        }
    },
    "encoder": {
        "type": "bag_of_embeddings",
        "embedding_dim": 50
    }
},
```

Index: for the “tokens” key, represent each token as a single integer

```
["Hello", ",", "world"]
→ torch.tensor([101, 2, 729])
```

Text handling in the config

Tokenize: keep letters and numbers together, separate otherwise

```
"Hello, world"
→ ["Hello", ",", "world"]
```

Index: for the “tokens” key, represent each token as a single integer

```
["Hello", ",", "world"]
→ torch.tensor([101, 2, 729])
```

Embed: for the “tokens” key, get embeddings by projecting them into a 50d space

```
torch.tensor([101, 2, 729])
→ torch.tensor(
      [[0.6645, ..., 0.0155], ...]
  )
```

Example: word and character embeddings

- Add a `"chars"` key to the indexer and embedder

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        },
        "chars": {
            "type": "characters",
            "min_padding_length": 1
        }
    },
    "tokenizer": "letters_digits"
},
```

```
"model": {
    "type": "simple_classifier",
    "embedder": {
        "token_embedders": {
            "tokens": {
                "type": "embedding",
                "embedding_dim": 50
            },
            "chars": {
                "type": "character_encoding",
                "embedding": {
                    "embedding_dim": 10,
                    "vocab_namespace": "token_characters"
                },
                "encoder": {
                    "type": "lstm",
                    "input_size": 10,
                    "hidden_size": 5,
                    "bidirectional": true
                }
            }
        }
    },
    "encoder": {
        "type": "bag_of_embeddings",
        "embedding_dim": 60
    }
},
```

Example: word and character embeddings

- Add a `"chars"` key to the indexer and embedder
- Make sure characters are in a separate vocab namespace

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        },
        "chars": {
            "type": "characters",
            "min_padding_length": 1
        }
    },
    "tokenizer": "letters_digits"
},
```

```
"model": {
    "type": "simple_classifier",
    "embedder": {
        "token_embedders": {
            "tokens": {
                "type": "embedding",
                "embedding_dim": 50
            },
            "chars": {
                "type": "character_encoding",
                "embedding": {
                    "embedding_dim": 10,
                    "vocab_namespace": "token_characters"
                },
                "encoder": {
                    "type": "lstm",
                    "input_size": 10,
                    "hidden_size": 5,
                    "bidirectional": true
                }
            }
        }
    },
    "encoder": {
        "type": "bag_of_embeddings",
        "embedding_dim": 60
    }
},
```

Example: word and character embeddings

- Add a `"chars"` key to the indexer and embedder
- Make sure characters are in a separate vocab namespace
- Update encoder dimensions

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        },
        "chars": {
            "type": "characters",
            "min_padding_length": 1
        }
    },
    "tokenizer": "letters_digits"
},
```

