

Natural Language Understanding (NLU)

A

B

Natural Language Generation (NLG)

Lexical Ambiguity

1

Text Planning

1

Syntactic Ambiguity

2

Sentence Planning

2

Semantic Ambiguity

3

Anaphoric Ambiguity

4

Realization

3

Writing Code for NLP



ALLEN INSTITUTE
for ARTIFICIAL INTELLIGENCE

Who we are

Matt Gardner ([@nlpmattg](#))

Matt is a research scientist on AllenNLP. He was the original architect of AllenNLP, and he co-hosts the NLP Highlights podcast.

Mark Neumann ([@markneumannnn](#))

Mark is a research engineer on AllenNLP. He helped build AllenNLP and its precursor DeepQA with Matt, and has implemented many of the models in the demos.

Joel Grus ([@joelgrus](#))

Joel is a research engineer on AllenNLP, although you may know him better from "I Don't Like Notebooks" or from "Fizz Buzz in Tensorflow" or from his book *Data Science from Scratch*.

Outline

- How to write code when prototyping
- Developing good processes

BREAK

- How to write reusable code for NLP
- Case Study: A Part-of-Speech Tagger
- Sharing Your Research

**What we expect you
know already**

What we expect you know already

modern (neural) NLP

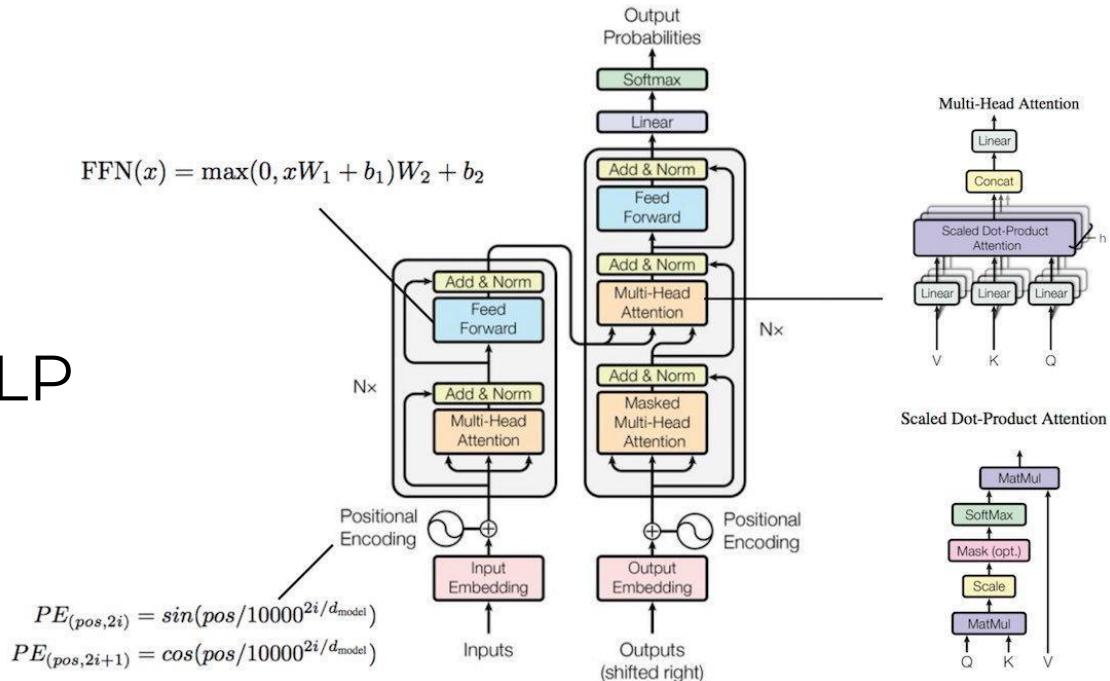


Figure 1: The Transformer - model architecture.

What we expect you know already

```
# handle the download
async for chunk in get_bytes():
    file_hash.|  
    file.w  
  
print(f'Downloaded {de  
  
ef main():  
    # get the URL from the  
    parser = argparse.ArgumentParser()  
    parser.add_argument('url')  
    arguments = parser.parse_args()  
  
    # get the filename from the URL
    url_parts = urls.split(arguments.url)
    file_name = url_parts[-1]  
  
    # start the download
    loop = asyncio.get_event_loop()
    loop.run_until_complete(download_url(arguments.url, file_name))
```



SHA256Type

SHA256Type

SHA256Type

SHA256Type

SHA256Type

SHA256Type

object

object

object

object

Ctrl+Down and Ctrl+Up will move caret down and up in the editor >> π

Python

What we expect you know already

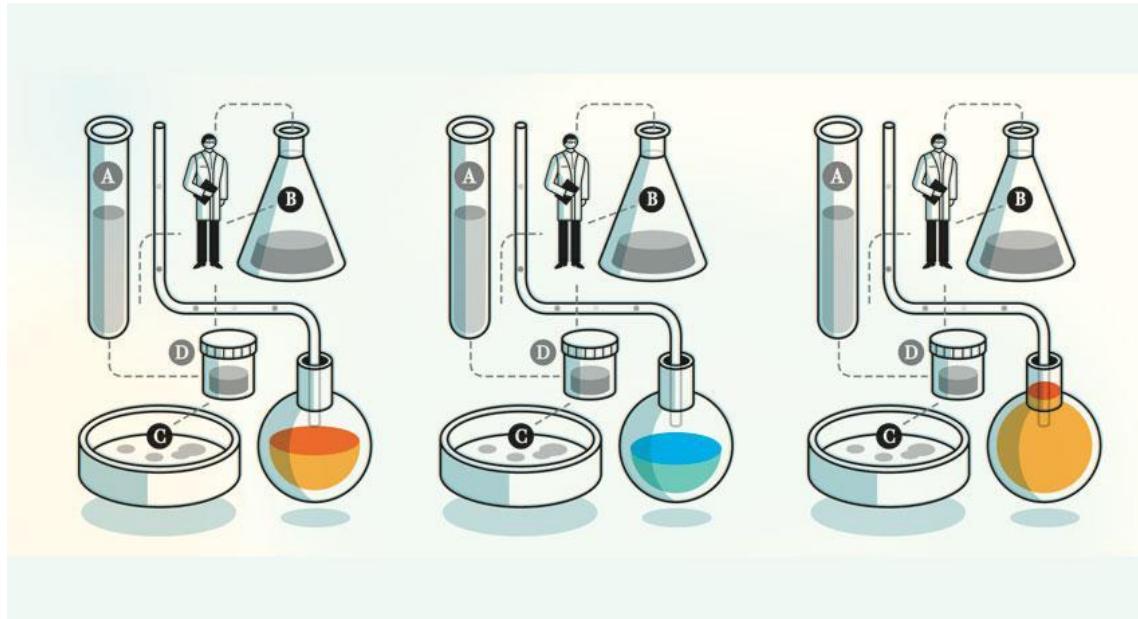
the difference between good science and bad science



**What you'll learn
today**

What you'll learn today

how to write code in a way that facilitates good science and reproducible experiments



What you'll learn today

how to write code in a way that makes your life easier



The Elephant in the Room: AllenNLP

- This is not a tutorial *about* AllenNLP
- But (obviously, seeing as we wrote it) AllenNLP represents our experiences and opinions about how best to write research code
- Accordingly, we'll use it in most of our examples
- And we hope you'll come out of this tutorial wanting to give it a try
- But our goal is that you find the tutorial useful even if you never use AllenNLP

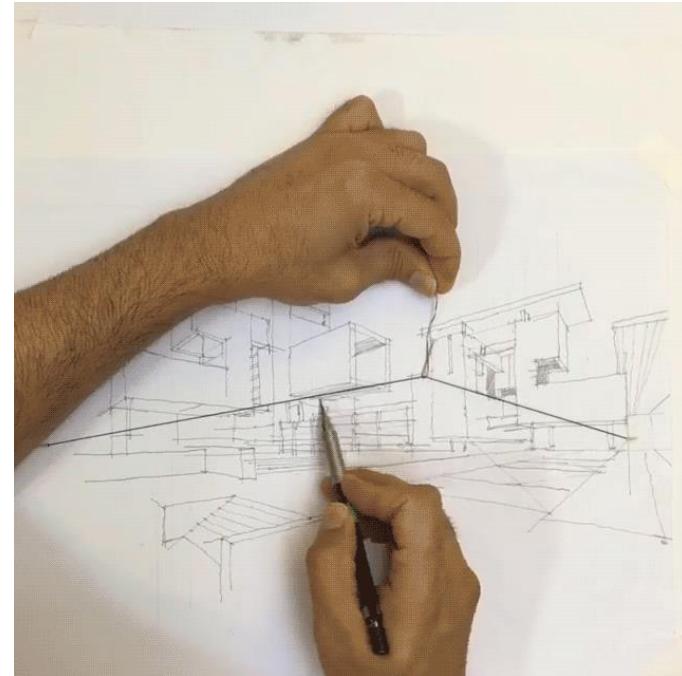


Two modes of writing research code

1: prototyping



2: writing components



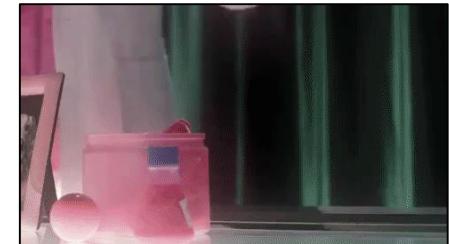
Prototyping New Models

Main goals during prototyping

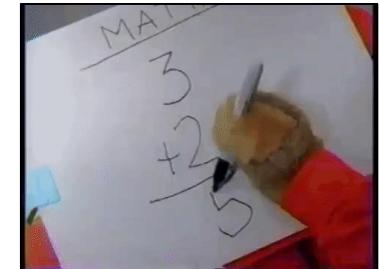
- Write code quickly



- Run experiments, keep track of what you tried



- Analyze model behavior - did it do what you wanted?



Main goals during prototyping

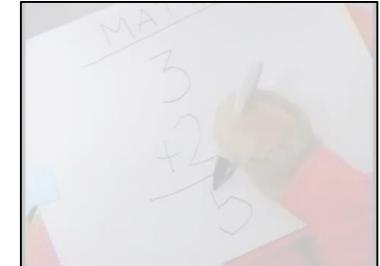
- **Write code quickly**



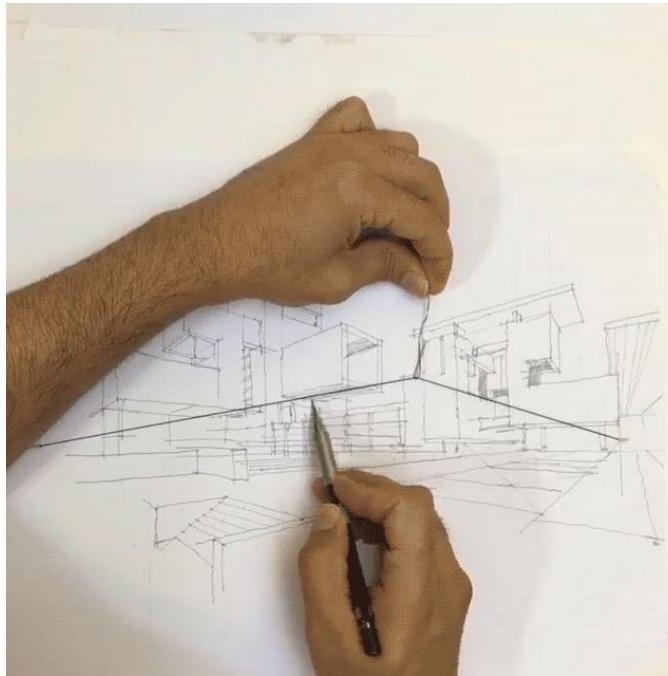
- Run experiments, keep track of what you tried



- Analyze model behavior - did it do what you wanted?



Writing code quickly - Use a framework!



Writing code quickly - Use a framework!

- Training loop?

Writing code quickly - Use a framework!

- Training loop?

```
model = LSTMTagger(EMBEDDING_DIM, HIDDEN_DIM,
                    len(word_to_ix), len(tag_to_ix))
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)
validation_losses = []
patience = 10
for epoch in range(1000):
    training_loss = 0.0
    validation_loss = 0.0

    for dataset, training in [(training_data, True),
                               (validation_data, False)]:
        correct = total = 0
        torch.set_grad_enabled(training)
        t = tqdm.tqdm(dataset)
        for i, (sentence, tags) in enumerate(t):
            model.zero_grad()
            model.hidden = model.init_hidden()

            sentence_in = prepare_sequence(sentence, word_to_ix)
            targets = prepare_sequence(tags, tag_to_ix)

            tag_scores = model(sentence_in)

            loss = loss_function(tag_scores, targets)
```

```
predictions = tag_scores.max(-1)[1]
correct += (predictions == targets).sum().item()
total += len(targets)
accuracy = correct / total

if training:
    loss.backward()
    training_loss += loss.item()
    t.set_postfix(training_loss=training_loss/(i + 1),
                  accuracy=accuracy)
    optimizer.step()
else:
    validation_loss += loss.item()
    t.set_postfix(validation_loss=validation_loss/(i + 1),
                  accuracy=accuracy)

validation_losses.append(validation_loss)

if (patience and
    len(validation_losses) >= patience and
    validation_losses[-patience] ==
        min(validation_losses[-patience:])):
    print("patience reached, stopping early")
    break
```

Writing code quickly - Use a framework!

- Tensorboard logging?
 - Model checkpointing?
 - Complex data processing, with smart batching?
 - Computing span representations?
 - Bi-directional attention matrices?
-
- Easily thousands of lines of code!

Writing code quickly - Use a framework!

- Don't start from scratch! Use someone else's components.

Writing code quickly - Use a framework!

- But...



Writing code quickly - Use a framework!

- But...



- Make sure you can bypass the abstractions when you need to

Writing code quickly - Get a good starting place

Writing code quickly - Get a good starting place

- First step: get a baseline running



- This is good research practice, too

Writing code quickly - Get a good starting place

- Could be someone else's code... as long as you can read it

```
with tf.variable_scope("char"):  
    Acx = tf.nn.embedding_lookup(char_emb_mat, self.cx) # [N, M, JX, W, dc]  
    Acq = tf.nn.embedding_lookup(char_emb_mat, self.cq) # [N, JQ, W, dc]  
    Acx = tf.reshape(Acx, [-1, JX, W, dc])  
    Acq = tf.reshape(Acq, [-1, JQ, W, dc])  
  
    filter_sizes = list(map(int, config.out_channel_dims.split(',')))  
    heights = list(map(int, config.filter_heights.split(',')))  
    assert sum(filter_sizes) == dco  
    with tf.variable_scope("conv"):  
        xx = multi_conv1d(Acx, filter_sizes, heights, "VALID", self.is_train, config.keep_prob, scope="xx")  
        if config.share_cnn_weights:  
            tf.get_variable_scope().reuse_variables()  
            qq = multi_conv1d(Acq, filter_sizes, heights, "VALID", self.is_train, config.keep_prob, scope="xx")  
        else:  
            qq = multi_conv1d(Acq, filter_sizes, heights, "VALID", self.is_train, config.keep_prob, scope="qq")  
        xx = tf.reshape(xx, [-1, M, JX, dco])  
        qq = tf.reshape(qq, [-1, JQ, dco])
```

Writing code quickly - Get a good starting place

- Could be someone else's code... as long as you can read it

```
encoded_question = self._dropout(self._phrase_layer(embedded_question, question_lstm_mask))
encoded_passage = self._dropout(self._phrase_layer(embedded_passage, passage_lstm_mask))
encoding_dim = encoded_question.size(-1)

# Shape: (batch_size, passage_length, question_length)
passage_question_similarity = self._matrix_attention(encoded_passage, encoded_question)
# Shape: (batch_size, passage_length, question_length)
passage_question_attention = util.masked_softmax(passage_question_similarity, question_mask)
# Shape: (batch_size, passage_length, encoding_dim)
passage_question_vectors = util.weighted_sum(encoded_question, passage_question_attention)

# We replace masked values with something really negative here, so they don't affect the
# max below.
masked_similarity = util.replace_masked_values(passage_question_similarity,
                                                question_mask.unsqueeze(1),
                                                -1e7)
```

Writing code quickly - Get a good starting place

- Even better if this code already modularizes what you want to change

```
def __init__(self, vocab: Vocabulary,  
             text_field_embedder: TextFieldEmbedder, ← Add ELMo / BERT here  
             num_highway_layers: int,  
             phrase_layer: Seq2SeqEncoder,  
             similarity_function: SimilarityFunction,  
             modeling_layer: Seq2SeqEncoder,  
             span_end_encoder: Seq2SeqEncoder,  
             dropout: float = 0.2,  
             mask_lstms: bool = True,  
             initializer: InitializerApplicator = InitializerApplicator(),  
             regularizer: Optional[RegularizerApplicator] = None) -> None:
```

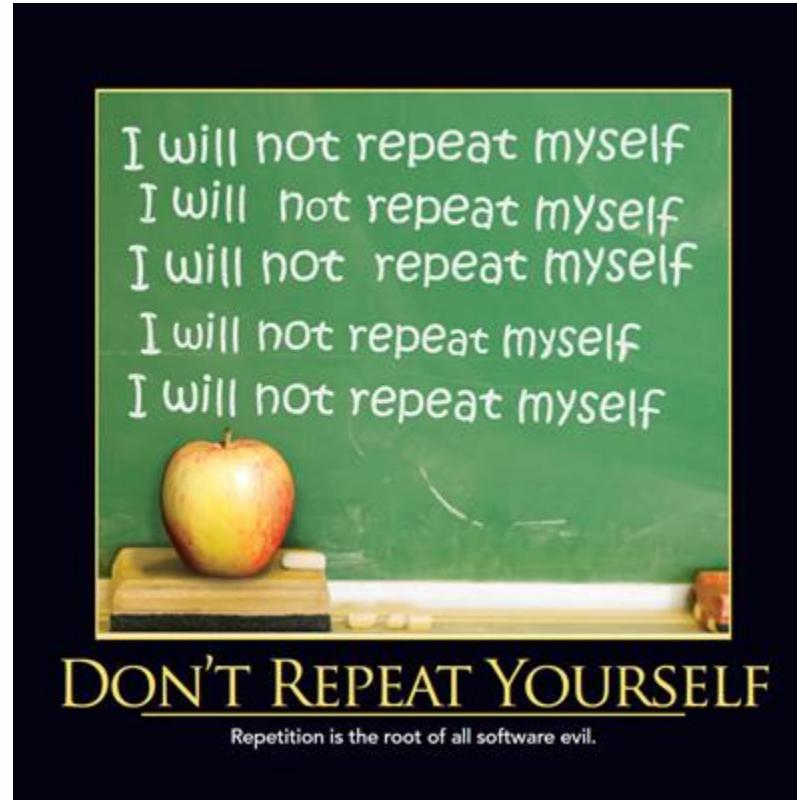
Writing code quickly - Get a good starting place



- Re-implementing a SOTA baseline is incredibly helpful for understanding what's going on, and where some decisions might have been made better

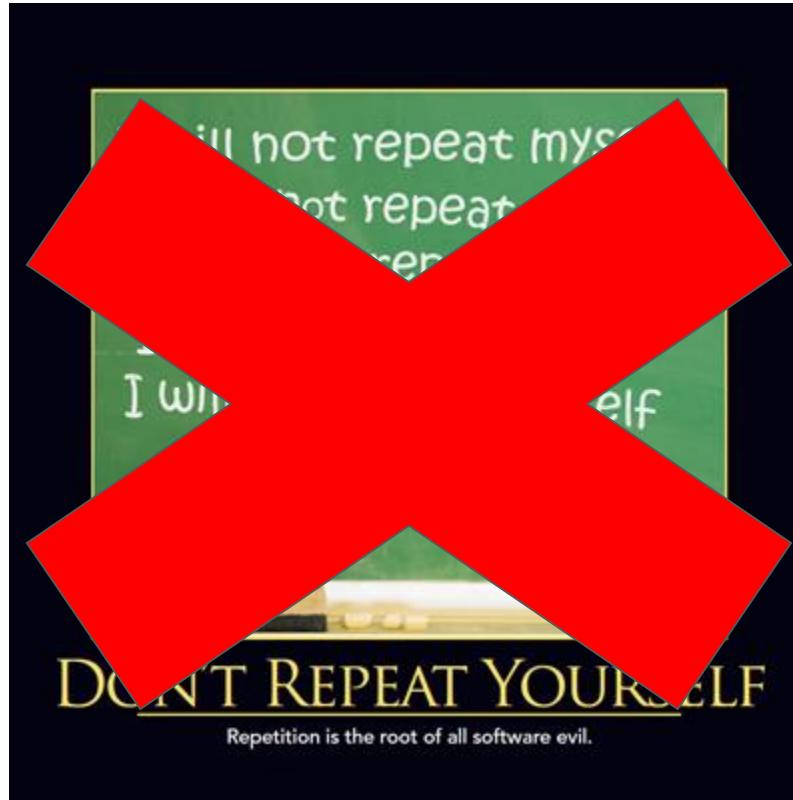
Writing code quickly - Copy first, refactor later

