Libraries and Tools
🤗 Transformers, AllenNLP

LING575 Analyzing Neural Language Models
Shane Steinert-Threlkeld
Apr 27 2022

Outline

- Very helpful tools
  - 🤗 Transformers
  - AllenNLP
    - Walk-through of a classifier and a tagger
- Second half: tips/tricks for experiment running and paper writing

🤗 Transformers

https://huggingface.co/transformers

Where to get LMs to analyze?

- RNNs: see week 3 slides
  - Josefewicz et al "Exploring the limits…"
  - Gulordava et al "Colorless green ideas…"
  - ELMo via AllenNLP (about which more later)
- Effectively a unique API for each model
- All (essentially) Transformer-based models: HuggingFace!

Overview of the Library

- Access to many variants of many very large LMs (BERT, RoBERTa, XLNET, ALBERT, T5, language-specific models, …) with fairly consistent API
  - Build tokenizer + model from string for name or config
  - Then use just like any PyTorch nn.Module
- Emphasis on ease-of-use
  - E.g. low barrier-to-entry to *using* the models, including for analysis
  - [new `pipeline` abstraction too, but I think this is *too easy* for most analysis / probing purposes, but can work if all you need are model judgments on data]
- Interoperable with PyTorch or TensorFlow 2.0

Example: Tokenization

```
>>> from transformers import AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

```
>>> encoded_input = tokenizer("Do not meddle in the affairs of wizards, for they are subtle and quick to
>>> print(encoded_input)
{'input_ids': [101, 2079, 2025, 19960, 10362, 1999, 1996, 3821, 1997, 16657, 1010, 2005, 2027, 2024, 112
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

See http://juditacs.github.io/2019/02/19/bert-tokenization-stats.html (h/t Naomi Shapiro)

Example: Tokenization

```
>>> from transformers import AutoTokenizer

>>> tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")
```

```
>>> encoded_input = tokenizer("Do not meddle in the affairs of wizards, for they are subtle and quick to
>>> print(encoded_input)
{'input_ids': [101, 2079, 2025, 19960, 10362, 1999, 1996, 3821, 1997, 16657, 1010, 2005, 2027, 2024, 112
 'token_type_ids': [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 'attention_mask': [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

```
>>> tokenizer.decode(encoded_input["input_ids"])
'[CLS] Do not meddle in the affairs of wizards, for they are subtle and quick to anger. [SEP]'
```

See http://juditacs.github.io/2019/02/19/bert-tokenization-stats.html (h/t Naomi Shapiro)

Example: Tokenizing a Batch

```
>>> batch_sentences = [
...     "But what about second breakfast?",
...     "Don't think he knows about second breakfast, Pip.",
...     "What about elevensies?",
... ]
>>> encoded_input = tokenizer(batch_sentences, padding=True)
>>> print(encoded_input)
{'input_ids': [[101, 1252, 1184, 1164, 1248, 6462, 136, 102, 0, 0, 0, 0, 0, 0, 0],
               [101, 1790, 112, 189, 1341, 1119, 3520, 1164, 1248, 6462, 117, 21902, 1643, 119, 102],
               [101, 1327, 1164, 5450, 23434, 136, 102, 0, 0, 0, 0, 0, 0, 0, 0]],
 'token_type_ids': [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                    [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]]}
```

Example: Tokenizing a Batch

```
>>> batch_sentences = [
...     "But what about second breakfast?",
...     "Don't think he knows about second breakfast, Pip.",
...     "What about elevensies?",
... ]
>>> encoded_input = tokenizer(batch_sentences, padding=True)
>>> print(encoded_input)
{'input_ids': [[101, 1252, 1184, 1164, 1248, 6462, 136, 102, 0, 0, 0, 0, 0, 0, 0],
               [101, 1790, 112, 189, 1341, 1119, 3520, 1164, 1248, 6462, 117, 21902, 1643, 119, 102],
               [101, 1327, 1164, 5450, 23434, 136, 102, 0, 0, 0, 0, 0, 0, 0, 0]],
 'token_type_ids': [[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
                    [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]],
 'attention_mask': [[1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0],
                    [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
                    [1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]]}
```

Add \`return_tensors="pt"\` to get these outputs as PyTorch Tensors

Example: Forward Pass