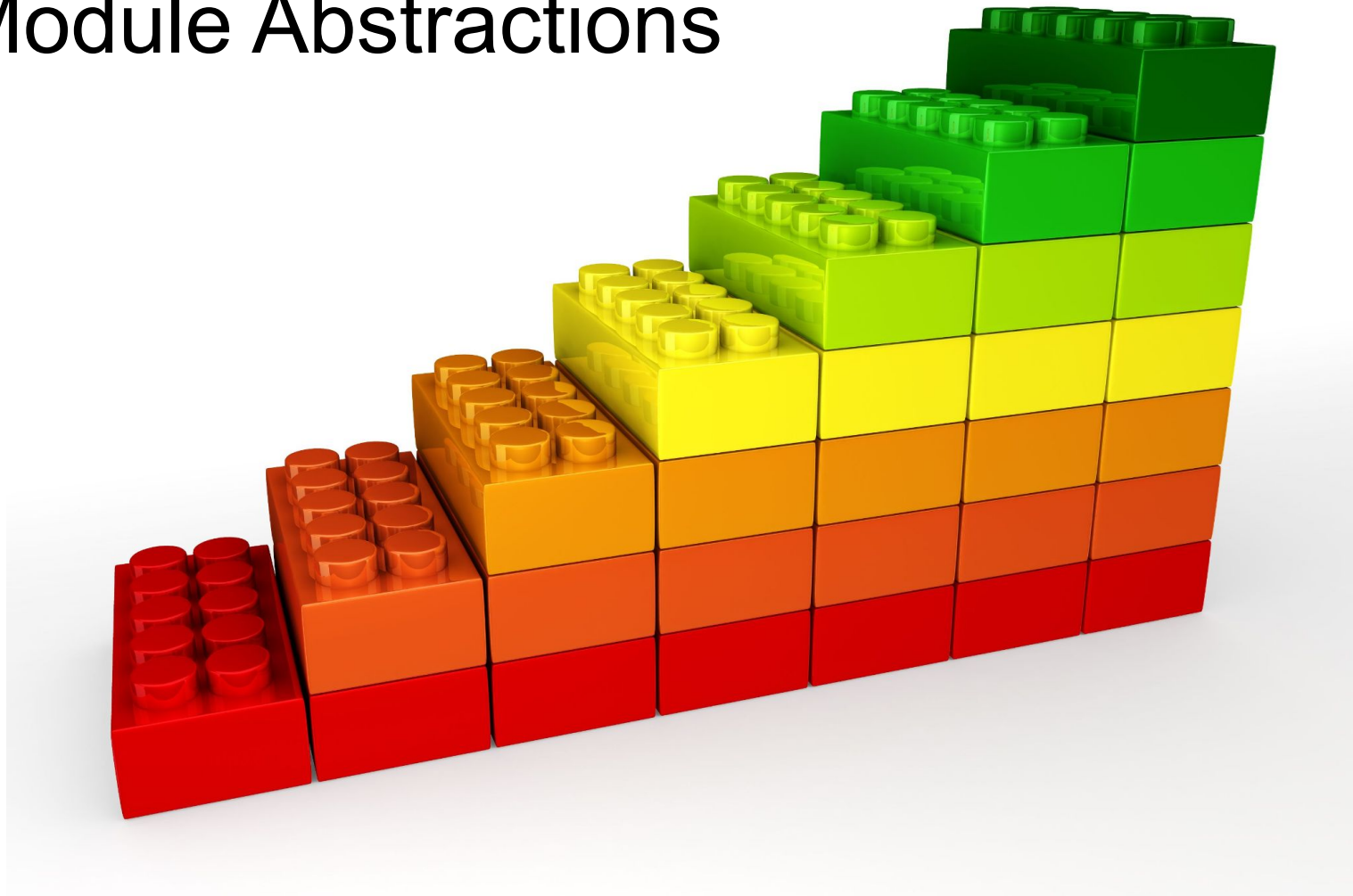
```
"model": {
    "type": "simple_classifier",
    "embedder": {
        "token_embedders": {
            "tokens": {
                "type": "embedding",
                "embedding_dim": 50
            },
            "chars": {
                "type": "character_encoding",
                "embedding": {
                    "embedding_dim": 10,
                    "vocab_namespace": "token_characters"
                },
                "encoder": {
                    "type": "lstm",
                    "input_size": 10,
                    "hidden_size": 5,
                    "bidirectional": true
                }
            }
        }
    },
    "encoder": {
        "type": "bag_of_embeddings",
        "embedding_dim": 60
    }
},
```

Warning: tokenizer/indexer/embedder incompatibilities

- There are many ways to tokenize/index/embed in NLP
- Many of them are not compatible
  - E.g. Transformer indexer with a non-Transformer tokenizer
  - See: https://guide.allennlp.org/representing-text-as-features#5
- If you are unsure, look at other models' configs.

Exercise Period 4

Module Abstractions



Config reading

- Recall: when beginning training, objects are constructed based on the config
- But how do you get the Python class **YelpReviewJsonLinesReader** from `"yelp-review-jsonl"`?

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
```

```
@DatasetReader.register("yelp-review-jsonl")
class YelpReviewJsonLinesReader(DatasetReader):
    def __init__(
        self,
        tokenizer: Tokenizer = None,
        token_indexers: Dict[str, TokenIndexer] = None,
        **kwargs
    ):
```

Registries

```python
@DatasetReader.register("yelp-review-jsonl")
class YelpReviewJsonLinesReader(DatasetReader):
    def __init__(
        self,
        tokenizer: Tokenizer = None,
        token_indexers: Dict[str, TokenIndexer] = None,
        **kwargs
    ):
```

- Answer: **Registry** system
  - **YelpReviewJsonLinesReader** extends **DatasetReader**
  - Call to **DatasetReader.register** associates `"yelp-review-jsonl"` with the class
- AllenNLP module classes have a **.register** function
  (internally: from the **Registrable** class)