- CS degree:



Writing code quickly - Copy first, refactor later

- CS degree:



Writing code quickly - Copy first, refactor later

- CS degree:



We're prototyping! Just go fast and find something that works, *then* go back and refactor (if you made something useful)



Writing code quickly - Copy first, refactor later

- Really bad idea: using inheritance to share code for related models

```
class MemoryNetwork(TextTrainer):  
  
    class SoftmaxMemoryNetwork(MemoryNetwork):  
  
        class MultipleTrueFalseMemoryNetwork(MemoryNetwork):  
  
            class MultipleTrueFalseSimilarity(MultipleTrueFalseMemoryNetwork):  
  
                class MultipleTrueFalseDecomposableAttention(MultipleTrueFalseMemoryNetwork):  
  
                    class QuestionAnswerMemoryNetwork(MemoryNetwork):
```

- Instead: just copy the code, figure out how to share later, if it makes sense

Writing code quickly - *Do* use good code style

- CS degree:

```
/**  
 * Code Readability  
 */  
if (readable()) {  
    be_happy();  
} else {  
    refactor();  
}
```

Writing code quickly - *Do* use good code style

- CS degree:

```
/**  
 * Code Readability  
 */  
if (readable()) {  
    be_happy();  
} else {  
    refactor();  
}
```



Writing code quickly - *Do use good code style*

```
with tf.variable_scope("char"):  
    Acx = tf.nn.embedding_lookup(char_emb_mat, self.cx) # [N, M, JX, W, dc]  
    Acq = tf.nn.embedding_lookup(char_emb_mat, self.cq) # [N, JQ, W, dc]  
    Acx = tf.reshape(Acx, [-1, JX, W, dc])  
    Acq = tf.reshape(Acq, [-1, JQ, W, dc])  
  
    filter_sizes = list(map(int, config.out_channel_dims.split(',')))  
    heights = list(map(int, config.filter_heights.split(',')))  
    assert sum(filter_sizes) == dco  
    with tf.variable_scope("conv"):  
        xx = multi_conv1d(Acx, filter_sizes, heights, "VALID", self.is_train, config.keep_prob, scope="xx")  
        if config.share_cnn_weights:  
            tf.get_variable_scope().reuse_variables()  
            qq = multi_conv1d(Acq, filter_sizes, heights, "VALID", self.is_train, config.keep_prob, scope="xx")  
        else:  
            qq = multi_conv1d(Acq, filter_sizes, heights, "VALID", self.is_train, config.keep_prob, scope="qq")  
        xx = tf.reshape(xx, [-1, M, JX, dco])  
        qq = tf.reshape(qq, [-1, JQ, dco])
```

Writing code quickly - *Do* use good code style

```
with tf.variable_scope("char"):  
    Acx = tf.nn.embedding_lookup(char_emb_mat, self.cx) # [N, M, JX, W, dc]  
    Acq = tf.nn.embedding_lookup(char_emb_mat, self.cq) # [N, JQ, W, dc]  
    Acx = tf.reshape(Acx, [-1, JX, W, dc])  
    Acq = tf.reshape(Acq, [-1, JQ, W, dc])  
  
    filter_sizes = list(map(int, config.out_channel_dims.split(',')))  
    heights = list(map(int, config.filter_heights.split(',')))  
    assert sum(filter_sizes) == dco  
    with tf.variable_scope('conv'):  
        xx = multi_conv1d(Acx, filter_sizes, heights, "VALID", self.is_train, config.keep_prob, scope="xx")  
        if config.share_cnn_weights:  
            tf.get_variable_scope().reuse_variables()  
            qq = multi_conv1d(Acq, filter_sizes, heights, "VALID", self.is_train, config.keep_prob, scope="xx")  
        else:  
            qq = multi_conv1d(Acq, filter_sizes, heights, "VALID", self.is_train, config.keep_prob, scope="qq")  
        xx = tf.reshape(xx, [-1, M, JX, dco])  
        qq = tf.reshape(qq, [-1, JQ, dco])
```

Writing code quickly - *Do use good code style*

```
encoded_question = self._dropout(self._phrase_layer(embedded_question, question_lstm_mask))
encoded_passage = self._dropout(self._phrase_layer(embedded_passage, passage_lstm_mask))
encoding_dim = encoded_question.size(-1)

# Shape: (batch_size, passage_length, question_length)
passage_question_similarity = self._matrix_attention(encoded_passage, encoded_question)
# Shape: (batch_size, passage_length, question_length)
passage_question_attention = util.masked_softmax(passage_question_similarity, question_mask)
# Shape: (batch_size, passage_length, encoding_dim)
passage_question_vectors = util.weighted_sum(encoded_question, passage_question_attention)

# We replace masked values with something really negative here, so they don't affect the
# max below.
masked_similarity = util.replace_masked_values(passage_question_similarity,
                                                question_mask.unsqueeze(1),
                                                -1e7)
```

Writing code quickly - *Do use good code style*

```
encoded_question = self._dropout(self._phrase_layer(embedded_question, question_lstm))
encoded_passage = self._dropout(self._phrase_layer(embedded_passage, passage_lstm))
encoding_dim = encoded_question.size(-1)
```

Meaningful names

```
# Shape: (batch_size, passage_length, question_length)
passage_question_similarity = self._matrix_attention(encoded_passage, encoded_question)
# Shape: (batch_size, passage_length, question_length)
passage_question_attention = util.masked_softmax(passage_question_similarity, question_mask)
# Shape: (batch_size, passage_length, encoding_dim)
passage_question_vectors = util.weighted_sum(encoded_question, passage_question_attention)
```

```
# We replace masked values with something really negative here, so they don't affect the
# max below.
masked_similarity = util.replace_masked_values(passage_question_similarity,
                                                question_mask.unsqueeze(1),
                                                -1e7)
```

Writing code quickly - *Do* use good code style

```
encoded_question = self._dropout(self._phrase_layer(embedded_question, question_lstm_mask))
encoded_passage = self._dropout(self._phrase_layer(embedded_passage, passage_lstm_mask))
encoding_dim = encoded_question.size(-1)
```

```
# Shape: (batch_size, passage_length, question_length)
passage_question_similarity = self._matrix_attention(encoded_passage, encoded_question)
```

```
# Shape: (batch_size, passage_length, question_length)
passage_question_attention = util.masked_softmax(passage_question_similarity, question_lstm_mask)
# Shape: (batch_size, passage_length, encoding_dim)
passage_question_vectors = util.weighted_sum(encoded_question, passage_question_attention)
```

```
# We replace masked values with something really negative here, so they don't affect the
# max below.
```

```
masked_similarity = util.replace_masked_values(passage_question_similarity,
                                               question_mask.unsqueeze(1),
                                               -1e7)
```

Shape comments on tensors

Writing code quickly - *Do* use good code style

```
encoded_question = self._dropout(self._phrase_layer(embedded_question, question_lstm_mask))
encoded_passage = self._dropout(self._phrase_layer(embedded_passage, passage_lstm_mask))
encoding_dim = encoded_question.size(-1)

# Shape: (batch_size, passage_length, question_length)
passage_question_similarity = self._matrix_attention(encoded_passage, encoded_question)
# Shape: (batch_size, passage_length, question_length)
passage_question_attention = util.masked_softmax(passage_question_similarity, question_mask)
# Shape: (batch_size, passage_length, encoding_dim)
passage_question_vectors = util.weighted_sum(encoded_question, passage_question_attention)
```

```
# We replace masked values with something really negative here, so they don't affect the
# max below.
```

```
masked_similarity = util.replace_masked_values(passage_question_similarity,
                                              question_mask.unsqueeze(1),
                                              -1e7)
```

Comments describing
non-obvious logic

Writing code quickly - *Do* use good code style

```
encoded_question = self._dropout(self._phrase_layer(embedded_question, question_lstm_mask))  
encoded_passage = self._dropout(self._phrase_layer(embedded_passage, passage_lstm_mask))  
encoding_dim = encoded_question.size(-1)
```

```
# Shape: (batch_size,  
passage_question_simil  
# Shape: (batch_size,  
passage_question_atten  
# Shape: (batch_size,  
passage_question_vectors
```

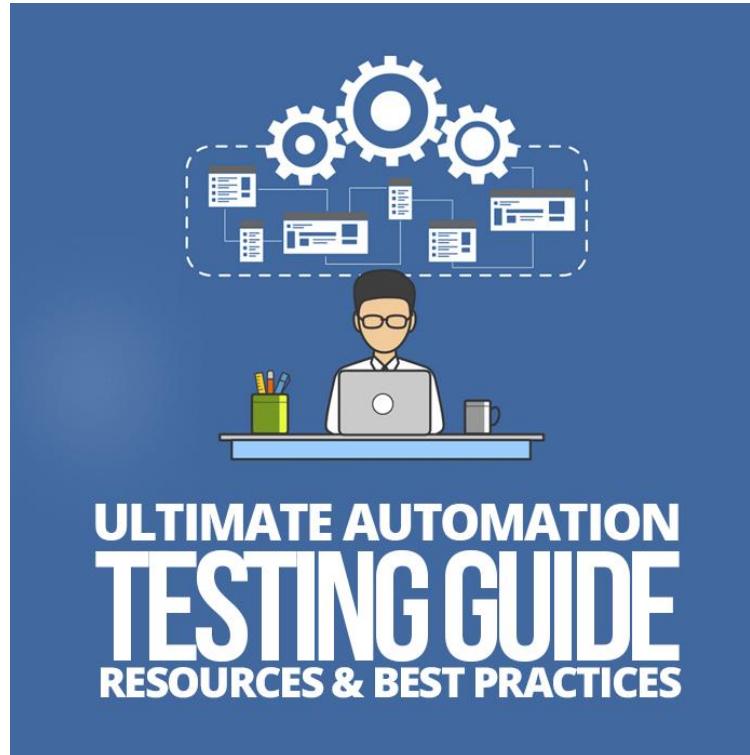
Write code for people,
not machines

```
estion)  
estion)  
attention)
```

```
# We replace masked values with something really negative here, so they don't affect the  
# max below.  
masked_similarity = util.replace_masked_values(passage_question_similarity,  
                                              question_mask.unsqueeze(1),  
                                              -1e7)
```

Writing code quickly - Minimal testing (but not *no* testing)

- CS degree:



Writing code quickly - Minimal testing (but not *no* testing)

- CS degree:



Writing code quickly - Minimal testing (but not *no* testing)

- A test that checks experimental behavior is a waste of time

```
encoded_question = self._dropout(self._phrase_layer(embedded_question, question_lstm_mask))
encoded_passage = self._dropout(self._phrase_layer(embedded_passage, passage_lstm_mask))
encoding_dim = encoded_question.size(-1)

# Shape: (batch_size, passage_length, question_length)
passage_question_similarity = self._matrix_attention(encoded_passage, encoded_question)
# Shape: (batch_size, passage_length, question_length)
passage_question_attention = util.masked_softmax(passage_question_similarity, question_mask)
# Shape: (batch_size, passage_length, encoding_dim)
passage_question_vectors = util.weighted_sum(encoded_question, passage_question_attention)

# We replace masked values with something really negative here, so they don't affect the
# max below.
masked_similarity = util.replace_masked_values(passage_question_similarity,
                                                question_mask.unsqueeze(1),
                                                -1e7)
```

Writing code quickly - Minimal testing (but not *no* testing)

- But, some parts of your code aren't experimental

```
class TestSquadReader:  
    @pytest.mark.parametrize("lazy", (True, False))  
    def test_read_from_file(self, lazy):  
        reader = SquadReader(lazy=lazy)  
        instances = ensure_list(reader.read(AllenNlpTestCase.FIXTURES_ROOT / 'data' / 'squad.json'))  
        assert len(instances) == 5  
  
        assert [t.text for t in instances[0].fields["question"].tokens[:3]] == ["To", "whom", "did"]  
        assert [t.text for t in instances[0].fields["passage"].tokens[:3]] == ["Architecturally", ",", "the"]  
        assert [t.text for t in instances[0].fields["passage"].tokens[-3:]] == ["of", "Mary", "."]  
        assert instances[0].fields["span_start"].sequence_index == 102  
        assert instances[0].fields["span_end"].sequence_index == 104  
  
        assert [t.text for t in instances[1].fields["question"].tokens[:3]] == ["What", "sits", "on"]  
        assert [t.text for t in instances[1].fields["passage"].tokens[:3]] == ["Architecturally", ",", "the"]  
        assert [t.text for t in instances[1].fields["passage"].tokens[-3:]] == ["of", "Mary", "."]  
        assert instances[1].fields["span_start"].sequence_index == 17  
        assert instances[1].fields["span_end"].sequence_index == 23
```

Writing code quickly - Minimal testing (but not *no* testing)

- And even experimental parts can have useful tests

```
class AtisSemanticParserTest(ModelTestCase):  
    def setUp(self):  
        super(AtisSemanticParserTest, self).setUp()  
        self.set_up_model(str(self.FIXTURES_ROOT / "semantic_parsing" / "atis" / "experiment.json"),  
                          str(self.FIXTURES_ROOT / "data" / "atis" / "sample.json"))  
  
    @flaky  
    def test_atis_model_can_train_save_and_load(self):  
        self.ensure_model_can_train_save_and_load(self.param_file)
```

Writing code quickly - Minimal testing (but not *no* testing)

- And even experimental parts can have useful tests

```
class AtisSemanticParserTest(ModelTest):
    def setUp(self):
        super(AtisSemanticParserTest, self).setUp()
        self.set_up_model(str(self.FIXTURES_ROOT / "atis"))
        str(self.FIXTURES_ROOT / "atis"))
```

Makes sure data processing works consistently, that tensor operations run, gradients are non-zero

```
@flaky
def test_atis_model_can_train_save_and_load(self):
    self.ensure_model_can_train_save_and_load(self.param_file)
```

Writing code quickly - Minimal testing (but not *no* testing)

- And even experimental parts can have useful tests

Run on small test fixtures, so debugging cycle is seconds, not minutes

```
class AtisSemanticParserTest(ModelTest):
    def setUp(self):
        super(AtisSemanticParserTest, self).setUp()
        self.set_up_model(str(self.FIXTURES_ROOT / "semantic_parsing" / "atis" / "experiment.json"),
                          str(self.FIXTURES_ROOT / "data" / "atis" / "sample.json"))

    @flaky
    def test_atis_model_can_train_save_and_load(self):
        self.ensure_model_can_train_save_and_load(self.param_file)
```

Writing code quickly - How much to hard-code?

- Which one should I do?

```
class MyModel(Model):
    def __init__(self):
        self.input_embedding = Embedding(100)
        self.encoder = LSTM(100, 200)
        self.my_novel_bits = ...
```

```
class MyModel(Model):
    def __init__(self,
                 input_embedding: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder):
        self.input_embedding = input_embedding
        self.encoder = encoder
        self.my_novel_bits = ...
```

Writing code quickly - How much to hard-code?

- Which one should I do?

```
class MyModel(Model):
    def __init__(self):
        self.input_embedding = Embedding(100)
        self.encoder = LSTM(100, 200)
        self.my_novel_bits = ...
```

I'm just prototyping! Why shouldn't I just hard-code an embedding layer?

```
class MyModel(Model):
    def __init__(self,
                 input_embedding: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder):
        self.input_embedding = input_embedding
        self.encoder = encoder
        self.my_novel_bits = ...
```

Writing code quickly - How much to hard-code?

- Which one should I do?

```
class MyModel(Model):
    def __init__(self):
        self.input_embedding = Embedding(100)
        self.encoder = LSTM(100, 200)
        self.my_novel_bits = ...
```

Why so abstract?

```
class MyModel(Model):
    def __init__(self,
                 input_embedding: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder):
        self.input_embedding = input_embedding
        self.encoder = encoder
        self.my_novel_bits = ...
```

Writing code quickly - How much to hard-code?

- Which one should I do?

```
class MyModel(Model):  
    def __init__(self):  
        self.input_embedding = Embedding(100)  
        self.encoder = LSTM(100, 200)  
        self.my_novel_bits = ...
```

```
class MyModel(Model):  
    def __init__(self,  
                 input_embedding: TextFieldEmbedder,  
                 encoder: Seq2SeqEncoder):  
        self.input_embedding = input_embedding  
        self.encoder = encoder  
        self.my_novel_bits = ...
```

On the parts that aren't what you're focusing on, you start simple. Later add ELMo, etc., *without rewriting your code.*

Writing code quickly - How much to hard-code?

- Which one should I do?

```
class MyModel(Model):
    def __init__(self):
        self.input_embedding = Embedding(100)
        self.encoder = LSTM(100, 200)
        self.my_novel_bits = ...
```

This also makes controlled experiments easier (both for you and for people who come after you).

```
class MyModel(Model):
    def __init__(self,
                 input_embedding: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder):
        self.input_embedding = input_embedding
        self.encoder = encoder
        self.my_novel_bits = ...
```

Writing code quickly - How much to hard-code?

- Which one should I do?

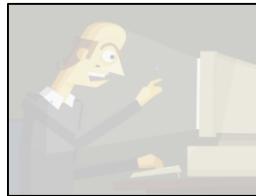
```
class MyModel(Model):
    def __init__(self):
        self.input_embedding = Embedding(100)
        self.encoder = LSTM(100, 200)
        self.my_novel_bits = ...
```

And it helps you think more clearly about the pieces of your model.

```
class MyModel(Model):
    def __init__(self,
                 input_embedding: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder):
        self.input_embedding = input_embedding
        self.encoder = encoder
        self.my_novel_bits = ...
```

Main goals during prototyping

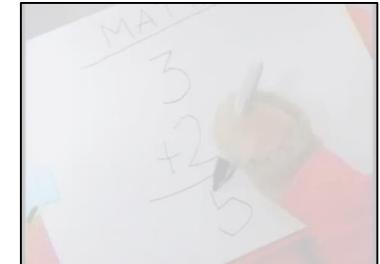
- Write code quickly



- **Run experiments, keep track of what you tried**



- Analyze model behavior - did it do what you wanted?



Running experiments - Keep track of what you ran

- You run a lot of stuff when you're prototyping, it can be hard to keep track of what happened when, and with what code



Running experiments - Keep track of what you ran

Experimenter	git SHA	Background Search Method	Model	Dataset	Train Acc	Validation Acc	Notes
Pradeep	fc8d6ca3	Lucene	QAMNS (50d)	Intermediate	0.3114	0.3045	patience=20
Pradeep	fc8d6ca3	Lucene	QAMNS (300d)	Intermediate	0.8317	0.3864	patience=20
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 50d	QAMNS (50d)	Intermediate	0.3008	0.35	patience=20
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 50d	QAMNS (300d)	Intermediate	0.7466	0.4227	patience=20
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 300d	QAMNS (50d)	Intermediate	0.3946	0.3591	patience=20
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 300d	QAMNS (300d)	Intermediate	0.7311	0.4227	patience=20
Pradeep	fc8d6ca3	BOW-LSH+IDF question+answers Glove 300d	QAMNS (300d)	Intermediate	0.7446	0.4227	patience=20
Pradeep	fc8d6ca3	BOW-LSH+IDF question+answers Paragram 300d	QAMNS (300d)	Intermediate	0.7853	0.3955	patience=20
Pradeep	fc8d6ca3	Lucene	QAMNS (300d)	SciQ	0.5551	0.571	patience=6
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 300d	QAMNS (300d)	SciQ	0.5434	0.524	patience=6

Running experiments - Keep track of what you ran

Experimenter	git SHA	Background Search	Training Acc	Validation Acc	Notes
Pradeep	fc8d6ca3	Lucene	0.3114	0.3045	patience=20
Pradeep	fc8d6ca3	Lucene	0.8317	0.3864	patience=20
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 50d	0.3008	0.35	patience=20
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 50d	0.7466	0.4227	patience=20
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 300d	0.3946	0.3591	patience=20
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 300d	0.7311	0.4227	patience=20
Pradeep	fc8d6ca3	BOW-LSH+IDF question+answers Glove 300d	0.7446	0.4227	patience=20
Pradeep	fc8d6ca3	BOW-LSH+IDF question+answers Paragram 300d	0.7853	0.3955	patience=20
Pradeep	fc8d6ca3	Lucene	0.5551	0.571	patience=6
Pradeep	fc8d6ca3	BOW-LSH question+answers Glove 300d	0.5434	0.524	patience=6

This is important!

