

Libraries and Tools

Transformers, AllenNLP

LING575 Analyzing Neural Language Models

Shane Steinert-Threlkeld

Apr 23 2021

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
- Interoperable with PyTorch or TensorFlow 2.0

Example: Tokenization

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

# OPTIONAL: if you want to have more information on what's happening under the hood, activate the logger as follows
import logging
logging.basicConfig(level=logging.INFO)

# Load pre-trained model tokenizer (vocabulary)
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')

# Tokenize input
text = "[CLS] Who was Jim Henson ? [SEP] Jim Henson was a puppeteer [SEP]"
tokenized_text = tokenizer.tokenize(text)

# Mask a token that we will try to predict back with `BertForMaskedLM`
masked_index = 8
tokenized_text[masked_index] = '[MASK]'
assert tokenized_text == ['[CLS]', 'who', 'was', 'jim', 'henson', '?', '[SEP]', 'jim', '[MASK]', 'was', 'a', 'puppet', '##eer', '[SEP]']

# Convert token to vocabulary indices
indexed_tokens = tokenizer.convert_tokens_to_ids(tokenized_text)
# Define sentence A and B indices associated to 1st and 2nd sentences (see paper)
segments_ids = [0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1]

# Convert inputs to PyTorch tensors
tokens_tensor = torch.tensor([indexed_tokens])
segments_tensors = torch.tensor([segments_ids])
```

Example: Forward Pass

```
# Load pre-trained model (weights)
model = BertModel.from_pretrained('bert-base-uncased')

# Set the model in evaluation mode to deactivate the DropOut modules
# This is IMPORTANT to have reproducible results during evaluation!
model.eval()

# If you have a GPU, put everything on cuda
tokens_tensor = tokens_tensor.to('cuda')
segments_tensors = segments_tensors.to('cuda')
model.to('cuda')

# Predict hidden states features for each layer
with torch.no_grad():
    # See the models docstrings for the detail of the inputs
    outputs = model(tokens_tensor, token_type_ids=segments_tensors)
    # Transformers models always output tuples.
    # See the models docstrings for the detail of all the outputs
    # In our case, the first element is the hidden state of the last layer of the Bert model
    encoded_layers = outputs[0]
```


Outputs from the forward pass

- Outputs are always *tuples of Tensors*
- BERT, by default, gives two things:
 - Top layer embeddings for each token.
Shape: (batch_size, max_length, embedding_dimension)
 - Pooled representation: embedding of '[CLS]' token, passed through one tanh layer
Shape: (batch_size, embedding_dimension)

Getting more out of a model

```
from transformers import BertConfig, BertModel

config = BertConfig(
    "bert-base-uncased", output_attentions=True, output_hidden_states=True)

model = BertModel(config)
```

Getting more out of a model

```
from transformers import BertConfig, BertModel

config = BertConfig(
    "bert-base-uncased", output_attentions=True, output_hidden_states=True)

model = BertModel(config)
```

- Now, it's a 4-tuple as output, additionally containing:
 - Hidden states. A tuple of tensors, one for each layer. Length: # layers
Shape of each: (batch_size, max_length, embedding_dimension)
 - Attention heads: tuple of tensors, one for each layer. Length: # layers
Shape of each: (batch_size, num_heads, max_length, max_length)

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
 - Read the docs carefully!

What the library does not do

- Anything related to training
 - Padding
 - Batching
 - Optimizing probe models, etc. Use PyTorch (or TF) for that
- Model search: <https://huggingface.co/models>
- Dataset search: <https://huggingface.co/datasets>

AllenNLP

<https://allennlp.org/>

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.

Overall Structure (Classification)

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

tokenizer = PretrainedTransformerTokenizer(
    model_string, do_lowercase=True)
token_indexer = PretrainedTransformerIndexer(
    model_string, do_lowercase=True)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})

train_dataset = reader.read('sst/trees/train.txt')
val_dataset = reader.read('sst/trees/dev.txt')

print(train_dataset[0])

vocab = Vocabulary.from_instances(train_dataset + val_dataset)

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder(
    {"tokens": bert_token_embedder})

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

iterator = BucketIterator(
    sorting_keys=[("tokens", "num_tokens")],
    batch_size=32)
iterator.index_with(vocab)

trainer = Trainer(model=model,
    optimizer=optim.Adam(model.parameters()),
    serialization_dir='/tmp/test',
    iterator=iterator,
    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, do_lowercase=True)
token_indexer = PretrainedTransformerIndexer(
    model_string, do_lowercase=True)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})

train_dataset = reader.read('sst/trees/train.txt')
val_dataset = reader.read('sst/trees/dev.txt')

print(train_dataset[0])

vocab = Vocabulary.from_instances(train_dataset + val_dataset)

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder(
    {"tokens": bert_token_embedder})

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

iterator = BucketIterator(
    sorting_keys=[("tokens", "num_tokens")],
    batch_size=32)
iterator.index_with(vocab)

trainer = Trainer(model=model,
    optimizer=optim.Adam(model.parameters()),
    serialization_dir='/tmp/test',
    iterator=iterator,
    train_dataset=train_dataset,
    validation_dataset=val_dataset,
    patience=5,
    num_epochs=30)

trainer.train()
```