```
In[13]: from pprint import pprint
In[14]: from allennlp.common.registrable import Registrable
In[15]: from allennlp.data.dataset_readers import DatasetReader
In[16]: pprint(Registrable._registry[DatasetReader], width=120)
{'babi': (<class 'allennlp.data.dataset_readers.babi.BabiReader'>, None),
 'conll2003': (<class 'allennlp.data.dataset_readers.conll2003.Conll2003DatasetReader'>, None),
 'interleaving': (<class 'allennlp.data.dataset_readers.interleaving_dataset_reader.InterleavingDatasetReader'>, None),
 'multitask': (<class 'allennlp.data.dataset_readers.multitask.MultiTaskDatasetReader'>, None),
 'multitask_shim': (<class 'allennlp.data.dataset_readers.multitask._MultitaskDatasetReaderShim'>, None),
 'sequence_tagging': (<class 'allennlp.data.dataset_readers.sequence_tagging.SequenceTaggingDatasetReader'>, None),
 'sharded': (<class 'allennlp.data.dataset_readers.sharded_dataset_reader.ShardedDatasetReader'>, None),
 'text_classification_json': (<class 'allennlp.data.dataset_readers.text_classification_json.TextClassificationJsonReader'>,
                              None),
 'yelp-review-jsonl': (<class 'simple_classifier.dataset_reader.YelpReviewJsonLinesReader'>, None)}
```

Registries and config interpretation

- Registry key is provided under `"type"`
  - If none provided, fall back to default impl.
- Registry keys are unique only relative to a given module
  - Type hint in the constructor used to know which registry to look in

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
```

```
@DatasetReader.register("yelp-review-jsonl")
class YelpReviewJsonLinesReader(DatasetReader):
    def __init__(
        self,
        tokenizer: Tokenizer = None,
        token_indexers: Dict[str, TokenIndexer] = None,
        **kwargs
    ):
```

```
@TokenIndexer.register("single_id")
class SingleIdTokenIndexer(TokenIndexer):
```

```
@Tokenizer.register("letters_digits")
class LettersDigitsTokenizer(Tokenizer):
```

```
In[18]: from allennlp.data.tokenizers import Tokenizer
In[19]: pprint(Registrable._registry[Tokenizer], width=120)
{'character': (<class 'allennlp.data.tokenizers.character_tokenizer.CharacterTokenizer'>, None),
 'just_spaces': (<class 'allennlp.data.tokenizers.whitespace_tokenizer.WhitespaceTokenizer'>, None),
 'letters_digits': (<class 'allennlp.data.tokenizers.letters_digits_tokenizer.LettersDigitsTokenizer'>, None),
 'pretrained_transformer': (<class 'allennlp.data.tokenizers.pretrained_transformer_tokenizer.PretrainedTransformerTokenizer'>,
                            None),
 'spacy': (<class 'allennlp.data.tokenizers.spacy_tokenizer.SpacyTokenizer'>, None),
 'whitespace': (<class 'allennlp.data.tokenizers.whitespace_tokenizer.WhitespaceTokenizer'>, None)}
In[20]: Tokenizer.default_implementation
Out[20]: 'spacy'
```

Registries and config interpretation

- Registry key is provided under `"type"`
  - If none provided, fall back to default impl.
- Registry keys are unique only relative to a given module
  - Type hint in the constructor used to know which registry to look in
- Objects are constructed bottom-up
  - Look at constructor args and types
  - Saturate constructor args with values from config
  - Default value otherwise
  - (Detail: **FromParams** class powers this)

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
```

```
@DatasetReader.register("yelp-review-jsonl")
class YelpReviewJsonLinesReader(DatasetReader):
    def __init__(
        self,
        tokenizer: Tokenizer = None,
        token_indexers: Dict[str, TokenIndexer] = None,
        **kwargs
    ):
```

```
@TokenIndexer.register("single_id")
class SingleIdTokenIndexer(TokenIndexer):
```

```
@Tokenizer.register("letters_digits")
class LettersDigitsTokenizer(Tokenizer):
```

Config example: **dataset_reader**

1. Look up `"yelp-review-jsonl"` in the **DatasetReader** registry, find **YelpReviewJsonLinesReader**

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
```

```
@DatasetReader.register("yelp-review-jsonl")
class YelpReviewJsonLinesReader(DatasetReader):
    def __init__(
        self,
        tokenizer: Tokenizer = None,
        token_indexers: Dict[str, TokenIndexer] = None,
        **kwargs
    ):
```