Running experiments - Keep track of what you ran



- Currently in invite-only alpha; public beta coming soon
- <https://github.com/allenai/beaker>
- <https://beaker-pub.allenai.org>

Running experiments - Keep track of what you ran

The screenshot shows the Beaker interface, which is a platform for running scientific experiments. At the top, there's a navigation bar with the Allen Institute for Artificial Intelligence logo, the Beaker logo, and links for Home, Experiments, Datasets, Groups, and a user profile. A search bar at the top right contains the text "wikitable".

On the left, there are several filter and search options:

- Created by:**
 - Myself
 - Anyone
 - Another User
- From:**
 - Any time
- Additional Filters:**
 - Show Results
 - Show Uncommitted
- Actions:**
 - Reset Search

The main area displays a list of experiment runs, each with a title, description, and a timestamp indicating when it was committed or last modified. All entries appear to be committed, as indicated by the "5 months ago" timestamp.

Title	Description	Committed
wikitables_mml_model_iter_2		5 months ago
wikitables_preprocessed_er...		5 months ago
wikitables_dpd_and_erm_pr...		5 months ago
wikitables_dpd_and_erm_ou...		5 months ago
wikitables_erm_output_iter_1		5 months ago
wikitables_tables_csv_tsv		5 months ago
wikitables_example_files		5 months ago
wikitables_preprocessed_20...		6 months ago

Running experiments - Keep track of what you ran

The screenshot shows the Beaker platform interface. At the top left is the Allen Institute for Artificial Intelligence logo. To its right is the Beaker logo with a stylized orange beaker icon. The top navigation bar includes links for Home, Experiments, Datasets, Groups, and a user profile. A search bar is located at the top center.

On the left side, there are filters for "Created by" (with "Myself" selected), "From" (set to "Any time"), and "Actions" (with buttons for "+ Create Group" and "Reset Search").

The main content area displays a table of experiments. The columns are "Title", "Description", and "Created". Each experiment row includes a checkbox, a green circular icon with a number (e.g., 1), the experiment name, and the creation date.

	Title	Description	Created
<input type="checkbox"/>	ex_1ddhr6gel5ll		a month ago
<input type="checkbox"/>	ex_enaqgsro7kpg		2 months ago
<input type="checkbox"/>	state_fixed6		3 months ago
<input type="checkbox"/>	state_fixed5		3 months ago
<input type="checkbox"/>	state_fixed4		3 months ago
<input type="checkbox"/>	state_fixed3		3 months ago
<input type="checkbox"/>	state_fixed2		3 months ago

Running experiments - Keep track of what you ran

The screenshot shows the Beaker interface for managing experiments. At the top, there's a navigation bar with the Allen Institute for Artificial Intelligence logo, the Beaker logo, and links for Home, Experiments, Datasets, Groups, and user Hello, mattg. A dropdown menu icon is also present.

The main content area displays an experiment titled "action_refactoring3". Below the title, it says "None" and "Created 3 months ago by mattg". There are two tabs: "Comparisons" and "Permissions", with "Comparisons" being active.

Below the tabs is a search bar with placeholder "Search by task or experiment name" and a "Search" button. To the right are buttons for "Manage Comparisons" and "Export to CSV".

A table follows, showing experimental results. The columns are: Task (dropdown), Experiment (dropdown), best_validation_denotation_acc, numpy_seed, pytorch_seed, and random_seed. The rows list five training runs with their respective parameters and validation scores:

Task	Experiment	best_validation_denotation_acc	numpy_seed	pytorch_seed	random_seed
training	state_fixed2	0.381	1,234.0	123.0	1,336.0
training	master4	0.377	9,834.0	953.0	4,536.0
training	master2	0.373	1,234.0	123.0	1,336.0
training	state_fixed6	0.373	283,701.0	36,301.0	953,631.0
training	state_fixed1	0.372			

Running experiments - Controlled experiments

- Which one gives more understanding?

Model	Ensemble		
	Size	Dev.	Test
Neelakantan et al. (2017)	1	34.1	34.2
Haug et al. (2017)	1	-	34.8
Pasupat and Liang (2015)	1	37.0	37.1
Neelakantan et al. (2017)	15	37.5	37.7
Haug et al. (2017)	15	-	38.7
Our Parser	1	42.7	43.3
Our Parser	5	-	45.9

Table 1: Development and test set accuracy of our semantic parser compared to prior work on WIKI TABLE QUESTIONS.

Model	Dev. Accuracy
Full model	42.7
token features, no similarity	28.1
all features, no similarity	37.8
similarity only, no features	27.5

Table 3: Development accuracy of ablated parser variants trained without parts of the entity linking module.

Running experiments - Controlled experiments

- Which one gives more understanding?

Model	Ensemble		
	Size	Dev.	Test
Neelakantan et al. (2017)	1	34.1	34.2
Haug et al. (2017)	1	-	34.8
Pasupat and Liang (2015)	1	37.0	37.1
Neelakantan et al. (2017)	15	37.5	37.7
Haug et al. (2017)	15	-	38.7
Our Parser	1	42.7	43.3
Our Parser	5	-	45.9

Table 1: Development and test set accuracy of our semantic parser compared to prior work on WIKITABLEQUESTIONS.

Important for putting your work in context

Table 3: Development accuracy of ablated parser variants trained without parts of the entity linking module.

Running experiments - Controlled experiments

- Which one gives more understanding?

Model	Ensemble		
	Size	Dev.	Test
Neelakantan et al. (2017)	1	34.1	34.2
Haug et al. (2017)	1	-	34.8
Pasupat and Liang (2015)	1	37.0	37.1
Neelakantan et al. (2017)	15	37.5	37.7
Haug et al. (2017)	15	-	38.7
Our Parser	1	42.7	43.3
Our Parser	5	-	45.9

Table 1: Development and test set accuracy of our semantic parser compared to prior work on WIKITABLEQUESTIONS.

But... too many moving parts, hard to know *what caused the difference*

Table 3: Development accuracy of ablated parser variants trained without parts of the entity linking module.

Running experiments - Controlled experiments

- Which one gives more understanding?

Model	Ensemble	Size	Dev.	Test
Very controlled experiments, varying one thing: we can make causal claims				
Our Parser		1	42.7	43.3
Our Parser		5	-	45.9

Table 1: Development and test set accuracy of our semantic parser compared to prior work on WIKI-TABLEQUESTIONS.

Model	Dev. Accuracy
Full model	42.7
token features, no similarity	28.1
all features, no similarity	37.8
similarity only, no features	27.5

Table 3: Development accuracy of ablated parser variants trained without parts of the entity linking module.

Running experiments - Controlled experiments

- Which one gives more understanding?

Model	Ensemble Size	Dev.	Test
How do you set up your code for this?			
Our Parser	1	42.7	43.3
Our Parser	5	-	45.9

Table 1: Development and test set accuracy of our semantic parser compared to prior work on WIKITABLEQUESTIONS.

Model	Dev. Accuracy
Full model	42.7
token features, no similarity	28.1
all features, no similarity	37.8
similarity only, no features	27.5

Table 3: Development accuracy of ablated parser variants trained without parts of the entity linking module.

Running experiments - Controlled experiments

```
class CrfTagger(Model):
    def __init__(self, vocab: Vocabulary,
                 text_field_embedder: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder,
                 label_namespace: str = "labels",
                 feedforward: Optional[FeedForward] = None,
                 label_encoding: Optional[str] = None,
                 constraint_type: Optional[str] = None,
                 include_start_end_transitions: bool = True,
                 constrain_crf_decoding: bool = None,
                 calculate_span_f1: bool = None,
                 dropout: Optional[float] = None,
                 verbose_metrics: bool = False,
                 initializer: InitializerApplicator = InitializerApplicator(),
                 regularizer: Optional[RegularizerApplicator] = None) -> None:
    ...

```

Running experiments - Controlled experiments

```
class CrfTagger(Model):
    def __init__(self, vocab: Vocabulary,
                 text_field_embedder: TextFieldEmbe
                 encoder: Seq2SeqEncoder,
                 label_namespace: str = "labels",
                 feedforward: Optional[FeedForward] = None,
                 label_encoding: Optional[str] = None,
                 constraint_type: Optional[str] = None,
                 include_start_end_transitions: bool = True,
                 constrain_crf_decoding: bool = None,
                 calculate_span_f1: bool = None,
                 dropout: Optional[float] = None,
                 verbose_metrics: bool = False,
                 initializer: InitializerApplicator = InitializerApplicator(),
                 regularizer: Optional[RegularizerApplicator] = None) -> None:
    ...

```

Possible ablations

Running experiments - Controlled experiments

```
class CrfTagger(Model):
    def __init__(self, vocab: Vocabulary,
                 text_field_embedder: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder,
                 label_namespace: str = "labels",
                 feedforward: Optional[FeedForward] = None,
                 label_encoding: Optional[str] = None,
                 constraint_type: Optional[str] = None,
                 include_start_end_transitions: bool = True,
                 constrain_crf_decoding: bool = None,
                 calculate_span_f1: bool = None,
                 dropout: Optional[float] = None,
                 verbose_metrics: bool = False,
                 initializer: InitializerApplicator = InitializerApplicator(),
                 regularizer: Optional[RegularizerApplicator] = None) -> None:
        ...
    
```

GloVe vs. character CNN vs.
ELMo vs. BERT

Running experiments - Controlled experiments

```
class CrfTagger(Model):
    def __init__(self, vocab: Vocabulary,
                 text_field_embedder: TextFieldEmbedder,
                 encoder: Seq2SeqEncoder,
                 label_namespace: str = "labels",
                 feedforward: Optional[FeedForward] = None,
                 label_encoding: Optional[str] = None,
                 constraint_type: Optional[str] = None,
                 include_start_end_transitions: bool = True,
                 constrain_crf_decoding: bool = None,
                 calculate_span_f1: bool = None,
                 dropout: Optional[float] = None,
                 verbose_metrics: bool = False,
                 initializer: InitializerApplicator = InitializerApplicator(),
                 regularizer: Optional[RegularizerApplicator] = None) -> None:
        ...
        self._tagger = CrfTagger(
            text_field_embedder=text_field_embedder,
            encoder=encoder,
            label_namespace=label_namespace,
            feedforward=feedforward,
            label_encoding=label_encoding,
            constraint_type=constraint_type,
            include_start_end_transitions=include_start_end_transitions,
            constrain_crf_decoding=constrain_crf_decoding,
            calculate_span_f1=calculate_span_f1,
            dropout=dropout,
            verbose_metrics=verbose_metrics,
            initializer=initializer,
            regularizer=regularizer)
```

LSTM vs. Transformer vs.
GatedCNN vs. QRNN

Running experiments - Controlled experiments

- Not good: modifying code to run different variants; hard to keep track of what you ran
- Better: configuration files, or separate scripts, or something

```
"model": {  
    "type": "crf_tagger",  
    "label_encoding": "BIOUL",  
    "constrain_crf_decoding": true,  
    "calculate_span_f1": true,  
    "dropout": 0.5,  
    "include_start_end_transitions": false,  
    "text_field_embedder": {  
        "token_embedders": {  
            "tokens": {  
                "type": "embedding",  
                "embedding_dim": 50,  
                "pretrained_file": "https://s3-us-west-1.amazonaws.com/allennlp-public-models/bert-large-cased-finetuned-cner-best-threshold/elmo_bert-large-cased-finetuned-cner-best-threshold.tar.gz",  
                "trainable": true  
            },  
            "elmo": {  
                "type": "elmo_token_embedder",  
                "options_file": "https://s3-us-west-2.amazonaws.com/allennlp-public-models/bert-large-cased-finetuned-cner-best-threshold/elmo_bert-large-cased-finetuned-cner-best-threshold.tar.gz",  
                "weight_file": "https://s3-us-west-2.amazonaws.com/allennlp-public-models/bert-large-cased-finetuned-cner-best-threshold/elmo_bert-large-cased-finetuned-cner-best-threshold.weight",  
                "do_layer_norm": false,  
                "dropout": 0.0  
            },  
            "roberta": {  
                "type": "roberta_token_embedder",  
                "options_file": "https://s3-us-west-2.amazonaws.com/allennlp-public-models/bert-large-cased-finetuned-cner-best-threshold/roberta_bert-large-cased-finetuned-cner-best-threshold.tar.gz",  
                "weight_file": "https://s3-us-west-2.amazonaws.com/allennlp-public-models/bert-large-cased-finetuned-cner-best-threshold/roberta_bert-large-cased-finetuned-cner-best-threshold.weight",  
                "do_layer_norm": false,  
                "dropout": 0.0  
            }  
        }  
    }  
}
```

Main goals during prototyping

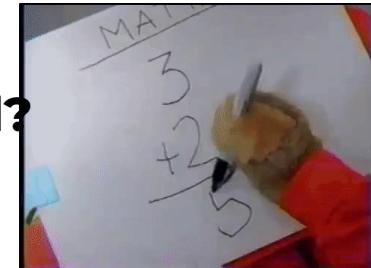
- Write code quickly



- Run experiments, keep track of what you tried



- **Analyze model behavior - did it do what you wanted?**



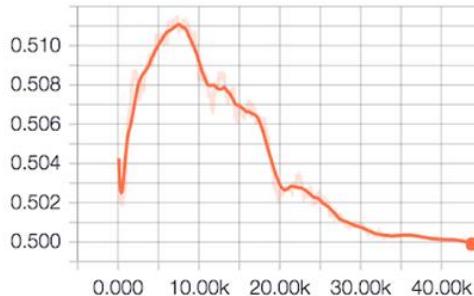
Analyze results - Tensorboard

- Crucial tool for understanding model behavior during training
- There is no better visualizer. If you don't use this, start now.

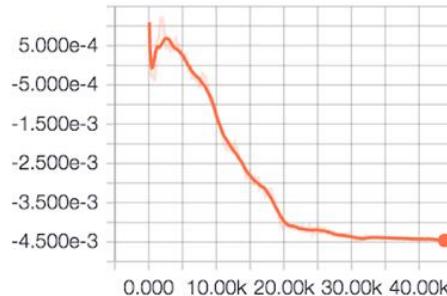
parameter_mean

46

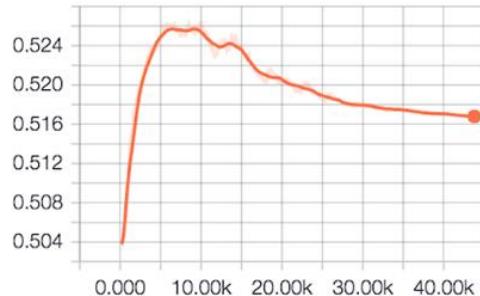
parameter_mean/_highway_layer_module_lay...



parameter_mean/_highway_layer_module_lay...



parameter_mean/_highway_layer_module_lay...



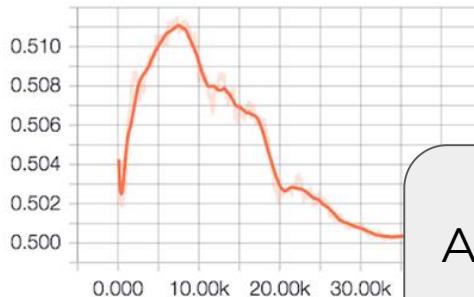
Analyze results - Tensorboard

- Crucial tool for understanding model behavior during training
- There is no better visualizer. If you don't use this, start now.

parameter_mean

46

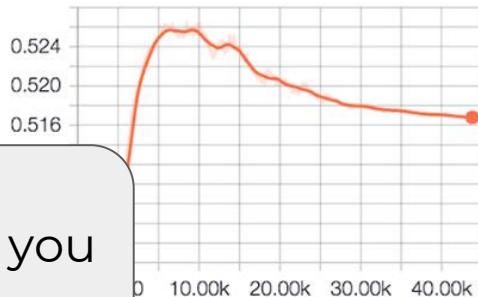
parameter_mean/_highway_layer_module_lay...



parameter_mean/_highway_layer_module_lay...



parameter_mean/_highway_layer_module_lay...



A good training loop will give you this for free, for any model.

Analyze results - Tensorboard

- Metrics
 - Loss
 - Accuracy etc.
- Gradients
 - Mean values
 - Std values
 - Actual update values
- Parameters
 - Mean values
 - Std values
- Activations
 - Log problematic activations

The screenshot shows the TensorBoard interface with the "SCALARS" tab selected. The top navigation bar includes links for "TensorBoard", "SCALARS", "DISTRIBUTIONS", and "HISTOGRAMS". On the left, there are several configuration options: "Show data download links" (unchecked), "Ignore outliers in chart scaling" (unchecked), and a dropdown for "Tooltip sorting method" set to "default". Below these are sliders for "Smoothing" (set to 0.6) and "Horizontal Axis" (set to "STEP"). A "Runs" section allows filtering runs using a regex input field, which contains "Write a regex to filter runs". A list of runs is shown with checkboxes:

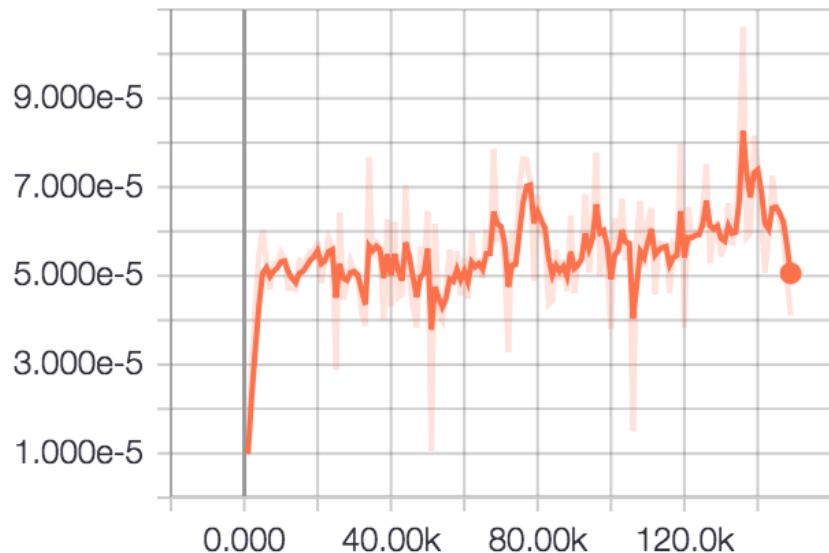
- ex_dpiy7d1yl497/tk_s3fqc3uctvtf/log/train (unchecked)
- ex_dpiy7d1yl497/tk_s3fqc3uctvtf/log/validation (checked)
- ex_k2r3fj70in0p/tk_jld9xwbz7adu/log/train (checked)
- ex_k2r3fj70in0p/tk_jld9xwbz7adu/log/validation (checked)

Analyze results - Tensorboard

Tensorboard will find optimisation bugs for you **for free**.

