Writing a Transformer Classifier in PyTorch

Tags: neuralnetworks • machinelearning

Aug 24, 2021 • n8henrie

Bottom Line: I made a transformer-encoder-based classifier in PyTorch.

About a year ago, I was learning a bit about the transformer-based neural networks that have become the new state-of-the-art for natural language processing, like BERT. There are some excellent libraries by the likes of HuggingFace that make it extremely easy to get up and running with these architectures, but I was hoping to gain some experience using PyTorch directly.

My initial several attempts didn't seem to learn much, so I reached out for help on the PyTorch forum as well as r/learnmachinelearning, and also reached out to the authors of two really helpful example posts (which also include example code):

- http://peterbloem.nl/blog/transformers
- https://buomsoo-kim.github.io/attention/2020/04/22/Attention-mechanism-20.md/

I eventually got a simple network that could learn reasonably well; as shown below, it gets a *train*-set accuracy of up to 97% with test-set around 81%. The fact that it can over-fit proves that it can learn! The biggest sticking points I ran into were:

- trying to understand the role and the dimensions of the position encoder
- noting the changes in the label after tokenization with vs without padding and unknown tokens
- finding a workable learning rate

This post doesn't explain everything at length, in part because that has already done by others with much better understanding than I have. In addition to the posts above, some of the most heplful links and discussion that I ran across include:

- https://pytorch.org/tutorials/beginner/transformer_tutorial.html
  - EDIT: The tutorial above has changed since I wrote this code; here's a wayback machine link to the version I was using at the time
- https://github.com/jensjepsen/imdb-transformer
- https://github.com/pbloem/former

I initially wrote this code in a Jupyter notebook, so you'll see a few helper functions that I like to use to do things like automatically format cells with black. You'll probably also notice that I also silence some warnings – there are some deprecation warnings with the particular versions of `torchtext` that I use; if you use a more recent version you may need to modify the code to account for these deprecations, but as of the time of writing this post it works with the versions of `torch` and `torchtext` listed below.

At the very top of my notebooks, I like to define a function that can be used to determine if code is running a notebook or not; this makes it so that I can put notebook-specific logic in a conditional, and if I export the notebook to a python script (`.ipynb` -> `.py`) I can run it with `ipython myscript.py` without the notebook-specific cells messing things up.

As you can see below, for this code I was using `torchtext==0.8.1` and `torch==1.7.1`, which you should install prior to running any of the code below.

```
def running_in_notebook() -> bool:
    """
    https://stackoverflow.com/a/39662359\
    It returns 'TerminalInteractiveShell' on a terminal IPython,
    'ZMQInteractiveShell' on Jupyter (notebook AND qtconsole) and fails
    (NameError) on a regular Python interpreter. The method get_python() seems
    to be available in the global namespace
    """
    try:
        return get_ipython().__class__.__name__ == "ZMQInteractiveShell"
    except NameError:
        return False
```

Next, I install my fork of `nb_black`, which adds a hook to jupyter that automatically formats cells with `black`. My fork lets me set my preferred line length (79) instead of using the `black` default value.

```
# pip install git+https://github.com/n8henrie/nb_black
import lab_black

if running_in_notebook():
    lab_black.load_ipython_extension(get_ipython(), line_length=79)
```

Essential imports and preparing to use `cuda` if available.

```
import math
```

```
import torch
import torch.nn as nn

import torchtext

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

Show the versions I am using for this example:

```
torch.__version__, torchtext.__version__
```

```
('1.7.1', '0.8.1')
```

Silence the warnings as noted above.

```
import warnings

# Ignore some torchtext warnings due to originally writing this code with an
# older version of torchtext
warnings.filterwarnings("ignore", category=UserWarning)
```

Set up my dataset; for this I was using the IMDB sentiment classification dataset, which is a popular one for NLP classification tasks. It is also especially convenient to use with `torchtext`.

```
batch_size = 30
max_length = 256

TEXT = torchtext.data.Field(
    lower=True, include_lengths=False, batch_first=True
```

```
)
LABEL = torchtext.data.Field(sequential=False)
train_txt, test_txt = torchtext.datasets.IMDB.splits(TEXT, LABEL)

TEXT.build_vocab(
    train_txt,
    vectors=torchtext.vocab.GloVe(name="6B", dim=50, max_vectors=50_000),
    max_size=50_000,
)

LABEL.build_vocab(train_txt)

train_iter, test_iter = torchtext.data.BucketIterator.splits(
    (train_txt, test_txt),
    batch_size=batch_size,
)
```

Use the PositionalEncoding module from the official PyTorch tutorial.

```
class PositionalEncoding(nn.Module):
    """
    https://pytorch.org/tutorials/beginner/transformer_tutorial.html
    """

    def __init__(self, d_model, vocab_size=5000, dropout=0.1):
        super().__init__()
        self.dropout = nn.Dropout(p=dropout)

        pe = torch.zeros(vocab_size, d_model)
        position = torch.arange(0, vocab_size, dtype=torch.float).unsqueeze(1)
        div_term = torch.exp(
```

```
            torch.arange(0, d_model, 2).float()
            * (-math.log(10000.0) / d_model)
        )
        pe[:, 0::2] = torch.sin(position * div_term)
        pe[:, 1::2] = torch.cos(position * div_term)
        pe = pe.unsqueeze(0)
        self.register_buffer("pe", pe)

    def forward(self, x):
        x = x + self.pe[:, : x.size(1), :]
        return self.dropout(x)
```

Set up my example neural network, with `nn.TransformerEncoder` at its core.