```
>>> from transformers import BertTokenizer, BertModel
>>> import torch

>>> tokenizer = BertTokenizer.from_pretrained("bert-base-uncased")
>>> model = BertModel.from_pretrained("bert-base-uncased")

>>> inputs = tokenizer("Hello, my dog is cute", return_tensors="pt")
>>> outputs = model(**inputs)

>>> last_hidden_states = outputs.last_hidden_state
```

Outputs from the forward pass

- Outputs are *usually* Python objects with various attributes corresponding to different model outputs (NB: can be *tuples of Tensors* if specified, but I recommend against that)
  - BERT, by default, gives two things:
    - **`last_hidden_state`**: sequence of hidden states at the last layer of the model.
      Shape: (batch_size, max_length, embedding_dimension)
    - **`pooler_output`**: embedding of '[CLS]' token, passed through one tanh layer (more on this later)
      Shape: (batch_size, embedding_dimension)

Getting more out of a model

```
from transformers import BertModel
model = BertModel.from_pretrained(“bert-base-uncased”)

outputs = model(inputs, output_hidden_states=True,
output_attentions=True)
```

Getting more out of a model

```
from transformers import BertModel
model = BertModel.from_pretrained("bert-base-uncased")

outputs = model(inputs, output_hidden_states=True,
output_attentions=True)
```

- Now, the output object has additional attributes:
  - **`hidden_states`**: A tuple of tensors, one for each layer. Length: # layers
    Shape of each: `(batch_size, max_length, embedding_dimension)`
  - **`attentions:`** tuple of tensors, one for each layer. Length: # layers
    Shape of each: `(batch_size, num_heads, max_length, max_length)`
- [Can also be done with BertConfig object]

What the library does well

- Very easy tokenization
- Forward pass of models
  - Exposing as many internals as possible
    - All layers, attention heads, etc
- As unified an interface as possible
  - But: different models have different properties, controlled by Configs or by arguments
  - Read the docs carefully!
    - e.g. https://huggingface.co/docs/transformers/model_doc/bert#transformers.BertModel
    - The docs for `forward` just below explain the inputs/outputs

More Info

- Model search: https://huggingface.co/models
- Dataset search: https://huggingface.co/datasets
- Relatively new portions of the library (Trainer) may be useful for probing, but we'll look at another route for that now.

AllenNLP

https://allenai.org/allennlp/software/allennlp-library

Overview of AllenNLP

- Built on top of PyTorch
- Flexible data API
- Abstractions for common use cases in NLP
  - e.g. take a sequence of representations and give me a single one
- Modular:
  - Because of that, can swap in and out different options, for good experiments
- Declarative model-building / training via config files
- See https://github.com/allenai/writing-code-for-nlp-research-emnlp2018
- Guide: https://guide.allennlp.org/ <— very helpful for explaining some of the abstractions

Some Advantages

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:
- Training loop:

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:
- Training loop:

```
for each epoch:
  for each batch:
    get model outputs on batch
    compute loss
    compute gradients
    update parameters
```

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:
- Training loop:

```
for each epoch:
  for each batch:
    get model outputs on batch
    compute loss
    compute gradients
    update parameters
```

- Not *that* complicated, but:

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:
- Training loop:

```
for each epoch:
  for each batch:
    get model outputs on batch
    compute loss
    compute gradients
    update parameters
```

- Not *that* complicated, but:
  - Early stopping

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:
- Training loop:

```
for each epoch:
  for each batch:
    get model outputs on batch
    compute loss
    compute gradients
    update parameters
```

- Not *that* complicated, but:
  - Early stopping
  - Check-pointing (saving best model(s))

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:
- Training loop:

```
for each epoch:
  for each batch:
    get model outputs on batch
    compute loss
    compute gradients
    update parameters
```

- Not *that* complicated, but:
  - Early stopping
  - Check-pointing (saving best model(s))
  - Generating and padding the batches

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:
- Training loop:

```
for each epoch:
  for each batch:
    get model outputs on batch
    compute loss
    compute gradients
    update parameters
```

- Not *that* complicated, but:
  - Early stopping
  - Check-pointing (saving best model(s))
  - Generating and padding the batches
  - Logging results

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:
- Training loop:

```
for each epoch:
  for each batch:
    get model outputs on batch
    compute loss
    compute gradients
    update parameters
```

- Not *that* complicated, but:
  - Early stopping
  - Check-pointing (saving best model(s))
  - Generating and padding the batches
  - Logging results
  - ….

Some Advantages

- Focus on modeling / experimenting, not writing boilerplate, e.g.:
- Training loop:

```
for each epoch:
  for each batch:
    get model outputs on batch
    compute loss
    compute gradients
    update parameters
```

- Not *that* complicated, but:
  - Early stopping
  - Check-pointing (saving best model(s))
  - Generating and padding the batches
  - Logging results
  - ….

```
allennlp train myexperiment.jsonnet
```

Example Abstractions

- TextFieldEmbedder
- Seq2SeqEncoder
- Seq2VecEncoder
- Attention
- …

- Allows for easy swapping of different choices at every level in your model.

AllenNLP Bert Example

- See https://github.com/shanest/allennlp-bert-example [linked on course webpage as well]
- Using AllenNLP to probe BERT for two tasks:
  - Classification [Stanford Sentiment Treebank]
  - Tagging [Semantic Tagging]

Classifying

Overall Structure (Classification)

```
model_string = "bert-base-uncased"

tokenizer = PretrainedTransformerTokenizer(model_string)
token_indexer = PretrainedTransformerIndexer(model_string)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})
train_path = "sst/trees/train.txt"
dev_path = "sst/trees/dev.txt"

train_dataset = reader.read(train_path)
val_dataset = reader.read(dev_path)

print(list(train_dataset)[0])

vocab = Vocabulary.from_instances(chain(train_dataset, val_dataset))

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder({"tokens": bert_token_embedder})
cls_pooler = ClsPooler(bert_token_embedder.get_output_dim())

model = BertClassifier(
    vocab, bert_textfield_embedder, cls_pooler, freeze_encoder=False
)

data_loader = MultiProcessDataLoader(reader, train_path, batch_size=32)
data_loader.index_with(vocab)

trainer = GradientDescentTrainer(
    model=model,
    optimizer=optim.Adam(model.parameters()),
    serialization_dir="/tmp/test",
    data_loader=data_loader,
    train_dataset=train_dataset,
    validation_dataset=val_dataset,
    patience=5,
    num_epochs=30,
)
trainer.train()
```

Overall Structure (Classification)

```
model_string = "bert-base-uncased"

tokenizer = PretrainedTransformerTokenizer(model_string)
token_indexer = PretrainedTransformerIndexer(model_string)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})
train_path = "sst/trees/train.txt"
dev_path = "sst/trees/dev.txt"

train_dataset = reader.read(train_path)
val_dataset = reader.read(dev_path)

print(list(train_dataset)[0])

vocab = Vocabulary.from_instances(chain(train_dataset, val_dataset))

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder({"tokens": bert_token_embedder})
cls_pooler = ClsPooler(bert_token_embedder.get_output_dim())

model = BertClassifier(
    vocab, bert_textfield_embedder, cls_pooler, freeze_encoder=False
)

data_loader = MultiProcessDataLoader(reader, train_path, batch_size=32)
data_loader.index_with(vocab)

trainer = GradientDescentTrainer(
    model=model,
    optimizer=optim.Adam(model.parameters()),
    serialization_dir="/tmp/test",
    data_loader=data_loader,
    train_dataset=train_dataset,
    validation_dataset=val_dataset,
    patience=5,
    num_epochs=30,
)
trainer.train()
```

DatasetReader

Overall Structure (Classification)