Here, the gradient for the embedding is 2 orders of magnitude different from the rest of the gradients.

gradient_update/text_field_embedder.token_embedder_tokens.weight



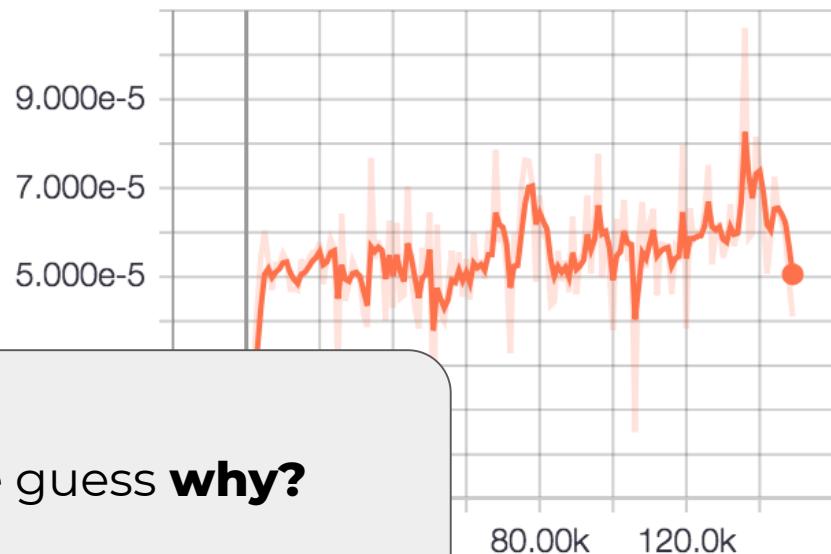
Analyze results - Tensorboard

Tensorboard will find optimisation bugs for you **for free.**

Here, the gradient for the embedding is 2 orders of magnitude different from the rest of the gradients.

Can anyone guess **why?**

gradient_update/text_field_embedder.token_embedder_tokens.weight



Analyze results - Tensorboard

Embeddings have **sparse gradients** (only some embeddings are updated), but the momentum coefficients from ADAM are calculated for the whole embedding every time.

The embedding is 2 orders of magnitude different from the rest of the gradients.

gradient_update/text_field_embedder.token_embeddings.weight



Solution:

```
from  
allennlp.training.optimizers  
import DenseSparseAdam
```

(uses sparse accumulators for gradient moments)

Analyze results - Look at your data!

- Good:

```
input: {"passage": "The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano.", "question": "Who stars in The Matrix?"}
prediction: {"best_span": [17, 33], "best_span_str": "Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano" }
```

mattg@dhcp-057215:allenlp\$ █

Analyze results - Look at your data!

- Better:

Passage

The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano.

Question

Who stars in The Matrix?

The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring **Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano**.

Analyze results - Look at your data!

- Better:

Passage

The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano.

The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano.

Question

How many people star in The Matrix?

Analyze results - Look at your data!

- Best:

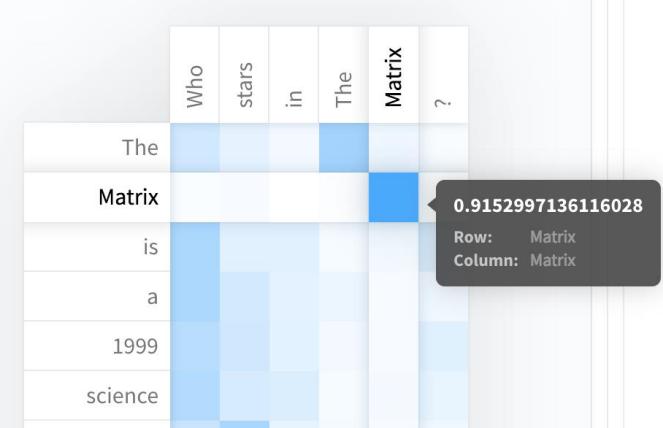
Passage

The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano.

Question

Who stars in The Matrix?

The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring **Keanu Reeves,** **Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and** **Joe Pantoliano .**



Analyze results - Look at your data!

- Best:

Passage

How do you design
your code for this?

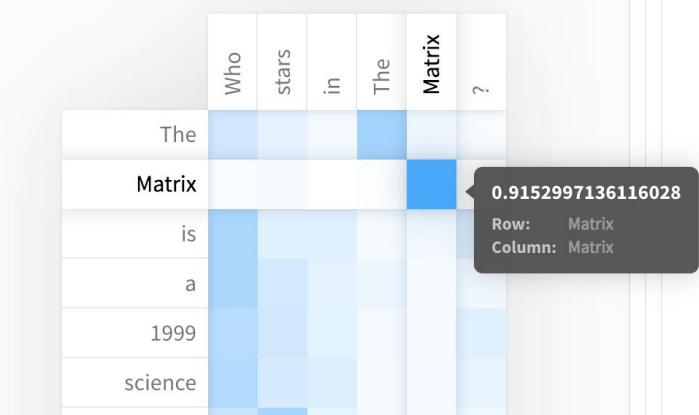
The Matrix
The Wachos-
Anne Moss

I directed by
rane, Carrie-

Question

Who stars in The Matrix?

The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring **Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano**.



Analyze results - Look at your data!

- Best:

Passage

How do you design
your code for this?

The Matrix
The Wachos-
Anne Moss

I directed by
Keanu, Carrie-

The Matrix is a 1999 science fiction action film written and directed by The Wachowskis, starring **Keanu Reeves, Laurence Fishburne, Carrie-Anne Moss, Hugo Weaving, and Joe Pantoliano**.

Question

Who stars

Well say more later, but the key points are:

- Separate data processing that also works on JSON
- Model needs to run without labels / computing loss

52997136116028
Matrix
mn: Matrix

**Key point during
prototyping:
The components that
you use matter. A lot.**

We'll give specific
thoughts on designing
components after the
break

Developing Good Processes

Source Control

We Hope You're Already Using Source Control!

makes it easy to safely experiment with
code changes

- if things go wrong, just revert!



We Hope You're Already Using Source Control!

- makes it easy to collaborate



We Hope You're Already Using Source Control!

- makes it easy to revisit older versions of your code



We Hope You're Already Using Source Control!

- makes it easy to implement code reviews



**That's right, code
reviews!**

About Code Reviews

- code reviewers find mistakes

Can you find the
the **mistake**?

1 2 3 4 5 6 7 8 9

About Code Reviews

- code reviewers point out improvements

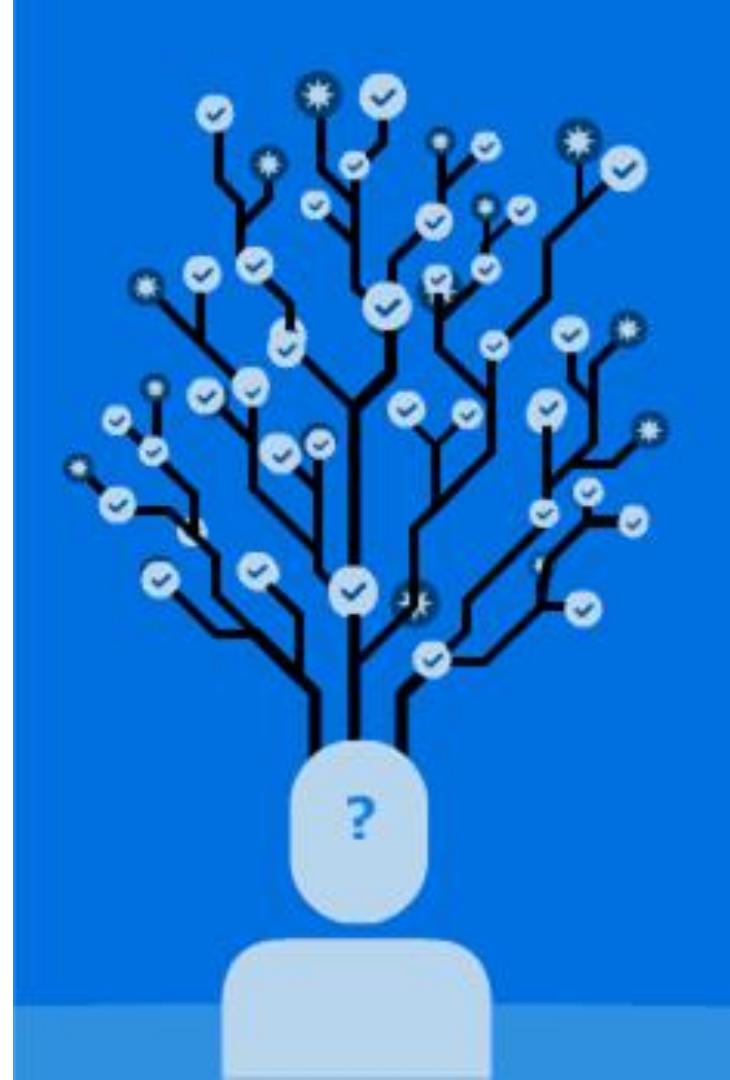


About Code Reviews

- code reviewers force you to make your code readable

About Code Reviews

and clear, readable code
allows your code reviews to
be discussions of your
modeling decisions



About Code Reviews

- code reviewers can be your scapegoat when it turns out your results are wrong because of a bug



Continuous Integration

(+ Build Automation)

Continuous Integration (+ Build Automation)

Continuous Integration

always be merging (into a branch)

Build Automation

always be running your tests (+ other checks)

(this means you have to write tests)

Example: Typical AllenNLP PR

✓ #4652.0fcf9a341d5e3d4954eb33310263cb2f53c566e8 (22 Oct 18 17:35) | ▾

Overview Changes 3 Build Log Parameters Artifacts Docker Info

« ⚠ #4633.0c99c12...edae1a2ddc53 | All history | Last recorded build

Tree view | Tail

Download full build log (~719.69 KB) | .zip

Repeat block names

[17:35:34] The build is removed from the queue to be prepared for the start
[17:35:34] ▶ Collecting changes in 1 VCS root
[17:35:34] Starting the build on the agent agent-1
[17:35:35] Clearing temporary directory: /local/deploy/agent1/temp/buildTmp
[17:35:35] ▶ Publishing internal artifacts
[17:35:35] Clean build enabled: removing old files from /local/deploy/agent1/work/98197cf33cb401e5
[17:35:35] Checkout directory: /local/deploy/agent1/work/98197cf33cb401e5
[17:35:35] ▶ Updating sources: auto checkout (on agent) (1s)
[17:35:36] ▶ Step 1/9: Docker Build (Docker) (8s)
[17:35:45] ▶ Step 2/9: Docker Push SHA (Command Line)
[17:35:45] ▶ Step 3/9: Unit Tests (pytest) (Command Line) (6m:28s)
[17:42:13] ▶ Step 4/9: Linter (pylint) (Command Line) (1m:51s)
[17:44:05] ▶ Step 5/9: Type Checker (mypy) (Command Line) (33s)
[17:44:38] ▶ Step 6/9: Build Docs (Command Line) (1m:26s)
[17:46:05] ▶ Step 7/9: Check Docs (Command Line) (4s)
[17:46:10] ▶ Step 8/9: Check Links (Command Line)
[17:46:10] ▶ Step 9/9: Sniff Tests (Command Line) (4m:25s)
[17:50:35] ▶ Publishing artifacts
[17:50:35] ▶ Publishing internal artifacts
[17:50:36] Build finished

```
[17:35:34] The build is removed from the queue to be prepared for the start
[17:35:34] ▶ Collecting changes in 1 VCS root
[17:35:34] Starting the build on the agent agent-1
[17:35:35] Clearing temporary directory: /local/deploy/agent1/temp/buildTmp
[17:35:35] ▶ Publishing internal artifacts
[17:35:35] Clean build enabled: removing old files from /local/deploy/agent1
[17:35:35] Checkout directory: /local/deploy/agent1/work/98197cf33cb401e5
[17:35:35] ▶ Updating sources: auto checkout (on agent) (1s)
[17:35:36] ▶ Step 1/9: Docker Build (Docker) (8s)
[17:35:45] ▶ Step 2/9: Docker Push SHA (Command Line)
[17:35:45] ▶ Step 3/9: Unit Tests (pytest) (Command Line) (6m:28s)
[17:42:13] ▶ Step 4/9: Linter (pylint) (Command Line) (1m:51s)
[17:44:05] ▶ Step 5/9: Type Checker (mypy) (Command Line) (33s)
[17:44:38] ▶ Step 6/9: Build Docs (Command Line) (1m:26s)
[17:46:05] ▶ Step 7/9: Check Docs (Command Line) (4s)
[17:46:10] ▶ Step 8/9: Check Links (Command Line)
[17:46:10] ▶ Step 9/9: Sniff Tests (Command Line) (4m:25s)
[17:50:35] ▶ Publishing artifacts
[17:50:35] ▶ Publishing internal artifacts
[17:50:36] Build finished
```

if you're not building a
library that lots of
other people rely on,
you probably don't
need all these steps

**but you do need some
of them**

Testing Your Code

What do we mean by "test your code"?

Write Unit Tests

a **unit test** is
an *automated*
check that a
small part of
your code
works correctly

```
1 import test from 'tape';
2 import compose from '../source/compose';
3
4 test('Compose function output type', () => {
5   const actual = typeof compose();
6   const expected = 'function';
7
8   assert.equal(actual, expected, 'compose() should return a function');
9
10  assert.end();
11 });
12});
```

What should I test?

If You're Prototyping,
Test the Basics

Prototyping? Test the Basics

```
def test_read_from_file(self):
    conll_reader = Conll2003DatasetReader()
    instances = conll_reader.read('data/conll2003.txt'))
    instances = ensure_list(instances)

    expected_labels = ['I-ORG', 'O', 'I-PER', 'O', 'O', 'I-LOC', 'O']

    fields = instances[0].fields
    tokens = [t.text for t in fields['tokens'].tokens]
    assert tokens == ['U.N.', 'official', 'Ekeus', 'heads', 'for', 'Baghdad',
        '.']
    assert fields["tags"].labels == expected_labels

    fields = instances[1].fields
    tokens = [t.text for t in fields['tokens'].tokens]
    assert tokens == ['AI2', 'engineer', 'Joel', 'lives', 'in', 'Seattle', '.']
    assert fields["tags"].labels == expected_labels
```

Prototyping? Test the Basics

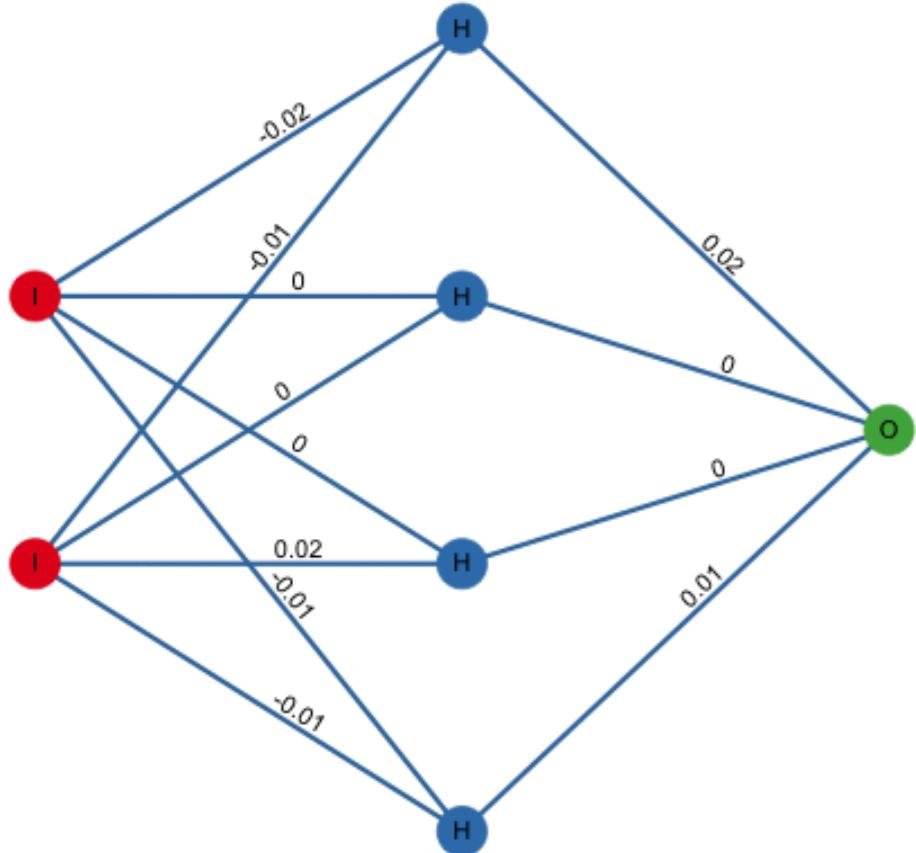
```
def test_forward_pass_runs_correctly(self):
    output_dict = self.model(**self.training_tensors)
    tags = output_dict['tags']
    assert len(tags) == 2
    assert len(tags[0]) == 7
    assert len(tags[1]) == 7
    for example_tags in tags:
        for tag_id in example_tags:
            tag = idx_to_token[tag_id]
            assert tag in {'O', 'I-ORG', 'I-PER', 'I-LOC'}
```

If You're Writing
Reusable Components,
Test Everything

Weights after iteration 0

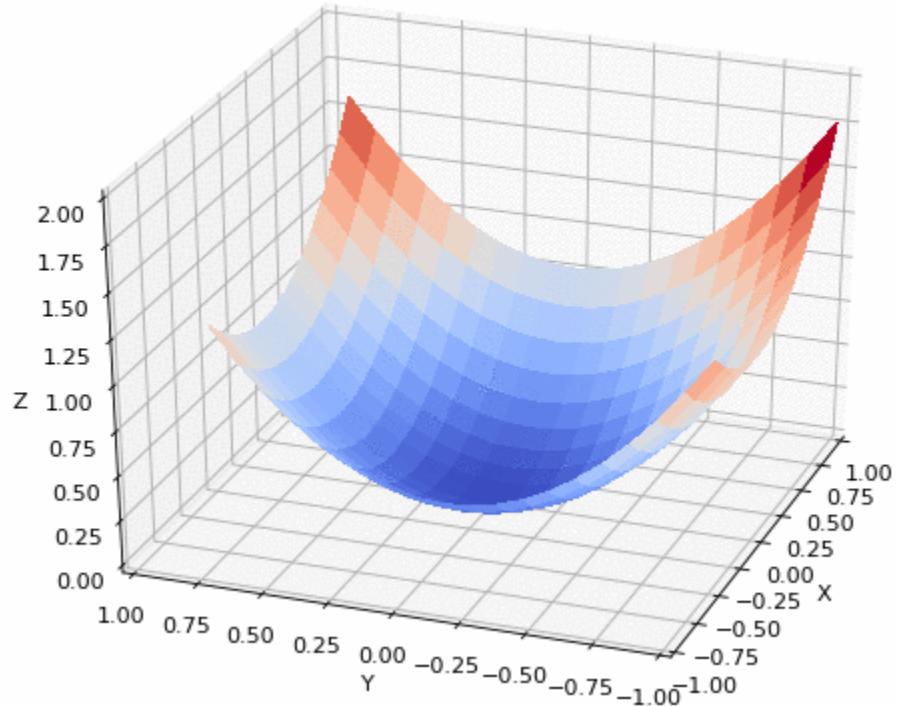
Test Everything

test your model can train,
save, and load



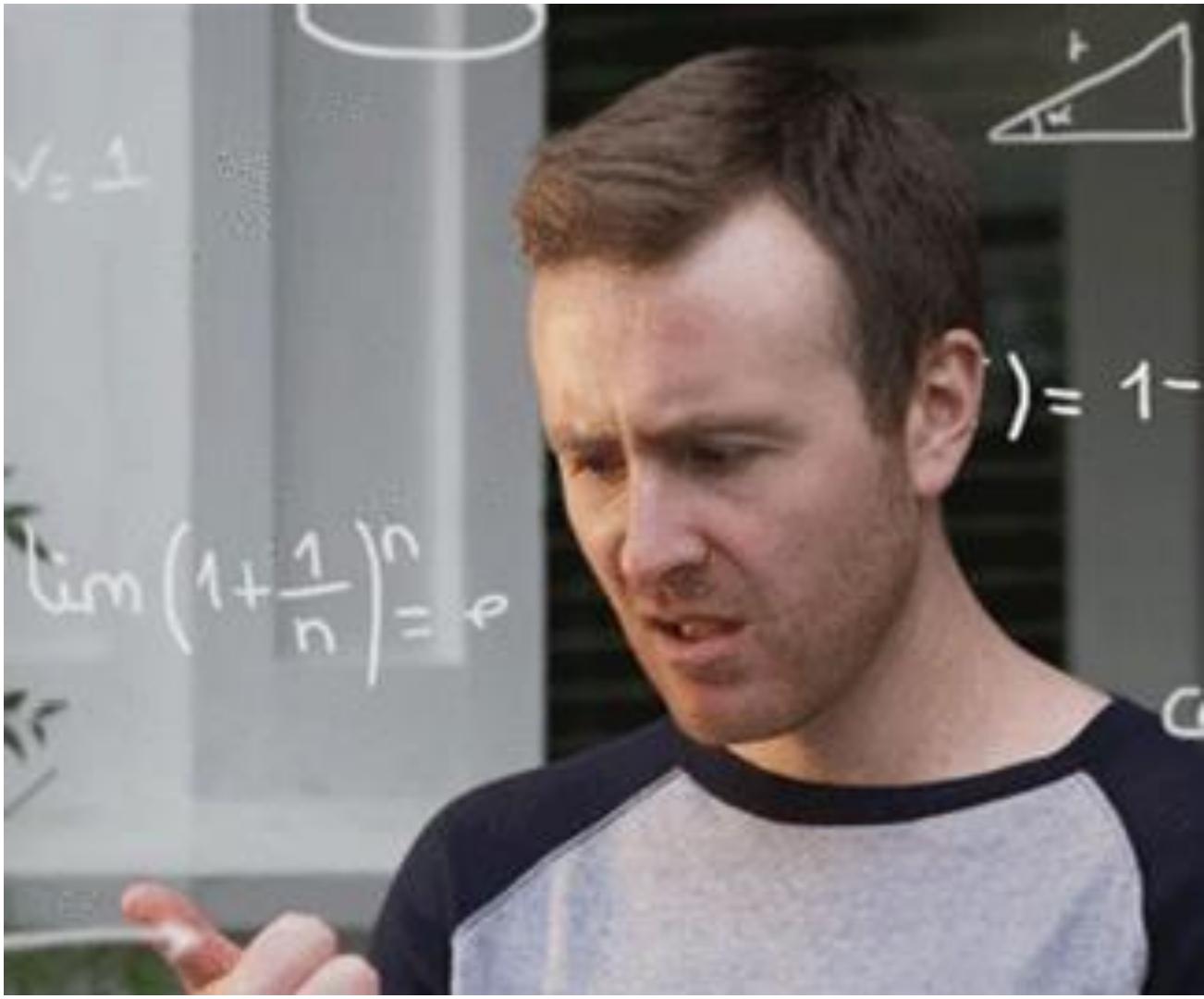
Test Everything

test that it's computing /
backpropagating gradients



Test Everything

but how?