Config example: **dataset_reader**

1. Look up `"yelp-review-jsonl"` in the **DatasetReader** registry, find **YelpReviewJsonLinesReader**
2. Note two constructor arguments with type hints

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
```

```
@DatasetReader.register("yelp-review-jsonl")
class YelpReviewJsonLinesReader(DatasetReader):
    def __init__(
        self,
        tokenizer: Tokenizer = None,
        token_indexers: Dict[str, TokenIndexer] = None,
        **kwargs
    ):
```

Config example: **dataset_reader**

1. Look up `"yelp-review-jsonl"` in the **DatasetReader** registry, find **YelpReviewJsonLinesReader**
2. Note two constructor arguments with type hints
3. Interpret `"token_indexers"`
   a. We find a constructor argument with the same name
   b. Find **SingleIdTokenIndexer** in the **TokenIndexer** registry
   c. Make a dict with a single pair: `"type"` as key, **SingleIdTokenIndexer** instance as value

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
```

```
@DatasetReader.register("yelp-review-jsonl")
class YelpReviewJsonLinesReader(DatasetReader):
    def __init__(
        self,
        tokenizer: Tokenizer = None,
        token_indexers: Dict[str, TokenIndexer] = None,
        **kwargs
    ):
```

```
@TokenIndexer.register("single_id")
class SingleIdTokenIndexer(TokenIndexer):
```

Config example: **dataset_reader**

1. Look up `"yelp-review-jsonl"` in the **DatasetReader** registry, find **YelpReviewJsonLinesReader**
2. Note two constructor arguments with type hints
3. Interpret `"token_indexers"`
   a. We find a constructor argument with the same name
   b. Find **SingleIdTokenIndexer** in the **TokenIndexer** registry
   c. Make a dict with a single pair: `"type"` as key, **SingleIdTokenIndexer** instance as value
4. Interpret `"tokenizer"`
   a. Note: `"letters_digits"` = `{"type": "letters_digits"}`
   b. We find a constructor argument with the same name
   c. Find **LettersDigitsTokenizer** in the **Tokenizer** registry

```
"dataset_reader": {
    "type": "yelp-review-jsonl",
    "token_indexers": {
        "tokens": {
            "type": "single_id"
        }
    },
    "tokenizer": "letters_digits"
},
```

```
@DatasetReader.register("yelp-review-jsonl")
class YelpReviewJsonLinesReader(DatasetReader):
    def __init__(
        self,
        tokenizer: Tokenizer = None,
        token_indexers: Dict[str, TokenIndexer] = None,
        **kwargs
    ):
```

```
@TokenIndexer.register("single_id")
class SingleIdTokenIndexer(TokenIndexer):
```

```
@Tokenizer.register("letters_digits")
class LettersDigitsTokenizer(Tokenizer):
```