```python
model_string = "bert-base-uncased"

tokenizer = PretrainedTransformerTokenizer(model_string)
token_indexer = PretrainedTransformerIndexer(model_string)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})
train_path = "sst/trees/train.txt"
dev_path = "sst/trees/dev.txt"

train_dataset = reader.read(train_path)
val_dataset = reader.read(dev_path)

print(list(train_dataset)[0])

vocab = Vocabulary.from_instances(chain(train_dataset, val_dataset))

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder({"tokens": bert_token_embedder})
cls_pooler = ClsPooler(bert_token_embedder.get_output_dim())

model = BertClassifier(
    vocab, bert_textfield_embedder, cls_pooler, freeze_encoder=False
)

data_loader = MultiProcessDataLoader(reader, train_path, batch_size=32)
data_loader.index_with(vocab)

trainer = GradientDescentTrainer(
    model=model,
    optimizer=optim.Adam(model.parameters()),
    serialization_dir="/tmp/test",
    data_loader=data_loader,
    train_dataset=train_dataset,
    validation_dataset=val_dataset,
    patience=5,
    num_epochs=30,
)
trainer.train()
```

← DatasetReader

← Model

Overall Structure (Classification)

```python
model_string = "bert-base-uncased"

tokenizer = PretrainedTransformerTokenizer(model_string)
token_indexer = PretrainedTransformerIndexer(model_string)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})
train_path = "sst/trees/train.txt"
dev_path = "sst/trees/dev.txt"

train_dataset = reader.read(train_path)
val_dataset = reader.read(dev_path)

print(list(train_dataset)[0])

vocab = Vocabulary.from_instances(chain(train_dataset, val_dataset))

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder({"tokens": bert_token_embedder})
cls_pooler = ClsPooler(bert_token_embedder.get_output_dim())

model = BertClassifier(
    vocab, bert_textfield_embedder, cls_pooler, freeze_encoder=False
)

data_loader = MultiProcessDataLoader(reader, train_path, batch_size=32)
data_loader.index_with(vocab)

trainer = GradientDescentTrainer(
    model=model,
    optimizer=optim.Adam(model.parameters()),
    serialization_dir="/tmp/test",
    data_loader=data_loader,
    train_dataset=train_dataset,
    validation_dataset=val_dataset,
    patience=5,
    num_epochs=30,
)
trainer.train()
```

DatasetReader

Model

Data Loader/Iterator

Overall Structure (Classification)

```
model_string = "bert-base-uncased"

tokenizer = PretrainedTransformerTokenizer(model_string)
token_indexer = PretrainedTransformerIndexer(model_string)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})
train_path = "sst/trees/train.txt"
dev_path = "sst/trees/dev.txt"

train_dataset = reader.read(train_path)
val_dataset = reader.read(dev_path)

print(list(train_dataset)[0])

vocab = Vocabulary.from_instances(chain(train_dataset, val_dataset))

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder({"tokens": bert_token_embedder})
cls_pooler = ClsPooler(bert_token_embedder.get_output_dim())

model = BertClassifier(
    vocab, bert_textfield_embedder, cls_pooler, freeze_encoder=False
)

data_loader = MultiProcessDataLoader(reader, train_path, batch_size=32)
data_loader.index_with(vocab)