← DatasetReader

Overall Structure (Classification)

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

tokenizer = PretrainedTransformerTokenizer(
    model_string, do_lowercase=True)
token_indexer = PretrainedTransformerIndexer(
    model_string, do_lowercase=True)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})

train_dataset = reader.read('sst/trees/train.txt')
val_dataset = reader.read('sst/trees/dev.txt')

print(train_dataset[0])

vocab = Vocabulary.from_instances(train_dataset + val_dataset)

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder(
    {"tokens": bert_token_embedder})

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

iterator = BucketIterator(
    sorting_keys=[("tokens", "num_tokens")],
    batch_size=32)
iterator.index_with(vocab)

trainer = Trainer(model=model,
    optimizer=optim.Adam(model.parameters()),
    serialization_dir='/tmp/test',
    iterator=iterator,
    train_dataset=train_dataset,
    validation_dataset=val_dataset,
    patience=5,
    num_epochs=30)

trainer.train()
```

← DatasetReader

← Model

Overall Structure (Classification)

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

tokenizer = PretrainedTransformerTokenizer(
    model_string, do_lowercase=True)
token_indexer = PretrainedTransformerIndexer(
    model_string, do_lowercase=True)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})

train_dataset = reader.read('sst/trees/train.txt')
val_dataset = reader.read('sst/trees/dev.txt')

print(train_dataset[0])

vocab = Vocabulary.from_instances(train_dataset + val_dataset)

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder(
    {"tokens": bert_token_embedder})

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

iterator = BucketIterator(
    sorting_keys=[("tokens", "num_tokens")],
    batch_size=32)
iterator.index_with(vocab)

trainer = Trainer(model=model,
                  optimizer=optim.Adam(model.parameters()),
                  serialization_dir='/tmp/test',
                  iterator=iterator,
                  train_dataset=train_dataset,
                  validation_dataset=val_dataset,
                  patience=5,
                  num_epochs=30)

trainer.train()
```

← DatasetReader

← Model

← Iterator

Overall Structure (Classification)

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

tokenizer = PretrainedTransformerTokenizer(
    model_string, do_lowercase=True)
token_indexer = PretrainedTransformerIndexer(
    model_string, do_lowercase=True)

reader = SSTDatasetReader(tokenizer, {"tokens": token_indexer})

train_dataset = reader.read('sst/trees/train.txt')
val_dataset = reader.read('sst/trees/dev.txt')

print(train_dataset[0])

vocab = Vocabulary.from_instances(train_dataset + val_dataset)

bert_token_embedder = PretrainedTransformerEmbedder(model_string)
bert_textfield_embedder = BasicTextFieldEmbedder(
    {"tokens": bert_token_embedder})

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

iterator = BucketIterator(
    sorting_keys=[("tokens", "num_tokens")],
    batch_size=32)
iterator.index_with(vocab)

trainer = Trainer(model=model,
    optimizer=optim.Adam(model.parameters()),
    serialization_dir='/tmp/test',
    iterator=iterator,
    train_dataset=train_dataset,
    validation_dataset=val_dataset,
    patience=5,
    num_epochs=30)

trainer.train()
```

DatasetReader

Model

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 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 (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,
        freeze_encoder: bool = True
    ) -> None:
        super().__init__(vocab)

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

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

        in_features = self.embedder.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,
        freeze_encoder: bool = True
    ) -> None:
        super().__init__(vocab)

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

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

        in_features = self.embedder.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

```
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)

    # the first embedding is for the [CLS] token
    # NOTE: this pre-supposes BERT encodings; not the most elegant!
    # (batch_size, embedding_dim)
    cls_embedding = embeddings[:, 0, :]

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

    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 `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";
```