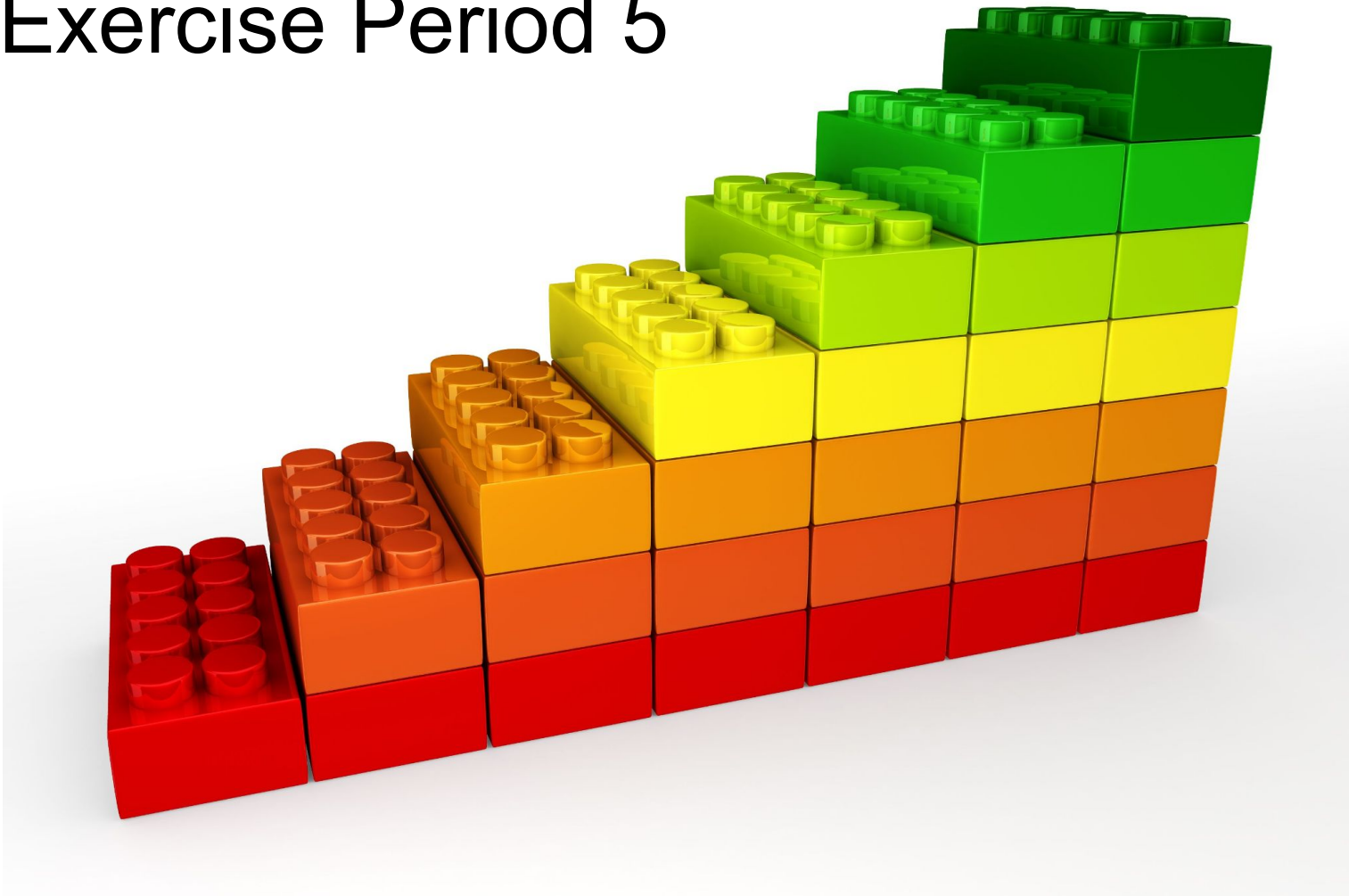
The Power of Registries

- Any class that inherits from **Registrable** can be used in configs in this way
  - Most AllenNLP classes are Registrables
- Implementing a **Registrable** class is a deal
  - You agree to perform certain functions
    - A **Tokenizer** subclass needs to implement **tokenizer**
  - In exchange, you may use your subclass anywhere the superclass is called for

The Power of Registries

- You can also create your own **Registrable** class
- Registries facilitate easy interchangeability of modules
  - If working on a new kind of embedding: make many configs, each with different encoders, compare results
  - “Dummy”/“no-op” implementations for ablations (e.g. **PassThroughEncoder**)

Exercise Period 5



Conclusion

What now?

- You have everything you need to start writing and reading AllenNLP code
- Run through the official guide: guide.allennlp.org
  - We have roughly followed it here, but skipped some sections
- Get to know the (great) documentation site: docs.allennlp.org
- Use your debugger to step through programs
- See **allennlp-models** for great, clean implementations of popular models: github.com/allenai/allennlp-models
- Ask questions on Stack Overflow: stackoverflow.com/questions/tagged/allennlp
- Ask me questions at lg876@georgetown.edu

Advanced topics

- Multitask Learning: https://docs.allennlp.org/main/api/models/multitask/
- Automatic hyperparameter tuning: https://github.com/allenai/allentune
- Model demos: https://github.com/allenai/allennlp-server
- Model gallery: https://gallery.allennlp.org/

Reminder: further reading

- PyTorch: Howard and Gugger, *Deep Learning for Coders with Fastai and PyTorch*
- NLP, intro: Jurafsky and Martin, *Speech and Language Processing*
- NLP advanced: Goldberg and Hirst: *Neural Network Methods in Natural Language Processing*