trainer = GradientDescentTrainer(
    model=model,
    optimizer=optim.Adam(model.parameters()),
    serialization_dir="/tmp/test",
    data_loader=data_loader,
    train_dataset=train_dataset,
    validation_dataset=val_dataset,
    patience=5,
    num_epochs=30,
)
trainer.train()
```

DatasetReader

Model

Data Loader/Iterator

Trainer

Basic Components: Dataset Reader

- Datasets are collections of *Instances*, which are collections of *Fields*
  - For text classification, e.g.: one TextField, one LabelField
  - Many more: https://guide.allennlp.org/reading-data
- DatasetReaders….. read data sets. Two primary methods:
  - _read(file): reads data from disk, yields Instances. By calling:
  - text_to_instance (variable signature)
    - Processing of the “raw” data from disk into final form
    - Produces one Instance at a time

DatasetReader: Stanford Sentiment Treebank

- One line from train.txt:

(**3** (2 (2 The) (2 Rock)) (4 (3 (2 is) (4 (2 destined) (2 (2 (2 (2 (2 to) (2 (2 be) (2 (2 the) (2 (2 21st) (2 (2 (2 Century) (2 's)) (2 (3 new) (2 (2 ``) (2 Conan)))))))) (2 '')) (2 and)) (3 (2 that) (3 (2 he) (3 (2 's) (3 (2 going) (3 (2 to) (4 (3 (2 make) (3 (3 (2 a) (3 splash)) (2 (2 even) (3 greater)))) (2 (2 than) (2 (2 (2 (2 (1 (2 Arnold) (2 Schwarzenegger)) (2 ,)) (2 (2 Jean-Claud) (2 (2 Van) (2 Damme)))) (2 or)) (2 (2 Steven) (2 Segal))))))))))))) (2 .)))

- Core of _read:

```
parsed_line = Tree.fromstring(line)
instance = self.text_to_instance(parsed_line.leaves(), parsed_line.label())
if instance is not None:
    yield instance
```

- Core of text_to_instance:

```
if self._tokenizer:
    new_tokens = self._tokenizer.tokenize(' '.join(tokens))
else:
    new_tokens = [Token(token) for token in tokens]
text_field = TextField(new_tokens, token_indexers=self._token_indexers)
fields: Dict[str, Field] = {"tokens": text_field}
```

. . .

```
    fields["label"] = LabelField(sentiment)
return Instance(fields)
```

Model

```
@Model.register("bert_classifier")
class BertClassifier(Model):
    def __init__(
        self,
        vocab: Vocabulary,
        embedder: TextFieldEmbedder,
        pooler: Seq2VecEncoder,
        freeze_encoder: bool = True,
    ) -> None:
        super().__init__(vocab)

        self.vocab = vocab
        self.embedder = embedder
        self.pooler = pooler
        self.freeze_encoder = freeze_encoder

        for parameter in self.embedder.parameters():
            parameter.requires_grad = not self.freeze_encoder

        in_features = self.pooler.get_output_dim()
        out_features = vocab.get_vocab_size(namespace="labels")

        self._classification_layer = torch.nn.Linear(in_features, out_features)
        self._accuracy = CategoricalAccuracy()
        self._loss = torch.nn.CrossEntropyLoss()
```

Model

```
@Model.register("bert_classifier")
class BertClassifier(Model):
    def __init__(
        self,
        vocab: Vocabulary,
        embedder: TextFieldEmbedder,
        pooler: Seq2VecEncoder,
        freeze_encoder: bool = True,
    ) -> None:
        super().__init__(vocab)

        self.vocab = vocab
        self.embedder = embedder
        self.pooler = pooler
        self.freeze_encoder = freeze_encoder

        for parameter in self.embedder.parameters():
            parameter.requires_grad = not self.freeze_encoder

        in_features = self.pooler.get_output_dim()
        out_features = vocab.get_vocab_size(namespace="labels")

        self._classification_layer = torch.nn.Linear(in_features, out_features)
        self._accuracy = CategoricalAccuracy()
        self._loss = torch.nn.CrossEntropyLoss()
```

Fine tune or not

Model

```python
def forward(  # type: ignore
    self, tokens: Dict[str, torch.Tensor], label: torch.IntTensor = None
) -> Dict[str, torch.Tensor]:

    # (batch_size, max_len, embedding_dim)
    embeddings = self.embedder(tokens)

    # get the pooled representation of the tokens in each sentence
    # e.g. [CLS] rep, mean pool, ...
    # (batch_size, embedding_dim)
    sentence_representation = self.pooler(embeddings)