Use Test Fixtures

create tiny datasets that look like the real thing

The###DET dog###NN ate###V
the###DET apple###NN

Everybody###NN read###V
that###DET book###NN



Use Test Fixtures

use them to create tiny
pretrained models

It's ok if the weights are
essentially random. We're
not testing that the model is
any **good**.



Use Test Fixtures

- write unit tests that use them to run your data pipelines and models
 - detect logic errors
 - detect malformed outputs
 - detect incorrect outputs



Use your knowledge to write clever tests

```
def test_attention_is_normalised_correctly(self):
    input_dim = 7
    sequence_tensor = torch.randn([2, 5, input_dim])
    extractor = SelfAttentiveSpanExtractor(input_dim=input_dim)
    # In order to test the attention, we'll make the weight which
    # computes the logits zero, so the attention distribution is
    # uniform over the sentence. This lets us check that the
    # computed spans are just the averages of their representations.
    extractor._global_attention._module.weight.data.fill_(0.0)
    extractor._global_attention._module.bias.data.fill_(0.0)
    span_representations = extractor(sequence_tensor,
    spans = span_representations[0]
    mean_embeddings = sequence_tensor[0, 1:4, :].mean(0)
    numpy.testing.assert_array_almost_equal(spans[0].data.numpy(),
                                            mean_embeddings.data.numpy()))
```

Attention is hard to
test because it relies
on parameters

Use your knowledge to write clever tests

```
def test_attention_is_normalised_correctly(self):
    input_dim = 7
    sequence_tensor = torch.randn([2, 5, input_dim])
    extractor = SelfAttentiveSpanExtractor(input_dim=input_dim)
    # In order to test the attention, we'll make the weight which
    # computes the logits zero, so the attention distribution is
    # uniform over the sentence. This lets us check that the
    # computed spans are just the averages of their representations.
    extractor._global_attention._module.weight.data.fill_(0.0)
    extractor._global_attention._module.bias.data.fill_(0.0)
    span_representations = extractor(sequence_tensor)

    spans = span_representations[0]
    mean_embeddings = sequence_tensor[0, 1:4, :].mean(dim=0)
    numpy.testing.assert_array_almost_equal(spans[0], mean_embeddings)
```

Idea: Make the parameters deterministic so you can test **everything** else

Pre-Break Summary

- Two Modes of Writing Research Code
 - Difference between prototyping and building components
 - When should you transition?
 - Good ways to analyse results
- Developing Good Processes
 - How to write good tests
 - How to know what to test
 - Why you should do code reviews

BREAK

please fill out our survey:
<https://tinyurl.com/emnlp-tutorial-survey>

will tweet out link to slides after talk
@ai2_allennlp

Reusable Components

What are the right abstractions for NLP?

The Right Abstractions

- AllenNLP now has more than 20 models in it
 - some simple
 - some complex
- Some abstractions have consistently proven useful
- (Some haven't)



Things That We Use A Lot

- training a model
- mapping words (or characters, or labels) to indexes
- summarizing a sequence of tensors with a single tensor



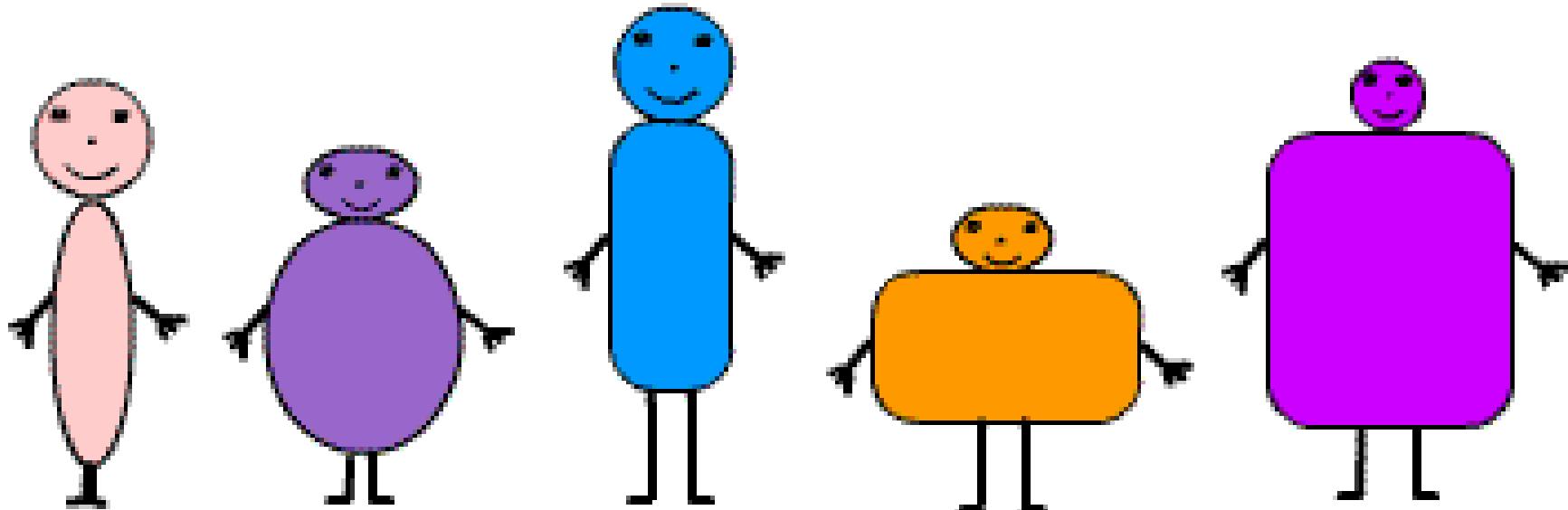
Things That Require a Fair Amount of Code

- training a model
- (some ways of) summarizing a sequence of tensors with a single tensor
- some neural network modules



Things That Have Many Variations

- turning a word (or a character, or a label) into a tensor
- summarizing a sequence of tensors with a single tensor
- transforming a sequence of tensors into a sequence of tensors



Things that reflect our higher-level thinking

- we'll have some inputs:
 - text, almost certainly
 - tags/labels, often
 - spans, sometimes
- we need some ways of *embedding* them as tensors
 - one hot encoding
 - low-dimensional embeddings
- we need some ways of dealing with sequences of tensors
 - sequence in -> sequence out (e.g. all outputs of an LSTM)
 - sequence in -> tensor out (e.g. last output of an LSTM)



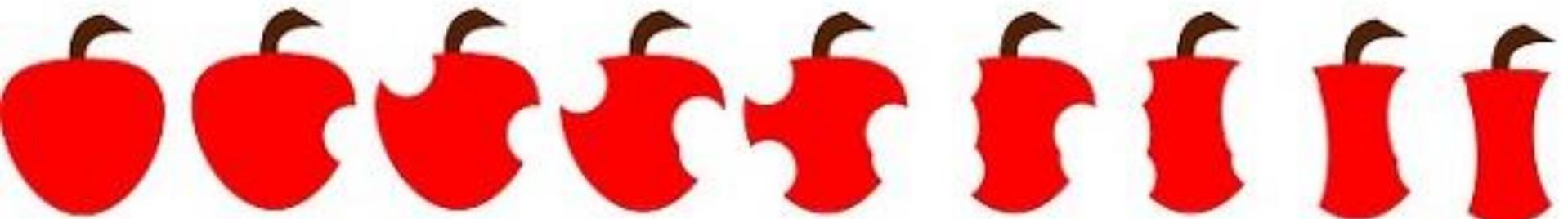
**Along the way, we need to worry
about some things that make
NLP tricky**

Inputs are *text*, but neural models want *tensors*

Formatted as Text		Formatted as Numbers	
	00007969910000		7969910000
	000098255950		98255950
	0000526982		526982
	00002416450		2416450
	0000475301		475301
	0000184595		184595
	0000434741		434741
	0000128432		128432
	0000282064		282064
	00002318918		2318918
	0000460158		460158
SUM	0	SUM	8075393591

Inputs are *sequences* of things

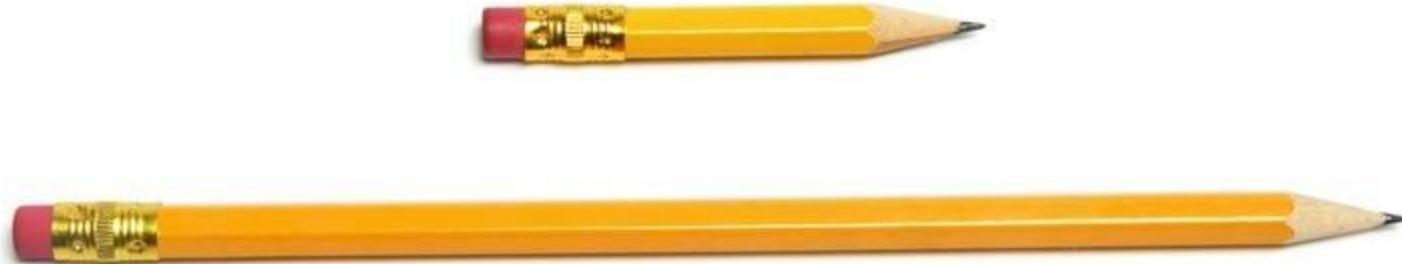
and order matters



Inputs can vary in length

Some sentences are short.

Whereas other sentences are so long that by the time you finish reading them you've already forgotten what they started off talking about and you have to go back and read them a second time in order to remember the parts at the beginning.



Reusable Components in AllenNLP

**AllenNLP is built on
PyTorch**

AllenNLP is built on PyTorch

and is inspired by the question
"what higher-level components
would help NLP researchers do
their research better + more
easily?"



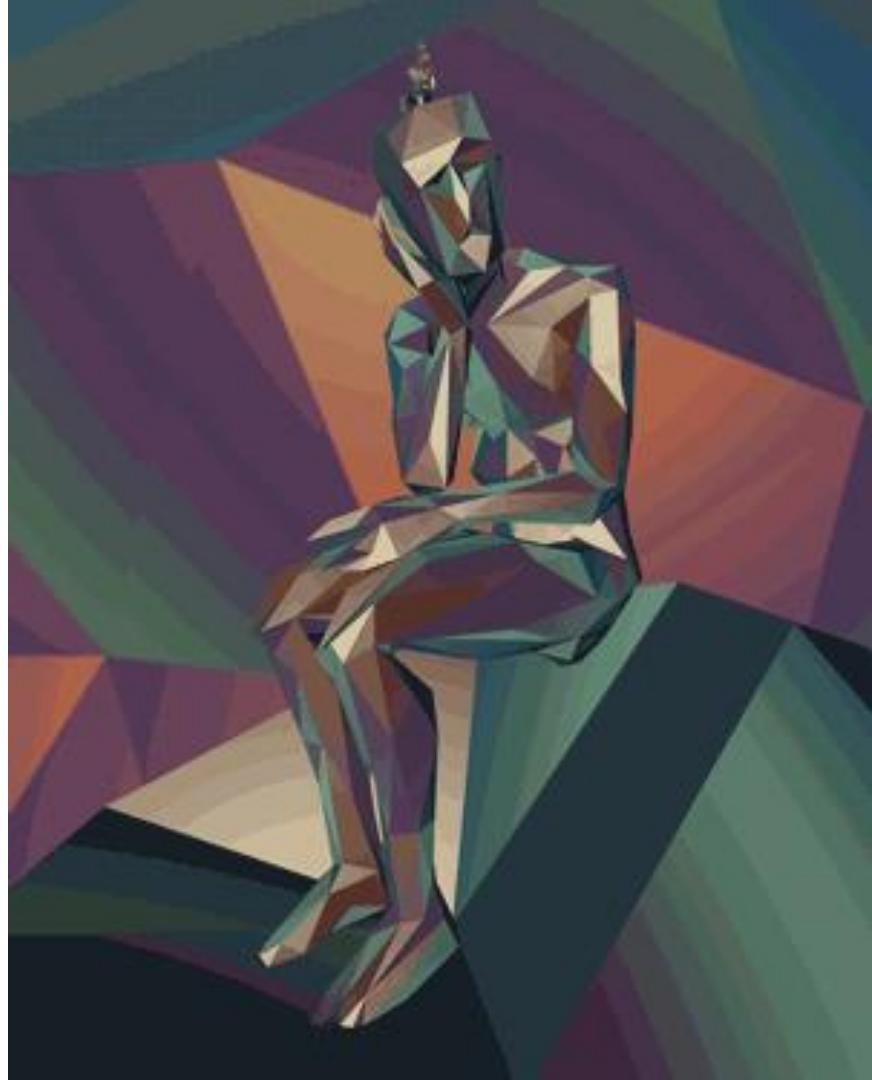
AllenNLP is built on PyTorch

under the covers, every piece
of a model is a `torch.nn.Module`
and every number is part of a
`torch.Tensor`



AllenNLP is built on PyTorch

but we want you to be able to reason at a higher level most of the time



**hence the higher level
concepts**

the Model

```
class Model(torch.nn.Module, Registrable):
    def __init__(self,
                 vocab: Vocabulary,
                 regularizer: RegularizerApplicator = None) -> None: ...

    def forward(self, *inputs) -> Dict[str, torch.Tensor]: ...

    def get_metrics(self, reset: bool = False) -> Dict[str, float]: ...

    @classmethod
    def load(cls,
            config: Params,
            serialization_dir: str,
            weights_file: str = None,
            cuda_device: int = -1) -> 'Model': ...
```

Model.forward

```
def forward(self, *inputs) -> Dict[str, torch.Tensor]: ...
```

- returns a dict [!]
- by convention, "loss" tensor is what the training loop will optimize
- but as a dict entry, "loss" is completely optional
 - which is good, since at inference / prediction time you don't have one
- can also return predictions, model internals, or any other outputs you'd want in an output dataset or a demo

every NLP project needs a Vocabulary

```
class Vocabulary(Registrable):

    def __init__(self,
                 counter: Dict[str, Dict[str, int]] = None,
                 min_count: Dict[str, int] = None,
                 max_vocab_size: Union[int, Dict[str, int]] = None,
                 non_padded_namespaces: Iterable[str] = DEFAULT_NON_PADDED_NAMESPACES,
                 pretrained_files: Optional[Dict[str, str]] = None,
                 only_include_pretrained_words: bool = False,
                 tokens_to_add: Dict[str, List[str]] = None,
                 min_pretrained_embeddings: Dict[str, int] = None) -> None: ...

    @classmethod
    def from_instances(cls, instances: Iterable['Instance'], ...) -> 'Vocabulary': ...

    def add_token_to_namespace(self, token: str, namespace: str = 'tokens') -> int: ...

    def get_token_index(self, token: str, namespace: str = 'tokens') -> int: ...

    def get_token_from_index(self, index: int, namespace: str = 'tokens') -> str: ...
        return self._index_to_token[namespace][index]

    def get_vocab_size(self, namespace: str = 'tokens') -> int: ...
        return len(self._token_to_index[namespace])
```

a Vocabulary is built from Instances

```
class Instance(Mapping[str, Field]):  
    def __init__(self, fields: MutableMapping[str, Field]) -> None: ...  
  
    def add_field(self, field_name: str, field: Field, vocab: Vocabulary = None) -> None: ...  
  
    def count_vocab_items(self, counter: Dict[str, Dict[str, int]]): ...  
  
    def index_fields(self, vocab: Vocabulary) -> None: ...  
  
    def get_padding_lengths(self) -> Dict[str, Dict[str, int]]: ...  
  
    def as_tensor_dict(self,  
                      padding_lengths: Dict[str, Dict[str, int]] = None) -> Dict[str, DataArray]:
```

an Instance is a collection of Fields

a Field contains a data element and knows how to turn it into a tensor

```
class Field(Generic[DataArray]):  
    def count_vocab_items(self, counter: Dict[str, Dict[str, int]]): ...  
  
    def index(self, vocab: Vocabulary): ...  
  
    def get_padding_lengths(self) -> Dict[str, int]: ...  
  
    def as_tensor(self, padding_lengths: Dict[str, int]) -> DataArray: ...  
  
    def empty_field(self) -> 'Field': ...  
  
    def batch_tensors(self, tensor_list: List[DataArray]) -> DataArray: ...
```

Many kinds of Fields

- `TextField`: represents a sentence, or a paragraph, or a question, or ...
- `LabelField`: represents a single label (e.g. "entailment" or "sentiment")
- `SequenceLabelField`: represents the labels for a sequence (e.g. part-of-speech tags)
- `SpanField`: represents a span (start, end)
- `IndexField`: represents a single integer index
- `ListField[T]`: for repeated fields
- `MetadataField`: represents anything (but not tensorizable)

Example: an Instance for SNLI

```
def text_to_instance(self,
                    premise: str,
                    hypothesis: str,
                    label: str = None) -> Instance:

    fields: Dict[str, Field] = {}

    premise_tokens = self._tokenizer.tokenize(premise)
    hypothesis_tokens = self._tokenizer.tokenize(hypothesis)

    fields['premise'] = TextField(premise_tokens, self._token_indexers)
    fields['hypothesis'] = TextField(hypothesis_tokens, self._token_indexers)

    if label:
        fields['label'] = LabelField(label)

    metadata = {"premise_tokens": [x.text for x in premise_tokens],
                "hypothesis_tokens": [x.text for x in hypothesis_tokens]}
    fields["metadata"] = MetadataField(metadata)

    return Instance(fields)
```

Example: an Instance for SQuAD

```
def make_reading_comprehension_instance(question_tokens: List[Token],  
                                         passage_tokens: List[Token],  
                                         token_indexers: Dict[str, TokenIndexer],  
                                         token_spans: List[Tuple[int, int]] = None) -> Instance:  
  
    fields: Dict[str, Field] = {}  
  
    fields['passage'] = TextField(passage_tokens, token_indexers)  
    fields['question'] = TextField(question_tokens, token_indexers)  
  
    if token_spans:  
        # There may be multiple answer annotations, so we pick the one that occurs the most.  
        candidate_answers: Counter = Counter()  
        for span_start, span_end in token_spans:  
            candidate_answers[(span_start, span_end)] += 1  
        span_start, span_end = candidate_answers.most_common(1)[0][0]  
  
        fields['span_start'] = IndexField(span_start, passage_field)  
        fields['span_end'] = IndexField(span_end, passage_field)  
  
    return Instance(fields)
```

What's a TokenIndexer?