```
local do_lowercase = true;
```

```
{
```

```
  "dataset_reader": {
```

```
    "type": "sst_reader",
```

```
    "tokenizer": {
```

```
      "type": "pretrained_transformer",
```

```
      "model_name": bert_model,
```

```
      "do_lowercase": do_lowercase
```

```
    },
```

```
    "token_indexers": {
```

```
      "tokens": {
```

```
        "type": "pretrained_transformer",
```

```
        "model_name": bert_model,
```

```
        "do_lowercase": do_lowercase
```

```
      }
```

```
    }
```

```
  },
```

```
  "train_data_path": "sst/trees/train.txt",
```

```
  "validation_data_path": "sst/trees/dev.txt",
```

`@DatasetReader.register("sst_reader")`



Arguments to SSTReader!



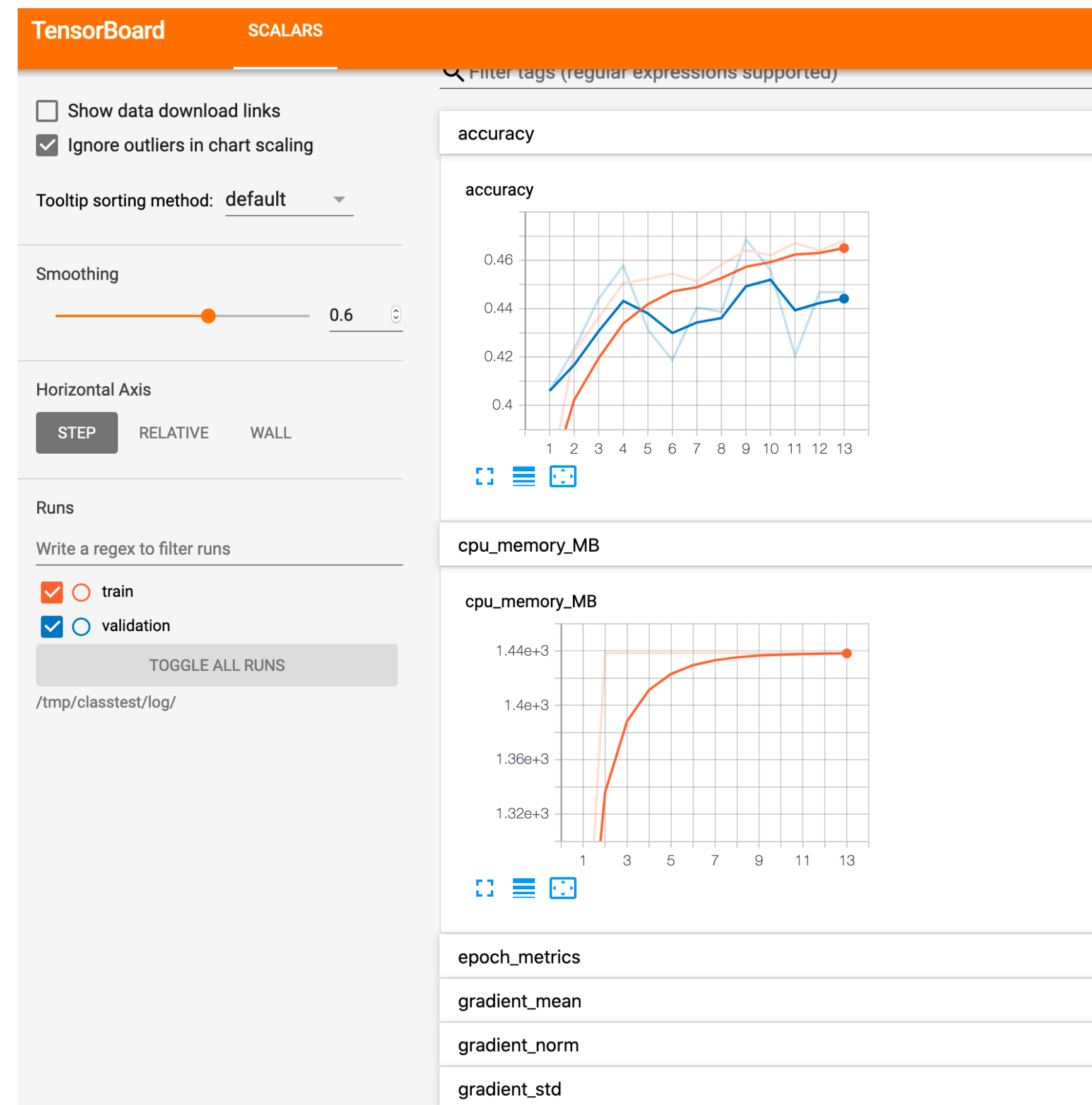
Config file (classifying_experiment.jsonnet)

```
{
  "model": {
    "type": "bert_classifier",
    "embedder": {
      "type": "basic",
      "tokens": {
        "type": "pretrained_transformer",
        "model_name": bert_model
      }
    },
    "freeze_encoder": true,
  },
  "iterator": {
    "type": "bucket",
    "sorting_keys": [["tokens", "num_tokens"]],
    "batch_size": 32
  },
  "trainer": {
    "optimizer": {
      "type": "adam",
      "lr": 0.001
    },
    "validation_metric": "+accuracy",
    "checkpointer": {
      "num_serialized_models_to_keep": 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

- 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

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

Tagging: Modeling

- My example: keep track of which spans of BERT tokens the original words correspond to
 - Some complication in the DatasetReader because of this
- And then combine those representations with an arbitrary Seq2VecEncoder
- Since then, they've added a PretrainedMismatchedTransformerEmbedder that has essentially the same functionality
 - (Spans are pooled by summing, not by an arbitrary Seq2Vec)
 - Might be safest to use that (and corresponding MismatchedIndexer)

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/spr21/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}}'
```