    # apply classification layer
    # (batch_size, num_labels)
    logits = self._classification_layer(sentence_representation)

    probs = torch.nn.functional.softmax(logits, dim=-1)

    output_dict = {"logits": logits, "probs": probs}

    if label is not None:
        loss = self._loss(logits, label.long().view(-1))
        output_dict["loss"] = loss
        self._accuracy(logits, label)

    return output_dict
```

NB: frozen embeddings can be pre-computed for efficiency

Where was BERT?

- In the TextFieldEmbedder!
- In run_classifying.py: initialized a PretrainedTransformerEmbedder
  - AllenNLP has wrappers around HuggingFace
  - But note: to extract more from a model, you'll probably need to write your own class, using the existing ones as inspiration

Config file (classifying_experiment.jsonnet)

```
local bert_model = "bert-base-uncased";
{
    "dataset_reader": {
        "type": "sst_reader",
        "tokenizer": {
            "type": "pretrained_transformer",
            "model_name": bert_model,
        },
        "token_indexers": {
            "tokens": {
                "type": "pretrained_transformer",
                "model_name": bert_model,
            }
        }
    },
    "train_data_path": "sst/trees/train.txt",
    "validation_data_path": "sst/trees/dev.txt",
```

`@DatasetReader.register("sst_reader")`

Arguments to SSTReader!

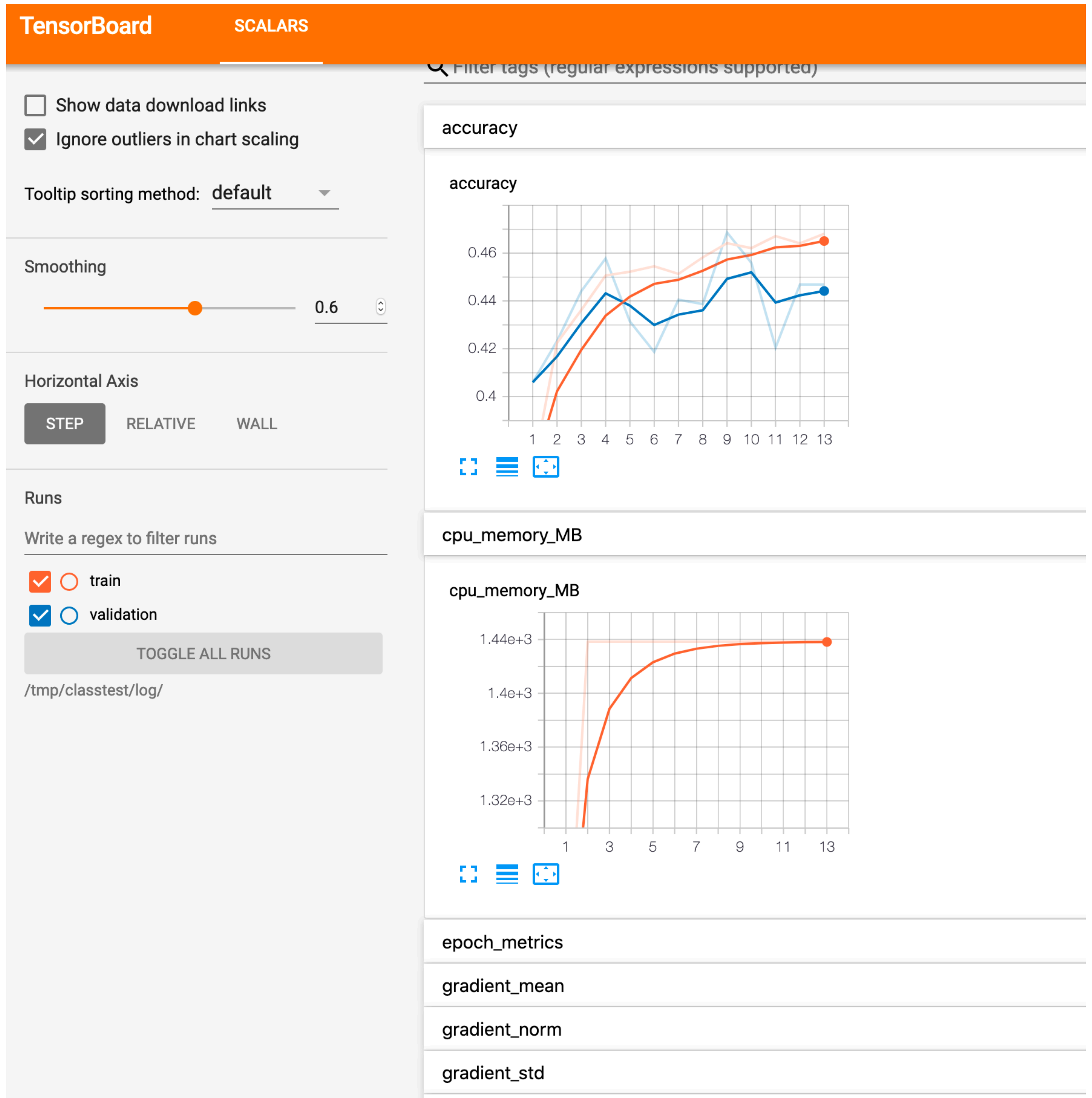
Config file (classifying_experiment.jsonnet)