- how to represent text in our model is one of the fundamental decisions in doing NLP
- many ways, but pretty much always want to turn text into indices
- many choices
 - sequence of unique `token_ids` (or `id` for OOV) from a vocabulary
 - sequence of sequence of `character_ids`
 - sequence of ids representing byte-pairs / word pieces
 - sequence of `pos_tag_ids`
- might want to use several
- this is (deliberately) independent of the choice about how to embed these as tensors

And don't forget DatasetReader

- "given a path [usually but not necessarily to a file], produce Instances"
- decouples your modeling code from your data-on-disk format
- two pieces:
 - `text_to_instance`: creates an instance from named inputs ("passage", "question", "label", etc..)
 - `read`: parses data from a file and (typically) hands it to `text_to_instance`
- new dataset -> create a new DatasetReader (not too much code), but keep the model as-is
- same dataset, new model -> just re-use the DatasetReader
- default is to read all instances into memory, but base class handles laziness if you want it

Library also handles batching, via DataIterator

- BasicIterator just shuffles (optionally) and produces fixed-size batches
- BucketIterator groups together instances with similar "length" to minimize padding
- (Correctly padding and sorting instances that contain a variety of fields is slightly tricky; a lot of the API here is designed around getting this right)
- Maybe someday we'll have a working AdaptiveIterator that creates variable GPU-sized batches



Tokenizer

- Single abstraction for both word-level and character-level tokenization
- Possibly this wasn't the right decision!
- Pros:
 - easy to switch between words-as-tokens and characters-as-tokens in the same model
- Cons:
 - non-standard names + extra complexity
 - doesn't seem to get used this way at all



back to the Model

Model is a subclass of torch.nn.Module

- so if you give it members that are `torch.nn.Parameters` or are themselves `torch.nn.Modules`, all the optimization will just work*
- for reasons we'll see in a bit, we'll also *inject* any model component that we might want to configure
- and AllenNLP provides NLP / deep-learning abstractions that allow us not to reinvent the wheel

*usually on the first try it won't "just work", but usually that's your fault not PyTorch's

TokenEmbedder

- turns ids (the outputs of your TokenIndexers) into tensors
- many options:
 - learned word embeddings
 - pretrained word embeddings
 - contextual embeddings (e.g. ELMo)
 - character embeddings + Seq2VecEncoder

Seq2VecEncoder

(batch_size, sequence_length, embedding_dim)



(batch_size, embedding_dim)

- bag of words
- (last output of) LSTM
- CNN + pooling

Seq2SeqEncoder

(batch_size, sequence_length, embedding_dim)

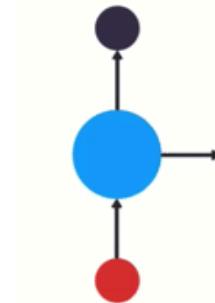


(batch_size, sequence_length, embedding_dim)

- LSTM (and friends)
- self-attention
- do-nothing

Wait, Two Different Abstractions for RNNs?

- Conceptually, RNN-for-Seq2Seq is different from RNN-for-Seq2Vec
- In particular, the class of possible replacements for the former is different from the class of replacements for the latter
- That is, "RNN" is not the right abstraction for NLP!



Attention

(batch_size, sequence_length, embedding_dim),

(batch_size, embedding_dim)



(batch_size, sequence_length)

- dot product ($x^T y$)
- bilinear ($x^T W y$)
- linear ($[x; y; x^T y; \dots]^T w$)

MatrixAttention

(batch_size, sequence_length1, embedding_dim),

(batch_size, sequence_length2, embedding_dim)



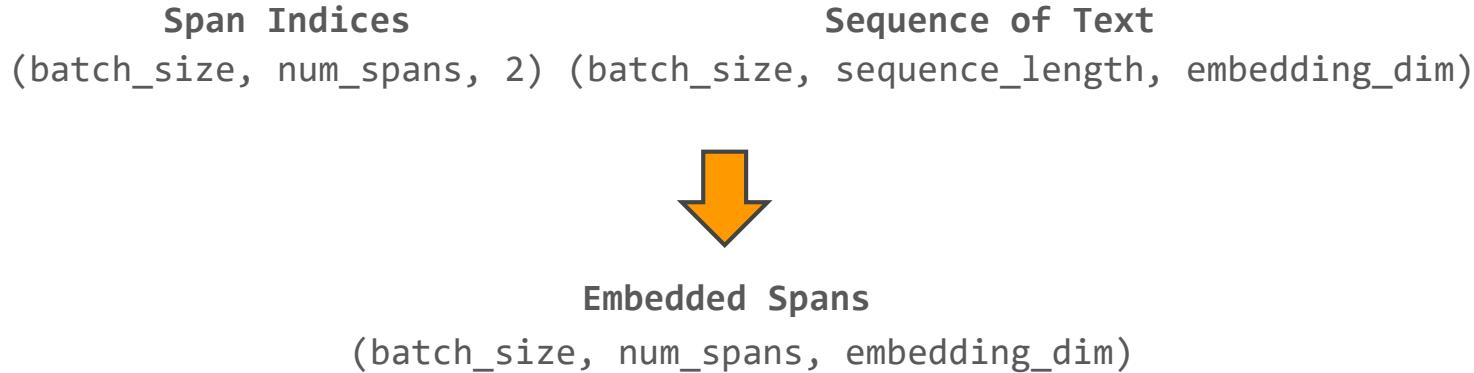
(batch_size, sequence_length1, sequence_length2)

- dot product ($x^T y$)
- bilinear ($x^T W y$)
- linear ($[x; y; x^T y; \dots]^T w$)

Attention and MatrixAttention

- These look similar - you could imagine sharing the similarity computation code
- We did this at first - code sharing, yay!
- But it was very memory inefficient - code sharing isn't always a good idea
- You could also imagine having a single Attention abstraction that also works for attention matrices
- But then you have a muddled and confusing input/output spec
- So, again, more duplicated (or at least very similar) code, but in this case that's probably the right decision, especially for efficiency

SpanExtractor



- Many modern NLP models use representations of spans of text
 - Used by the Constituency Parser and the Co-reference model in AllenNLP
 - We generalised this after needing it again to implement the Constituency Parser.
- Lots of ways to represent a span:
 - Difference of endpoints
 - Concatenation of endpoints (etc)
 - Attention over intermediate words

This seems like a lot of abstractions!

- But in most cases it's pretty simple:
 - create a DatasetReader that generates the Instances you want
 - (if you're using a standard dataset, likely one already exists)
 - create a Model that turns Instances into predictions and a loss
 - use off-the-shelf components => can often write little code
 - create a JSON config and use the AllenNLP training code
 - (and also often a Predictor, coming up next)
- We'll go through a detailed example at the end of the tutorial
- And you can write as much PyTorch as you want when the built-in components don't do what you need

Abstractions just to
make your life nicer

Declarative syntax

```
"model": {  
    "type": "crf_tagger",  
    "label_encoding": "BIOUL",  
    "constrain_crf_decoding": true,  
    "calculate_span_f1": true,  
    "dropout": 0.5,  
    "include_start_end_transitions": false,  
    "text_field_embedder": {  
        "token_embedders": {  
            "tokens": {  
                "type": "embedding",  
                "embedding_dim": 50,  
                "pretrained_file": "gLove.6B.50d.txt.gz",  
                "trainable": true  
            },  
        },  
    },  
},
```

most AllenNLP objects can be instantiated from Jsonnet blobs

```
"token_characters": {  
    "type": "character_encoding",  
    "embedding": {  
        "embedding_dim": 16  
    },  
    "encoder": {  
        "type": "cnn",  
        "embedding_dim": 16,  
        "num_filters": 128,  
        "ngram_filter_sizes": [3],  
        "conv_layer_activation": "relu"  
    }  
},  
},  
},  
"encoder": {  
    "type": "LSTM",  
    "input_size": 50 + 128,  
    "hidden_size": 200,  
    "num_layers": 2,  
    "dropout": 0.5,  
    "bidirectional": true  
},  
},
```

Declarative syntax

- allows us to specify an entire experiment using JSON
- allows us to change architectures without changing code

```
"encoder": {  
    "type": "Lstm",  
    "input_size": 50 + 128,  
    "hidden_size": 200,  
    "num_Layers": 2,  
    "dropout": 0.5,  
    "bidirectional": true  
},
```



```
"encoder": {  
    "type": "gru",  
    "input_size": 50 + 128,  
    "hidden_size": 200,  
    "num_Layers": 1,  
    "dropout": 0.5,  
    "bidirectional": true  
},
```



```
"encoder": {  
    "type": "pass_through",  
    "input_dim": 50 + 128  
},
```

Declarative syntax

How does it work?

- Registrable
 - retrieve a class by its name
- FromParams
 - instantiate a class instance from JSON



Registrable

```
class Model(torch.nn.Module, Registrable):  
...  
  
@Model.register("bidaf")  
class BidirectionalAttentionFlow(Model): ...  
  
@Model.register("decomposable_attention")  
class DecomposableAttention(Model): ...  
  
@Model.register("simple_tagger")  
class SimpleTagger(Model):
```

returns the class itself



```
model = Model.by_name("bidaf")(param1,  
                             param2,  
                             ...)
```

- so now, given a model "type" (specified in the JSON config), we can programmatically retrieve the class
- remaining problem: how do we programmatically call the constructor?

Model config, again

```
"model": {  
    "type": "crf_tagger",  
    "label_encoding": "BIOUL",  
    "constrain_crf_decoding": true,  
    "calculate_span_f1": true,  
    "dropout": 0.5,  
    "include_start_end_transitions": false,  
    "text_field_embedder": {  
        "token_embedders": {  
            "tokens": {  
                "type": "embedding",  
                "embedding_dim": 50,  
                "pretrained_file": "gLove.6B.50d.txt.gz",  
                "trainable": true  
            },  
            "token_characters": {  
                "type": "character_encoding",  
                "embedding": {  
                    "embedding_dim": 16  
                },  
                "encoder": {  
                    "type": "cnn",  
                    "embedding_dim": 16,  
                    "num_filters": 128,  
                    "ngram_filter_sizes": [3],  
                    "conv_layer_activation": "relu"  
                }  
            },  
            "encoder": {  
                "type": "LSTM",  
                "input_size": 50 + 128,  
                "hidden_size": 200,  
                "num_layers": 2,  
                "dropout": 0.5,  
                "bidirectional": true  
            },  
            "encoder": {  
                "type": "LSTM",  
                "input_size": 200 + 128,  
                "hidden_size": 200,  
                "num_layers": 2,  
                "dropout": 0.5,  
                "bidirectional": true  
            },  
            "encoder": {  
                "type": "LSTM",  
                "input_size": 200 + 128,  
                "hidden_size": 200,  
                "num_layers": 2,  
                "dropout": 0.5,  
                "bidirectional": true  
            }  
        }  
    }  
}
```

from_params, originally

```
@Model.register("crf_tagger")
class CrfTagger(Model):
    def __init__(
        self,
        vocab: Vocabulary,
        text_field_embedder: TextFieldEmbedder,
        encoder: Seq2SeqEncoder,
        label_namespace: str = "labels",
        constraint_type: str = None,
        include_start_end_transitions: bool = True,
        dropout: float = None,
        initializer: InitializerApplicator = None,
        regularizer: Optional[RegularizerApplicator] =
            None
    ) -> None:
```

- have to write all the parameters twice
- better make sure you use the same default values in both places!
- tedious + error-prone
- the way from_params works should (in most cases) be obvious from the constructor

```
@classmethod
def from_params(cls,
                 vocab: Vocabulary,
                 params: Params) -> 'CrfTagger':
    embedder_params = params.pop("text_field_embedder")
    text_field_embedder = TextFieldEmbedder.from_params(vocab,
                                                         embedder_params)
    encoder = Seq2SeqEncoder.from_params(params.pop("encoder"))
    label_namespace = params.pop("label_namespace", "labels")
    constraint_type = params.pop("constraint_type", None)
    dropout = params.pop("dropout", None)
    include_start_end_transitions = \
        params.pop("include_start_end_transitions", True)
    initializer_params = params.pop('initializer', [])
    initializer = InitializerApplicator.from_params(initializer_params)
    regularizer_params = params.pop('regularizer', [])
    regularizer = RegularizerApplicator.from_params(regularizer_params)
```

```
params.assert_empty(cls.__name__)
```

```
return cls(vocab=vocab,
           text_field_embedder=text_field_embedder,
           encoder=encoder,
           label_namespace=label_namespace,
           constraint_type=constraint_type,
           dropout=dropout,
           include_start_end_transitions=include_start_end_transitions,
           initializer=initializer,
```

from_params, now

```
class FromParams:
    @classmethod
    def from_params(cls: Type[T], params: Params, **extras) -> T:
        from allennlp.common.registerable import Registrable # import here to avoid circular imports

        if params is None: return None

        registered_subclasses = Registrable._registry.get(cls)

        if registered_subclasses is not None:
            as_registerable = cast(Type[Registrable], cls)
            default_to_first_choice = as_registerable.default_implementation is not None
            choice = params.pop_choice("type",
                                         choices=as_registerable.list_available(),
                                         default_to_first_choice=default_to_first_choice)
            subclass = registered_subclasses[choice]

            if not takes_arg(subclass.from_params, 'extras'):
                extras = {k: v for k, v in extras.items() if takes_arg(subclass.from_params, k)}

            return subclass.from_params(params=params, **extras)
        else:
            if cls.__init__ == object.__init__:
                kwargs: Dict[str, Any] = {}
            else:
                kwargs = create_kwargs(cls, params, **extras)

            return cls(**kwargs) # type: ignore
```

from_params, now

```
def create_kwargs(cls: Type[T], params: Params, **extras) -> Dict[str, Any]:
```

```
    """
```

Given some class, a `Params` object, and potentially other keyword arguments,
create a dict of keyword args suitable for passing to the class's constructor.

The function does this by finding the class's constructor, matching the constructor
arguments to entries in the `params` object, and instantiating values for the parameters
using the type annotation and possibly a `from_params` method.

Any values that are provided in the `extras` will just be used as is.
For instance, you might provide an existing `Vocabulary` this way.

```
"""
```

```
...
```

Trainer

```
class Trainer(Registrable):
    def __init__(
        self,
        model: Model,
        optimizer: torch.optim.Optimizer,
        iterator: DataIterator,
        train_dataset: Iterable[Instance],
        validation_dataset: Optional[Iterable[Instance]] =
        None,
        patience: Optional[int] = None,
        validation_metric: str = "-loss",
        validation_iterator: DataIterator = None,
        shuffle: bool = True,
        num_epochs: int = 20,
        serialization_dir: Optional[str] = None,
        num_serialized_models_to_keep: int = 20,
        keep_serialized_model_every_num_seconds: int = None,
        model_save_interval: float = None,
        cuda_device: Union[int, List] = -1,
        grad_norm: Optional[float] = None,
        grad_clipping: Optional[float] = None,
        learning_rate_scheduler: LearningRateScheduler = None,
        summary_interval: int = 100,
        histogram_interval: int = None,
        should_log_parameter_statistics: bool = True,
        should_log_learning_rate: bool = False) -> None:
```

- configurable training loop with tons of options
 - your favorite PyTorch optimizer
 - early stopping
 - many logging options
 - many serialization options
 - learning rate schedulers
- (almost all of them optional)
- as always, configuration happens in your JSON experiment config

Model archives

- training loop produces a model.tar.gz
 - config.json + vocabulary + trained model weights
- can be used with command line tools to evaluate on test datasets or to make predictions
- can be used to power an interactive demo



Making Predictions



Predictor

- models are tensor-in, tensor-out
- for creating a web demo, want JSON-in, JSON-out
- same for making predictions interactively
- Predictor is just a simple JSON wrapper for your model

and this is enabled by all of our models taking *optional* labels and returning an optional loss and also various model internals and interesting results

```
@Predictor.register('sentence-tagger')
class SentenceTaggerPredictor(Predictor):
    def __init__(self,
                 model: Model,
                 dataset_reader: DatasetReader) -> None:
        super().__init__(model, dataset_reader)
        self._tokenizer =
            SpacyWordSplitter(language='en_core_web_sm',
                               pos_tags=True)

    def predict(self, sentence: str) -> JsonDict:
        return self.predict_json({"sentence" : sentence})

    @overrides
    def _json_to_instance(self, json_dict: JsonDict) -> Instance:
        sentence = json_dict["sentence"]
        tokens = self._tokenizer.split_words(sentence)
        return self._dataset_reader.text_to_instance(tokens)
```

this is (partly) why we split out text_to_instance as its own function in the dataset reader

Serving a demo

With this setup, serving a demo is easy.

- DatasetReader gives us `text_to_instance`
- Labels are optional in the model and dataset reader
- Model returns an arbitrary dict, so can get and visualize model internals
- Predictor wraps it all in JSON
- Archive lets us load a pre-trained model in a server
- Even better: pre-built UI components (using React) to visualize standard pieces of a model, like attentions, or span labels

AllenNLP Tutorial

sentence

This is the best tutorial I've ever attended.

PREDICT

```
{
  "tag_logits": [
    [
      2.294965982437134,
      -1.4576776027679443,
      -1.383512020111084
    ],
    [
      2.6445960998535156,
      -1.9713417291641235,
      -1.2443891763687134
    ],
    [
      0.35024261474609375,
      3.0433900356292725,
      -3.6660397052764893
    ],
    [
      2.8452417850494385,
      -0.9580511450767517,
      -2.406663656234741
    ],
    [
      2.919186592102051,
      -1.7659225463867188,
      -1.7427881956100464
    ],
    [
      2.7985565662384033,
      -2.1611685752868652,
      -1.1986668109893799
    ],
    [
      2.8985729217529297,
      -2.2677223682403564,
      -1.2037614583969116
  ]]
```

We don't have it all figured out!

still figuring out some abstractions that we may not have correct

- regularization and initialization
- models with pretrained components
- more complex training loops
 - e.g. multi-task learning
- Caching preprocessed data
- Expanding vocabulary / embeddings at test time
- Discoverability of config options

you can do all these things, but almost certainly not in the most optimal / generalizable way



Case study

In this section, we will use an LSTM to get part of speech tags. We Backward or anything like that, but as a (challenging) exercise to t Viterbi could be used after you have seen what is going on.

The model is as follows: let our input sentence be w_1, \dots, w_M , T be our tag set, and y_i the tag of word w_i . Denote our prediction

This is a structure prediction, model, where our output is a sequen

To do the prediction, pass an LSTM over the sentence. Denote the Also, assign each tag a unique index (like how we had word_to_ix i section). Then our prediction rule for \hat{y}_i is

$$\hat{y}_i = \operatorname{argmax}_j (\log \operatorname{Softmax}(A h_i + b))$$

That is, take the log softmax of the affine map of the hidden state that has the maximum value in this vector. Note this implies imme the target space of A is $|T|$.