```
    "model": {
        "type": "bert_classifier",
        "embedder": {
            "type": "basic",
            "token_embedders": {
                "tokens": {
                    "type": "pretrained_transformer",
                    "model_name": bert_model
                }
            }
        },
        "pooler": {
            # NB: for probing, cls_pooler and boe_pooler are good choices
            # bert_pooler actually does more than what is wanted in that scenario
            "type": "cls_pooler",
            "embedding_dim": 768,
        },
        "freeze_encoder": true,
    },
    "data_loader": {
        "batch_size": 32
    },
    "trainer": {
        "optimizer": {
            "type": "adam",
            "lr": 0.001
        },
        "validation_metric": "+accuracy",
        "checkpointer": {
            "keep_most_recent_by_count": 1
        },
        "num_epochs": 30,
        "grad_norm": 10.0,
        "patience": 5,
        "cuda_device": -1
    }
}
```

```
allennlp train classifying_experiment.jsonnet \
  --serialization-dir test \
  --include-package classifying
```

TensorBoard

```
tensorboard --logdir /serialization_dir/log
```



Use SSH port forwarding to [view server-side results locally]

Tagging

Tagging

- The repository also has an example of training a semantic tagger
  - Like POS tagging, but with a richer set of “semantic” tags
- Issue: the data comes with its own tokenization:
  - BERT: ['the', 'ya', '##zuka', 'are', 'the', 'japanese', 'mafia', ‘.’]
- Need to get word-level representations out of BERT’s *subword* representations

ANA	PRO	pronoun
	DEF	definite
	HAS	possessive
	REF	reflexive
	EMP	emphasizing
ACT	GRE	greeting
	ITJ	interjection
	HES	hesitation
	QUE	interrogative
ATT	QUA	quantity
	UOM	measurement
	IST	intersective
	REL	relation
	RLI	rel. inv. scope
	SST	subsective
	PRI	privative
	INT	intensifier
	SCO	score
LOG	ALT	alternative
	EXC	exclusive
	NIL	empty
	DIS	disjunct./exist.
	IMP	implication
	AND	conjunct./univ.
	BUT	contrast
COM	EQA	equative
	MOR	comparative pos.
	LES	comparative neg.
	TOP	pos. superlative
	BOT	neg. superlative
	ORD	ordinal
DEM	PRX	proximal
	MED	medial
	DST	distal
DIS	SUB	subordinate
	COO	coordinate
	APP	appositional
MOD	NOT	negation
	NEC	necessity
	POS	possibility
ENT	CON	concept
	ROL	role
NAM	GPE	geo-political ent.
	PER	person
	LOC	location
	ORG	organisation
	ART	artifact
	NAT	natural obj./phen.
	HAP	happening
	URL	url
EVE	EXS	untensed simple
	ENS	present simple
	EPS	past simple
	EFS	future simple
	EXG	untensed prog.
	ENG	present prog.
	EPG	past prog.
	EFG	future prog.
	EXT	untensed perfect
	ENT	present perfect
	EPT	past perfect
	EFT	future perfect
	ETG	perfect prog.
	ETV	perfect passive
	EXV	passive
TNS	NOW	present tense
	PST	past tense
	FUT	future tense
TIM	DOM	day of month
	YOC	year of century
	DOW	day of week
	MOY	month of year
	DEC	decade
	CLO	clocktime