Prepare data:

```
def prepare_sequence(seq, to_ix):
    idxs = [to_ix[w] for w in seq]
    return torch.tensor(idxs, dtype=torch.long)

training_data = [
    ("The dog ate the apple".split(), ["DET", "NN",
                                         "VBD", "DT", "VBD", "NN"]),
    ("Everybody read that book".split(), ["NN", "V",
                                           "DT", "VBD", "DT", "NN"])
]
word_to_ix = {}
for sent, tags in training_data:
    for word in sent:
        if word not in word_to_ix:
            word_to_ix[word] = len(word_to_ix)
```

"an LSTM for part-of speech tagging"

(based on [the official PyTorch tutorial](#))

The Problem

Given a training dataset that looks like

```
The###DET dog###NN ate###V the###DET apple###NN  
Everybody###NN read###V that###DET book###NN
```

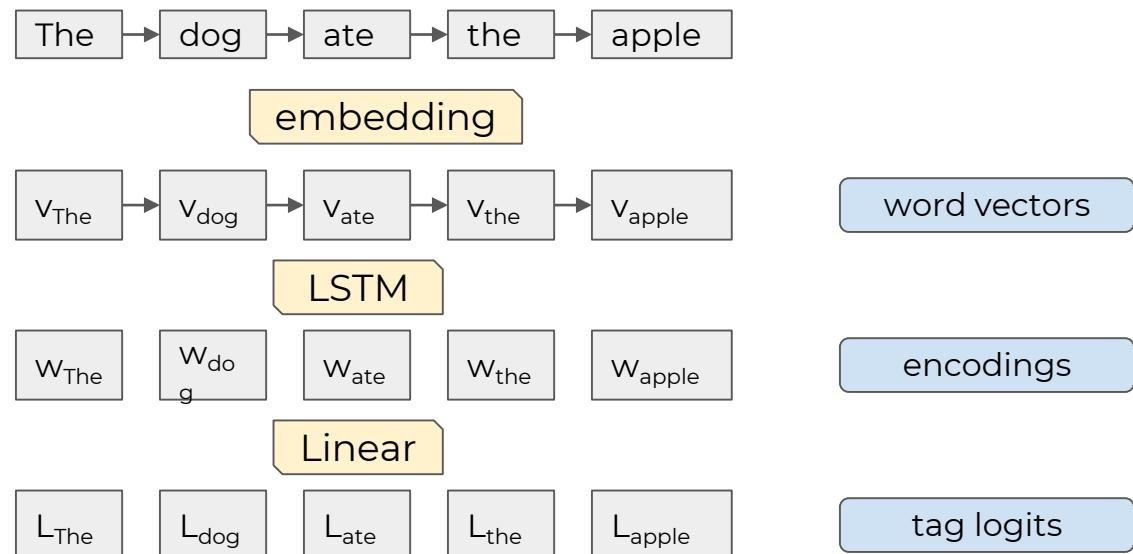
learn to predict part-of-speech tags

With a Few Enhancements to Make Things More Realistic

- read data from files
- check performance on a separate validation dataset
- use `tqdm` to track training progress
- implement early stopping based on validation loss
- track accuracy as we're training

Start With a Simple Baseline Model

- compute a vector embedding for each word
- feed the sequence of embeddings into an LSTM
- feed the hidden states into a feed-forward layer to produce a sequence of logits



v0: numpy

aka "this is why we use libraries"

v0: numpy (aka "this is why we use libraries")

```
class LSTM:  
    def __init__(self, input_size: int, hidden_size: int) -> None:  
        self.params = {  
            # forget gate  
            "w_f": np.random.randn(input_size, hidden_size)  
            "b_f": np.random.randn(hidden_size)  
            "u_f": np.random.randn(hidden_size, hidden_size)  
  
            # external input gate  
            "w_g": np.random.randn(input_size, hidden_size)  
            "b_g": np.random.randn(hidden_size)  
            "u_g": np.random.randn(hidden_size, hidden_size)  
  
            # output gate  
            "w_q": np.random.randn(input_size, hidden_size)  
            "b_q": np.random.randn(hidden_size)  
            "u_q": np.random.randn(hidden_size, hidden_size)  
  
            # usual params  
            "w": np.random.randn(input_size, hidden_size)  
            "b": np.random.randn(hidden_size)  
            "u": np.random.randn(hidden_size, hidden_size)  
        }  
  
        self.grads = {name: None for name in self.params}
```



v1: PyTorch

v1: PyTorch - Load Data

```
def load_data(file_path: str) -> List[Tuple[str, str]]:  
    """  
        One sentence per line, formatted like  
        The###DET dog###NN ate###V the###DET apple###NN  
  
        Returns a list of pairs (tokenized_sentence, tags)  
    """  
    data = []  
  
    with open(file_path) as f:  
        for line in f:  
            pairs = line.strip().split()  
            sentence, tags = zip(*(pair.split("###") for pair in pairs))  
            data.append((sentence, tags))  
  
    return data
```

seems reasonable

v1: PyTorch - Define Model

```
class LSTMTagger(nn.Module):
    def __init__(self, embedding_dim: int, hidden_dim: int, vocab_size: int, tagset_size: int) -> None:
        super().__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # The LSTM takes word embeddings as inputs,
        # and outputs hidden states with dimensionality hidden_dim.
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)

        # The linear layer that maps from hidden state space to tag space
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
        self.hidden = self.init_hidden()

    def forward(self, sentence: torch.Tensor) -> torch.Tensor:
        embeds = self.word_embeddings(sentence)
        lstm_out, self.hidden = self.lstm(embeds.view(len(sentence), 1, -1), self.hidden)
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

much nicer than writing
our own LSTM!

v1: PyTorch - Train Model

```
model = LSTMTagger(EMBEDDING_DIM, HIDDEN_DIM,
                    len(word_to_ix), len(tag_to_ix))
loss_function = nn.NLLLoss()
optimizer = optim.SGD(model.parameters(), lr=0.1)

validation_losses = []
patience = 10

for epoch in range(1000):
    training_loss = 0.0
    validation_loss = 0.0

    for dataset, training in [(training_data, True),
                               (validation_data, False)]:
        correct = total = 0
        torch.set_grad_enabled(training)
        t = tqdm.tqdm(dataset)
        for i, (sentence, tags) in enumerate(t):
            model.zero_grad()
            model.hidden = model.init_hidden()

            sentence_in = prepare_sequence(sentence, word_to_ix)
            targets = prepare_sequence(tags, tag_to_ix)

            tag_scores = model(sentence_in)

            loss = loss_function(tag_scores, targets)

            predictions = tag_scores.max(-1)[1]
            correct += (predictions == targets).sum().item()
            total += len(targets)
            accuracy = correct / total

            if training:
                loss.backward()
                training_loss += loss.item()
                t.set_postfix(training_loss=training_loss/(i + 1),
                              accuracy=accuracy)
                optimizer.step()
            else:
                validation_loss += loss.item()
                t.set_postfix(validation_loss=validation_loss/(i + 1),
                              accuracy=accuracy)

        validation_losses.append(validation_loss)

        if (patience and
            len(validation_losses) >= patience and
            validation_losses[-patience] ==
            min(validation_losses[-patience:])):
            print("patience reached, stopping early")
            break
```

this part is maybe less than ideal

v2: AllenNLP

(but without config files)

v2: AllenNLP - Dataset Reader

```
class PosDatasetReader(DatasetReader):
    def __init__(self, token_indexers: Dict[str, TokenIndexer] = None) -> None:
        super().__init__(lazy=False)
        self.token_indexers = token_indexers or {"tokens": SingleIdTokenIndexer()}

    def text_to_instance(self, tokens: List[Token], tags: List[str] = None) -> Instance:
        sentence_field = TextField(tokens, self.token_indexers)
        fields = {"sentence": sentence_field}

        if tags:
            label_field = SequenceLabelField(labels=tags, sequence_field=sentence_field)
            fields["labels"] = label_field

        return Instance(fields)

    def _read(self, file_path: str) -> Iterator[Instance]:
        with open(file_path) as f:
            for line in f:
                pairs = line.strip().split()
                sentence, tags = zip(*pair.split("###") for pair in pairs))
                yield self.text_to_instance([Token(word) for word in sentence], tags)
```

v2: AllenNLP - Model

```
class LstmTagger(Model):
    def __init__(self, word_embeddings: TextFieldEmbedder, encoder: Seq2SeqEncoder, vocab: Vocabulary) -> None:
        super().__init__(vocab)
        self.word_embeddings = word_embeddings
        self.encoder = encoder
        self.hidden2tag = torch.nn.Linear(in_features=encoder.get_output_dim(),
                                         out_features=vocab.get_vocab_size('labels'))
        self.accuracy = CategoricalAccuracy()

    def forward(self, sentence: Dict[str, torch.Tensor], labels: torch.Tensor = None) -> torch.Tensor:
        mask = get_text_field_mask(sentence)
        embeddings = self.word_embeddings(sentence)
        encoder_out = self.encoder(embeddings, mask)

        tag_logits = self.hidden2tag(encoder_out)
        output = {"tag_logits": tag_logits}

        if labels is not None:
            self.accuracy(tag_logits, labels, mask)
            output["loss"] = sequence_cross_entropy_with_logits(tag_logits, labels, mask)

    return output

    def get_metrics(self, reset: bool = False) -> Dict[str, float]:
        return {"accuracy": self.accuracy.get_metric(reset)}
```

v2: AllenNLP - Training

```
reader = PosDatasetReader()
train_dataset =
    reader.read(cached_path('https://raw.githubusercontent.com/allenai/allennlp/master/tutorials/tagger/training.txt'))
validation_dataset = reader.read(
    cached_path('https://raw.githubusercontent.com/allenai/allennlp/master/tutorials/tagger/validation.txt'))

vocab = Vocabulary.from_instances(train_dataset + validation_dataset)

EMBEDDING_DIM = 6
HIDDEN_DIM = 6

token_embedding = Embedding(num_embeddings=vocab.get_vocab_size('tokens'), embedding_dim=EMBEDDING_DIM)
word_embeddings = BasicTextFieldEmbedder({'tokens': token_embedding})
lstm = PytorchSeq2SeqWrapper(torch.nn.LSTM(EMBEDDING_DIM, HIDDEN_DIM, batch_first=True))
model = LstmTagger(word_embeddings, lstm, vocab)

optimizer = optim.SGD(model.parameters(), lr=0.1)
iterator = BucketIterator(batch_size=2, sorting_keys=[("sentence", "num_tokens")])
iterator.index_with(vocab)

trainer = Trainer(model=model, optimizer=optimizer, iterator=iterator,
                  train_dataset=train_dataset, validation_dataset=validation_dataset,
                  patience=10, num_epochs=1000)

trainer.train()
```

this is where the config-driven approach would make our lives a lot easier

v3: AllenNLP + config

v3: AllenNLP - config

```
local embedding_dim = 6;
local hidden_dim = 6;
local num_epochs = 1000;
local patience = 10;
local batch_size = 2;
local learning_rate = 0.1;

{
    "train_data_path": "...",
    "validation_data_path": "...",
    "dataset_reader": { "type": "pos-tutorial" },
    "model": {
        "type": "lstm-tagger",
        "word_embeddings": {
            "token_embedders": {
                "tokens": {
                    "type": "embedding",
                    "embedding_dim": embedding_dim
                }
            }
        },
        "encoder": {
            "type": "lstm",
            "input_size": embedding_dim,
            "hidden_size": hidden_dim
        }
    },
    "iterator": {
        "type": "bucket",
        "batch_size": batch_size,
        "sorting_keys": [["sentence", "num_tokens"]]
    },
    "trainer": {
        "num_epochs": num_epochs,
        "optimizer": {
            "type": "sgd",
            "lr": learning_rate
        },
        "patience": patience
    }
}

params = Params.from_file('...')
serialization_dir = tempfile.mkdtemp()
model = train_model(params, serialization_dir)
```

Augmenting the Tagger with Character-Level Features

EXERCISE: AUGMENTING THE LSTM PA WITH CHARACTER-LEVEL FEATURES

In the example above, each word had an embedding, which our model. Let's augment the word embeddings with a representation of the word. We expect that this should help significantly, since affixes have a large bearing on part-of-speech. For example, always tagged as adverbs in English.

To do this, let c_w be the character-level representation of word w , as before. Then the input to our sequence model is the concatenated vector of dimension 5, and c_w dimension 3, then our LSTM should also have dimension 8.

To get the character level representation, do an LSTM over the final hidden state of this LSTM. Hints:

- There are going to be two LSTM's in your new model. One that takes the concatenated vector as input and outputs word scores, and the new one that outputs a character-level representation.
- To do a sequence model over characters, you will have to loop over the words. The word embeddings will be the input to the character LSTM.

Total running time of the script: (0 minutes 1.250 seconds)

Download Python source code: [sequence_models_tutorial.py](#)

Download Jupyter notebook: [sequence_models_tutorial.ipynb](#)

v1: PyTorch

```
class LSTMTagger(nn.Module):
    def __init__(self, embedding_dim: int, hidden_dim: int,
                 vocab_size: int, tagset_size: int) -> None:
        super().__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # The LSTM takes word embeddings as inputs,
        # and outputs hidden states with dimensionality hidden_dim.
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)

        # Linear layer that maps from hidden state space to tag space
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
        self.hidden = self.init_hidden()

    def forward(self, sentence: torch.Tensor) -> torch.Tensor:
        embeds = self.word_embeddings(sentence)
        lstm_out, self.hidden = self.lstm(embeds.view(len(sentence), 1,
-1), self.hidden)
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

add char_embedding_dim

add char_embedding layer =
embedding + LSTM?

change LSTM input dim

compute char
embeddings

concatenate inputs

we really have to change our model code and how it works

v1: PyTorch

```
class LSTMTagger(nn.Module):
    def __init__(self, embedding_dim: int, hidden_dim: int,
                 vocab_size: int, tagset_size: int) -> None:
        super().__init__()
        self.hidden_dim = hidden_dim

        self.word_embeddings = nn.Embedding(vocab_size, embedding_dim)

        # The LSTM takes word embeddings as inputs,
        # and outputs hidden states with dimensionality hidden_dim.
        self.lstm = nn.LSTM(embedding_dim, hidden_dim)

        # Linear layer that maps from hidden state space to tag space
        self.hidden2tag = nn.Linear(hidden_dim, tagset_size)
        self.hidden = self.init_hidden()

    def forward(self, sentence: torch.Tensor) -> torch.Tensor:
        embeds = self.word_embeddings(sentence)
        lstm_out, self.hidden = self.lstm(embeds.view(len(sentence), 1,
-1), self.hidden)
        tag_space = self.hidden2tag(lstm_out.view(len(sentence), -1))
        tag_scores = F.log_softmax(tag_space, dim=1)
        return tag_scores
```

I'm not really
that thrilled to
do this exercise

v2: AllenNLP

```
reader = PosDatasetReader()
```

```
# ...
```

```
EMBEDDING_DIM = 6
```

```
HIDDEN_DIM = 6
```

```
# ...
```

```
token_embedding = Embedding(  
    num_embeddings=vocab.get_vocab_size('tokens'),  
    embedding_dim=EMBEDDING_DIM)
```

add a second
token indexer

```
reader = PosDatasetReader(token_indexers={  
    "tokens": SingleIdTokenIndexer(),  
    "token_characters": TokenCharactersIndexer()  
})
```

```
# ...
```

```
WORD_EMBEDDING_DIM = 5
```

```
CHAR_EMBEDDING_DIM = 3
```

```
EMBEDDING_DIM = WORD_EMBEDDING_DIM + CHAR_EMBEDDING_DIM
```

```
HIDDEN_DIM = 6
```

```
# ...
```

```
token_embedding = Embedding(  
    num_embeddings=vocab.get_vocab_size('tokens'),  
    embedding_dim=WORD_EMBEDDING_DIM)
```

add a
character
embedder

```
char_embedding = TokenCharactersEncoder(  
    embedding=Embedding(  
        num_embeddings=vocab.get_vocab_size('token_characters'),  
        embedding_dim=CHAR_EMBEDDING_DIM),  
    encoder=PytorchSeq2VecWrapper(  
        torch.nn.LSTM(CHAR_EMBEDDING_DIM, CHAR_EMBEDDING_DIM,  
        batch_first=True))
```

```
word_embeddings = BasicTextFieldEmbedder(  
    {"tokens": token_embedding})
```

```
# ...
```

use the
character
embedder

```
word_embeddings = BasicTextFieldEmbedder({  
    "tokens": token_embedding,  
    "token_characters": char_embedding})
```

```
# ...
```

no
changes
to the
model
itself!

v3: AllenNLP - config

```
local embedding_dim = 6;
local hidden_dim = 6;
local num_epochs = 1000;
local patience = 10;
local batch_size = 2;
local learning_rate = 0.1;

{
    "train_data_path": "...",
    "validation_data_path": "...",
    "dataset_reader": { "type": "pos-tutorial" },
    "model": {
        "type": "lstm-tagger",
        "word_embeddings": {
            "token_embedders": {
                "tokens": {
                    "type": "embedding",
                    "embedding_dim": embedding_dim
                }
            }
        },
        "encoder": {
            "type": "lstm",
            "input_size": embedding_dim,
            "hidden_size": hidden_dim
        }
    },
}
```

we can
accomplish this
with just a couple
of minimal config
changes

v3: AllenNLP - config

```
local embedding_dim = 6;  
local hidden_dim = 6;  
local num_epochs = 1000;  
local patience = 10;  
local batch_size = 2;  
local learning_rate = 0.1;
```



```
local word_embedding_dim = 5;  
local char_embedding_dim = 3;  
local embedding_dim = word_embedding_dim + char_embedding_dim;  
local hidden_dim = 6;  
local num_epochs = 1000;  
local patience = 10;  
local batch_size = 2;  
local learning_rate = 0.1;
```

add a couple of new Jsonnet variables

v3: AllenNLP - config

```
"dataset_reader": { "type": "pos-tutorial" }
```

```
}
```

```
"dataset_reader": {  
    "type": "pos-tutorial",  
    "token_indexers": {  
        "tokens": { "type": "single_id" },  
        "token_characters": { "type": "characters" }  
    }  
}
```



add a second token indexer

v3: AllenNLP - config

```
"model": {  
    "type": "lstm-tagger",  
    "word_embeddings": {  
        "token_embedders": {  
            "tokens": {  
                "type": "embedding",  
                "embedding_dim": embedding_dim  
            }  
        }  
    },  
    "encoder": {  
        "type": "lstm",  
        "input_size": embedding_dim,  
        "hidden_size": hidden_dim  
    }  
}
```

add a corresponding token embedder

```
"model": {  
    "type": "lstm-tagger",  
    "word_embeddings": {  
        "token_embedders": {  
            "tokens": {  
                "type": "embedding",  
                "embedding_dim": word_embedding_dim  
            },  
            "token_characters": {  
                "type": "character_encoding",  
                "embedding": {  
                    "embedding_dim": char_embedding_dim,  
                },  
                "encoder": {  
                    "type": "lstm",  
                    "input_size": char_embedding_dim,  
                    "hidden_size": char_embedding_dim  
                }  
            }  
        },  
        "encoder": {  
            "type": "lstm",  
            "input_size": embedding_dim,  
            "hidden_size": hidden_dim  
        }  
    }  
}
```

**For a one-time change this is
maybe not such a big win.**

**But being able to experiment
with lots of architectures
without having to change any
code (and with a reproducible
JSON description of each
experiment) is a huge boon to
research! (we think)**

Sharing Your Research

*How to make it easy to release
your code*



In the least amount of time possible:



Simplify your workflow
for installation and data



Make your code run
anywhere*



Isolated environments
for your project

Docker



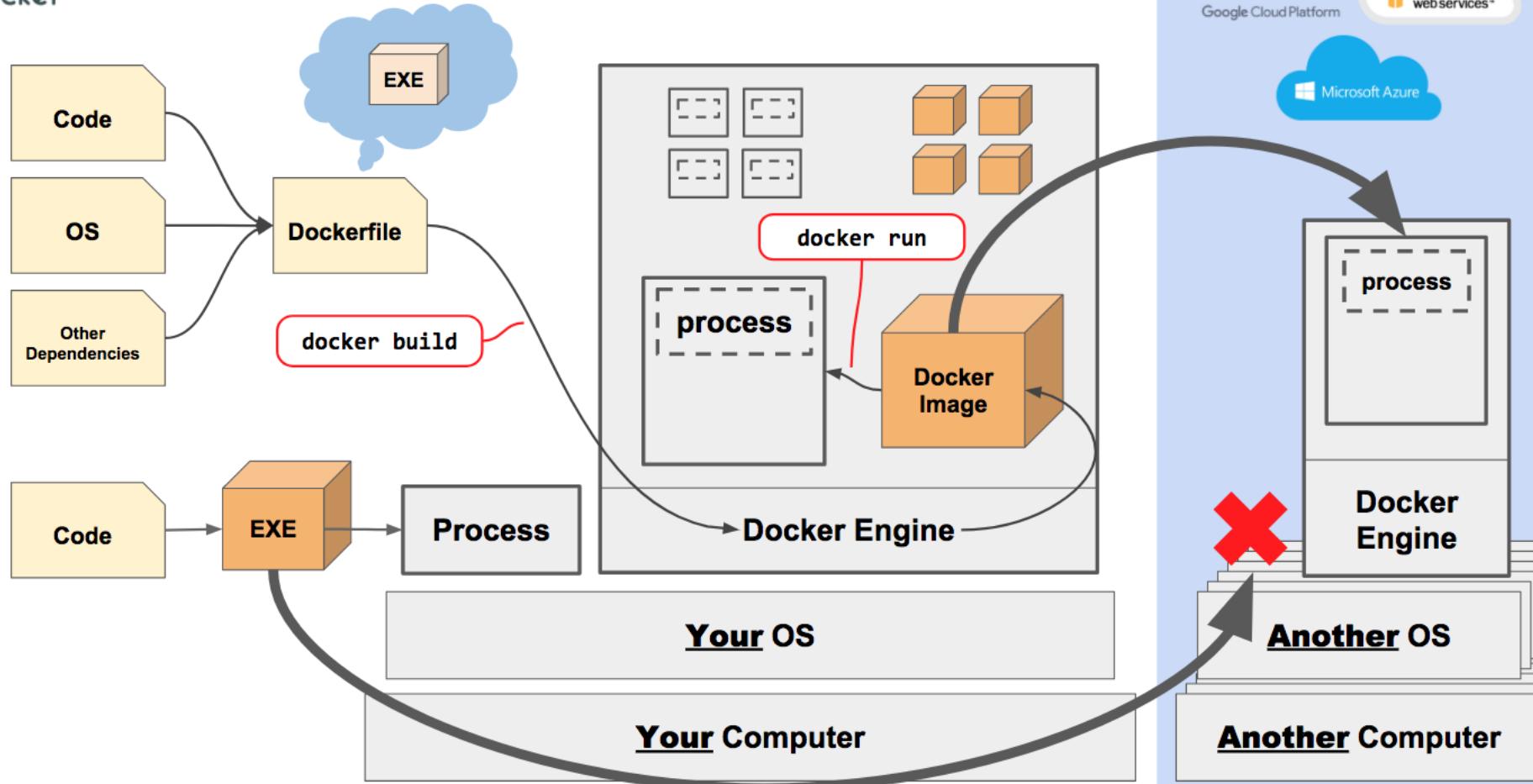
Objective: You don't feel like this about Docker

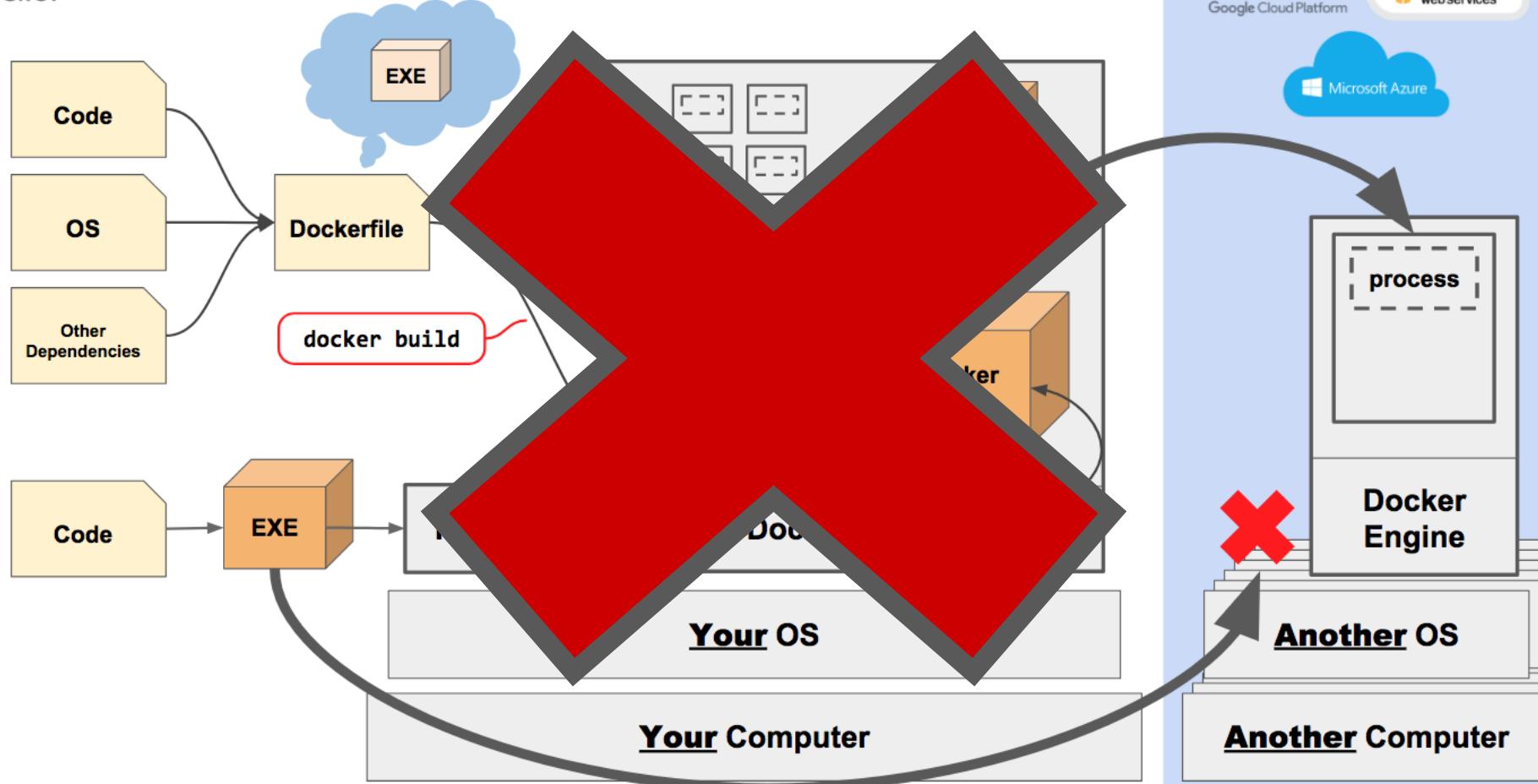


What does Docker Do?

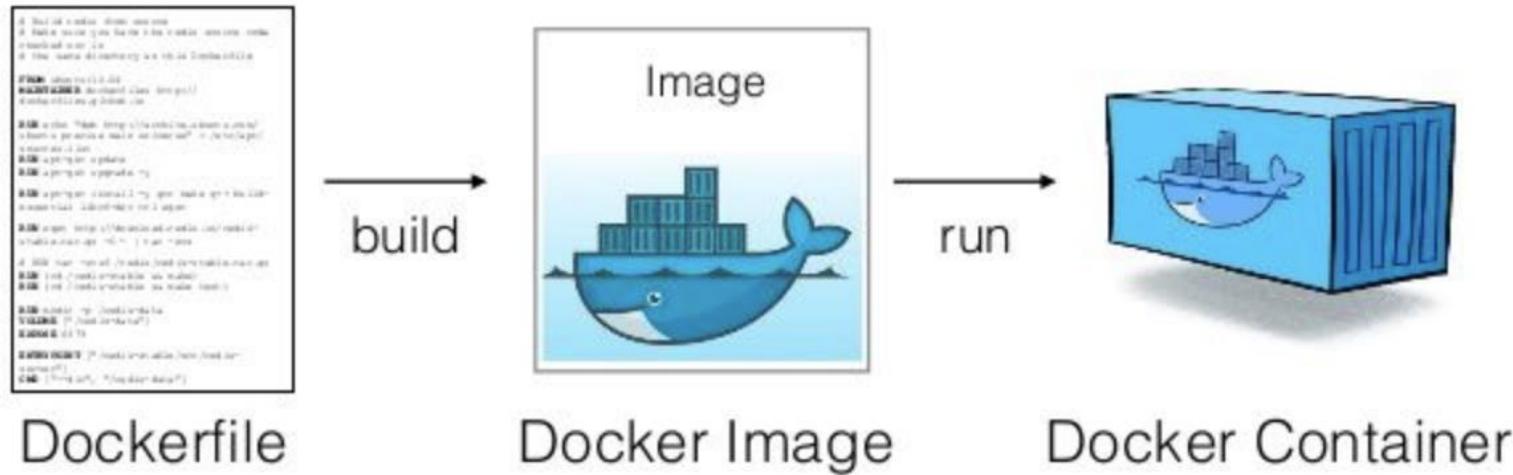
- Creates a virtual machine that will always run the same anywhere (In theory)
- Allows you to package up a virtual machine and some code and send it to someone, knowing the same thing will run
- Includes operating systems, dependencies for your code, your code etc.
- Let's you specify in a series of steps how to create this virtual machine and does clever caching when you change it.







3 Ideas: Dockerfiles, Images and Containers



Step 1: Write a Dockerfile

Here is a finished Dockerfile.

How does this work?

```
FROM python:3.6.3-jessie

ENV LC_ALL=C.UTF-8
ENV LANG=C.UTF-8


# This specifies where the current working
# directory is when we start copying files in.
WORKDIR /stage/allennlp


# Install base packages like gcc, git etc.
RUN apt-get update --fix-missing && apt-get install -y \
    bzip2 \
    ca-certificates \
    curl \
    gcc \
    git \
    libc-dev \
    libglib2.0-0 \
    libsm6 \
    libxext6 \
    libxrender1 \
    wget \
    libevent-dev \
    build-essential && \
    rm -rf /var/lib/apt/lists/*

# Optional argument to set an environment variable with the Git SHA
ARG SOURCE_COMMIT
ENV SOURCE_COMMIT $SOURCE_COMMIT


# This exposes port 8000 to outside of the Docker container.
# This is helpful if you want to run a model server or something.
EXPOSE 8000

CMD ["/bin/bash"]
```

Step 1: Write a Dockerfile

COMMAND <command>

Step 1: Write a Dockerfile

COMMAND <command>

Dockerfile commands are capitalised. Some important ones are:

**FROM, RUN, ENV, COPY
and CMD**

Step 1: Write a Dockerfile

```
FROM python:3.6.3-jessie
```

FROM includes another Dockerfile in your one. Here we start from a base Python Dockerfile.

Step 1: Write a Dockerfile

```
RUN pip install -r requirements.txt
```

RUN ... runs a command.
To use a command, it must
be installed in a previous
step!

Step 1: Write a Dockerfile

```
ENV LANG=C.UTF-8
```

ENV sets an environment variable which can be used inside the container.

Step 1: Write a Dockerfile

```
COPY my_research/ my_research/
```

COPY copies code from
your current folder into the
Docker image.

Step 1: Write a Dockerfile

```
COPY my_research/ my_research/
```

Do yourself a favour.
Don't change the names
of things during this
step.

Step 1: Write a Dockerfile

```
CMD [“/bin/bash”]  
CMD [“python”, “my/script.py”]
```

CMD is what gets run
when you *run* a built
image.

Step 1: Write a Dockerfile

Here is a finished
Dockerfile.

```
FROM python:3.6.3-jessie

ENV LC_ALL=C.UTF-8
ENV LANG=C.UTF-8


# This specifies where the current working
# directory is when we start copying files in.
WORKDIR /stage/allennlp


# Install base packages like gcc, git etc.
RUN apt-get update --fix-missing && apt-get install -y \
    bzip2 \
    ca-certificates \
    curl \
    gcc \
    git \
    libc-dev \
    libglib2.0-0 \
    libsm6 \
    libxext6 \
    libxrender1 \
    wget \
    libevent-dev \
    build-essential && \
    rm -rf /var/lib/apt/lists/*

# Optional argument to set an environment variable with the Git SHA
ARG SOURCE_COMMIT
ENV SOURCE_COMMIT $SOURCE_COMMIT


# This exposes port 8000 to outside of the Docker container.
# This is helpful if you want to run a model server or something.
EXPOSE 8000

CMD ["/bin/bash"]
```

Step 2: Build your Dockerfile into an Image

```
docker build --tag <name> .
```

Step 2: Build your Dockerfile into an Image

```
docker build --tag <name> .
```

This is what you want the image to be called, e.g markn/my-paper-code.

Step 2: Build your Dockerfile into an Image

```
docker build --tag <name> .
```

You can see what images you have built already by running
docker images

Step 2: Build your Dockerfile into an Image

```
docker build --tag <name> .
```

This describes where docker should look for a Dockerfile. It can also be a URL.

Step 2: Build your Dockerfile into an Image

```
docker build --tag <name> .
```

If you've already built a line of your dockerfile before, Docker will remember and not build it again (so long as things before it haven't changed.)

```
Sending build context to Docker daemon 192.3MB
Step 1/32 : FROM python:3.6.3-jessie
--> 79e1dc9af1c1
Step 2/32 : ENV LC_ALL C.UTF-8
--> Using cache
--> 661329a57650
Step 3/32 : ENV LANG C.UTF-8
--> Using cache
--> 623908bc3a31
```

Step 2: Build your Dockerfile into an Image

```
docker build --tag <name> .
```

TIP: Put things that change more frequently (like your code) lower down in your Dockerfile.

```
Sending build context to Docker daemon 192.3MB
Step 1/32 : FROM python:3.6.3-jessie
--> 79e1dc9af1c1
Step 2/32 : ENV LC_ALL C.UTF-8
--> Using cache
--> 661329a57650
Step 3/32 : ENV LANG C.UTF-8
--> Using cache
--> 623908bc3a31
```

Step 3: Run your Image as a Container

```
docker run <image-name>
```

Step 3: Run your Image as a Container

```
docker run -it <image-name>
```

-i: interactive

-t: tty (with a terminal)

Step 3: Run your Image as a Container

```
docker run -it -e /bin/bash ...
```

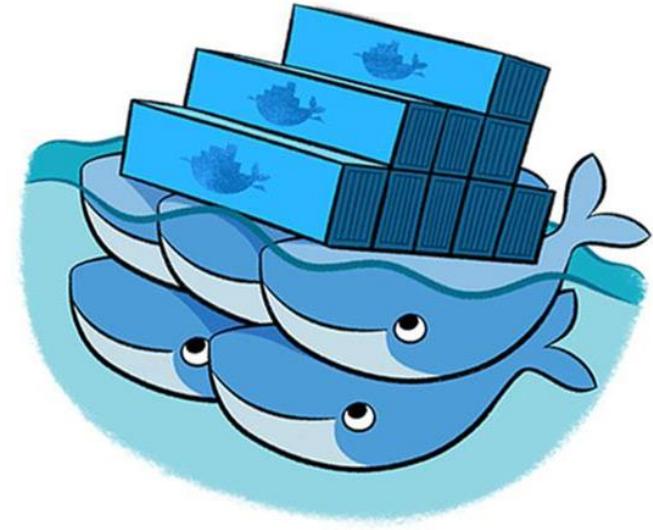
These arguments will give you a command prompt inside any docker container, regardless of the CMD in the Dockerfile.

Optional Step 4: DockerHub

DockerHub is to **Docker** as **Github** is to **Git**

Docker automatically looks at dockerhub to find Docker images to run

`docker push <image-name>`



Screenshot of a DockerHub user profile page for 'allenlp' showing a list of public Docker images:

Image Name	Type	Stars	Pulls	Details
allenlp/allennlp	public	6 STARS	100K+ PULLS	> DETAILS
allenlp/build	public	0 STARS	1.5K PULLS	> DETAILS
allenlp/bilm-tf	public	0 STARS	966 PULLS	> DETAILS
allenlp/commit	public	0 STARS	487 PULLS	> DETAILS
allenlp/demo	public	0 STARS	484 PULLS	> DETAILS
allenlp/allennlp-conda-linux-env	public	0 STARS	41 PULLS	> DETAILS

Pros of Docker

- Good for running CI - ALL your code dependencies are pinned, even system level stuff.
- Good for debugging people's problems with your code - just ask: Can you reproduce bug that in a Docker Container
- Great for deploying demos where you just need a model to run as a service.



Cons of Docker

- ✖ Docker is designed for production systems - it is very hard to debug inside a minimal docker container
- ✖ Takes up a lot of memory if you have a lot of large dependencies (e.g the JVM makes up about half of the AllenNLP Docker image)
- ✖ Just because your code is exactly reproducible doesn't mean that it's any good



Releasing your data

Use a simple file cache

First, download and unzip GloVe vectors from the Stanford NLP group website, with:

```
chmod +x download.sh; ./download.sh
```

Then prepare vocabulary and initial word vectors with:

```
python prepare_vocab.py dataset/tacred dataset/vocab --glove_dir dataset/glove
```

This will write vocabulary and word vectors as a numpy matrix into the dir `dataset/vocab`.

There are currently 27 CoreNLP Jar files you could download from the CoreNLP website

Data

We provide Quasar-T, SearchQA and TrivialQA dataset we used for the task in `data/` directory. We preprocess the original data to make it satisfy the input format of our codes, and can be download at [here](#).

To run our code, the dataset should be put in the folder `data/` using the following format:

`datasets/`

- `train.txt, dev.txt, test.txt`: format for each line: `{"question": question, "answers": [answer1, answer2, ...]}`.
- `train.json, dev.json, test.json`: format `[{"question": question, "document": document1}, {"question": question, "document": document2}, ...]`.

`embeddings/`

- `glove.840B.300d.txt`: word vectors obtained from [here](#).

`corenlp/`

- all jar files from Stanford CoreNLP.

Use a simple file cache

```
embedding_file = cached_path("embedding_url")
datasets = cached_path("dataset_url")
```

Data

We provide Quasar-T, SearchQA and TrivialQA dataset we used for the task in data/ directory. We preprocess the original data to make it satisfy the input format of our codes, and can be download at [here](#).

To run our code, the dataset should be put in the folder data/ using the following format:

datasets/

- train.txt, dev.txt, test.txt: format for each line: {"question": question, "answers": [answer1, answer2, ...]}.
- train.json, dev.json, test.json: format [{"question": question, "document": document1}, {"question": question, "document": document2}, ...].

embeddings/

- glove.840B.300d.txt: word vectors obtained from [here](#).

corenlp/

- all jar files from Stanford Corenlp.

Use a simple file cache

**But now I have to
write a file cache**



Use a simple file cache

Copy [this](#) file into
your project

```
from file_cache import  
cached_path  
embeddings = cached_path(url)
```

Isolated (Python) environments

Python environments

Stable environments
for Python can be
tricky

This makes releasing
code very annoying



Python environments

Docker is ideal, but not great for developing locally. For this, you should either use **virtualenvs** or **anaconda**.



Here we will talk about **anaconda**, because it's what we use.

Python environments

Anaconda is a very stable distribution of Python (amongst other things). Installing it is easy:

<https://www.anaconda.com/>



Python environments

One annoying install step - adding
where you installed it to the **front**
of your PATH environment
variable.



```
export PATH="/path/to/anaconda/bin:$PATH"
```

Python environments

Now, your default python should
be an anaconda one (*you did
install python > 3.6, didn't you*).



ANACONDA®

```
markn@markn ~ $ python
Python 3.6.3 |Anaconda, Inc.| (default, Oct  6 2017, 12:04:38)
[GCC 4.2.1 Compatible Clang 4.0.1 (tags/RELEASE_401/final)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Virtual environments

Every time you start a new project,
make a new virtual environment
which has only its dependencies
in.

```
conda create -n  
your-env-name  
python=3.6
```

Virtual environments

Before you work on your project, run this command. This prepends the location of this particular copy of Python to your PATH.

**source activate
your-project-name**

**pip install -r
requirements.txt**

etc

Virtual environments

When you're done, or you want to work on a different project, run:

```
source deactivate  
your-project-name
```

In Conclusion

In Conclusion

- Prototype fast (but still safely)
- Write production code safely (but still fast)
- Good processes => good science
- Use the right abstractions
- Check out [AllenNLP](#)

Thanks for Coming!

Questions?

please fill out our survey:
<https://tinyurl.com/emnlp-tutorial-survey>

will tweet out link to slides after talk
@ai2_allennlp