```
DEF     The
CON     yakuza
ENS     are
DEF     the
GPO     Japanese
CON     mafia
NIL     .
~
```

Tagging: Modeling

- Used to be complicated, BUT:
- They’ve added a PretrainedMismatchedTransformerEmbedder (and a corresponding PretrainedMismatchedTransformerIndexer for tokens)
  - Handles all of the mis-alignment between dataset tokens and model tokens for you!
  - How to pool subwords—>words:
    - `sub_token_mode` kwarg: default = avg, but can do first/last, etc

Tagging: Modeling

```python
@Model.register("subword_word_tagger")
class SubwordWordTagger(Model):

    # TODO: document!

    def __init__(
        self,
        embedder: TextFieldEmbedder,
        vocab: Vocabulary = None,
        freeze_encoder: bool = True,
    ):
        super().__init__(vocab)

        self._embedder = embedder
        self._freeze_encoder = freeze_encoder
        # turn off gradients if don't want to fine tune encoder
        for parameter in self._embedder.parameters():
            parameter.requires_grad = not self._freeze_encoder

        self.classifier = TimeDistributed(
            torch.nn.Linear(
                in_features=embedder.get_output_dim(),
                out_features=vocab.get_vocab_size("labels"),
            )
        )

        self.accuracy = CategoricalAccuracy()
```

```python
def forward(
    self, sentence: Dict[str, torch.Tensor], labels: torch.Tensor = None
) -> Dict[str, torch.Tensor]:

    # (batch_size, max_seq_len, embedding_dim)
    embeddings = self._embedder(sentence)

    # get mask to keep track of which tokens are padding
    # prevent them from contributing to loss
    # (batch_size, max_word_seq_len)
    word_mask = get_text_field_mask(sentence)

    # (batch_size, max_word_seq_len, num_labels)
    logits = self.classifier(embeddings)

    outputs = {"logits": logits}

    if labels is not None:
        self.accuracy(logits, labels, word_mask)
        outputs["loss"] = sequence_cross_entropy_with_logits(
            logits, labels, word_mask
        )

    return outputs
```

On These Libraries

- If you're using transformer-based LMs, I strongly recommend HuggingFace
- On the other hand, it's possible that learning AllenNLP's abstractions may cost you more time than it saves in the short term
- As always, try and use the best tool for the job at hand
- One more that makes fine-tuning and/or diagnostic classification easy:
  - jiant

Other tools for experiment management

- Disclaimer: I've never used them!
  - Might be over-kill in the short term
- Guild (entirely local): https://guild.ai/
- CodaLab: https://codalab.org/
- Weights and Biases: https://www.wandb.com/
- Neptune: https://neptune.ai/

Using GPUs on Patas

Setting up local environment

- Three GPU nodes:
  - 2xTesla P40
  - 8xTesla M10
  - 2xQuadro 8000
- For info on setting up your local environment to use these nodes in a fairly painless way:
  - https://www.shane.st/teaching/575/spr22/patas-gpu.pdf

Condor job file for patas

```
executable = run_exp_gpu.sh
getenv = True
error = exp.error
log = exp.log
notification = always
transfer_executable = false
request_memory = 8*1024
request_GPUs = 1
+Research = True
Queue
```

Example executable

```
#!/bin/sh
conda activate my-project

allennlp train tagging_experiment.jsonnet --serialization-dir test \
  --include-package tagging \
  --overrides "{'trainer': {'cuda_device': 1}}"
```