```
class Net(nn.Module):
    """
    Text classifier based on a pytorch TransformerEncoder.
    """

    def __init__(
        self,
        embeddings,
        nhead=8,
        dim_feedforward=2048,
        num_layers=6,
        dropout=0.1,
        activation="relu",
        classifier_dropout=0.1,
    ):

        super().__init__()
```

```
        vocab_size, d_model = embeddings.size()
        assert d_model % nhead == 0, "nheads must divide evenly into d_model"

        self.emb = nn.Embedding.from_pretrained(embeddings, freeze=False)

        self.pos_encoder = PositionalEncoding(
            d_model=d_model,
            dropout=dropout,
            vocab_size=vocab_size,
        )

        encoder_layer = nn.TransformerEncoderLayer(
            d_model=d_model,
            nhead=nhead,
            dim_feedforward=dim_feedforward,
            dropout=dropout,
        )
        self.transformer_encoder = nn.TransformerEncoder(
            encoder_layer,
            num_layers=num_layers,
        )
        self.classifier = nn.Linear(d_model, 2)
        self.d_model = d_model

    def forward(self, x):
        x = self.emb(x) * math.sqrt(self.d_model)
        x = self.pos_encoder(x)
        x = self.transformer_encoder(x)
        x = x.mean(dim=1)
        x = self.classifier(x)
```

```
        return x
```

Now we're going to set up our training loop. Note the learning rate – at `1e-3`, it wouldn't learn *anything*, which stumped me for a while. Also note `labels = batch.label.to(device) - 1` in a couple places; this was a big "gotcha" as well. This accounts for a difference in the labels, where the `LABEL.vocab` includes `<unk>` as index 0, but there are no `unknown` labels in the dataset, so comparing labels and predictions ends up being off by one:

```
>>> print(LABEL.vocab.itos)
['<unk>', 'neg', 'pos']
>>> set(row.label for row in iter(train_txt))
{'neg', 'pos'}
```

I am sure there is a better way to do this, but for this simple example just manually accounting for the offset seemed to work ¯\\(ツ)/¯.

```
epochs = 50
model = Net(
    TEXT.vocab.vectors,
    nhead=5,  # the number of heads in the multiheadattention models
    dim_feedforward=50,  # the dimension of the feedforward network model in nn.T
    num_layers=6,
    dropout=0.0,
    classifier_dropout=0.0,
).to(device)

criterion = nn.CrossEntropyLoss()
```

```
lr = 1e-4
optimizer = torch.optim.Adam(
    (p for p in model.parameters() if p.requires_grad), lr=lr
)

torch.manual_seed(0)

print("starting")
for epoch in range(epochs):
    print(f"{epoch=}")
    epoch_loss = 0
    epoch_correct = 0
    epoch_count = 0
    for idx, batch in enumerate(iter(train_iter)):
        predictions = model(batch.text.to(device))
        labels = batch.label.to(device) - 1

        loss = criterion(predictions, labels)

        correct = predictions.argmax(axis=1) == labels
        acc = correct.sum().item() / correct.size(0)

        epoch_correct += correct.sum().item()
        epoch_count += correct.size(0)

        epoch_loss += loss.item()

        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), 0.5)

        optimizer.step()
```

```
    with torch.no_grad():
        test_epoch_loss = 0
        test_epoch_correct = 0
        test_epoch_count = 0

        for idx, batch in enumerate(iter(test_iter)):
            predictions = model(batch.text.to(device))
            labels = batch.label.to(device) - 1
            test_loss = criterion(predictions, labels)

            correct = predictions.argmax(axis=1) == labels
            acc = correct.sum().item() / correct.size(0)

            test_epoch_correct += correct.sum().item()
            test_epoch_count += correct.size(0)
            test_epoch_loss += loss.item()

    print(f"{epoch_loss=}")
    print(f"epoch accuracy: {epoch_correct / epoch_count}")
    print(f"{test_epoch_loss=}")
    print(f"test epoch accuracy: {test_epoch_correct / test_epoch_count}")
```

And finally, our results:

```
starting
epoch=0
epoch_loss=559.5332527160645
epoch accuracy: 0.57612
test_epoch_loss=546.3963396549225
```

```
test epoch accuracy: 0.65192
epoch=1
epoch_loss=518.0106997191906
epoch accuracy: 0.66732
test_epoch_loss=512.6058307886124
test epoch accuracy: 0.69032
epoch=2
epoch_loss=498.6085506975651
epoch accuracy: 0.6928
test_epoch_loss=479.70046305656433
test epoch accuracy: 0.6886
epoch=3
epoch_loss=482.0771609544754
epoch accuracy: 0.7056
test_epoch_loss=536.8250241279602
test epoch accuracy: 0.68036
epoch=4
epoch_loss=463.482199460268
epoch accuracy: 0.73532
test_epoch_loss=399.51818919181824
test epoch accuracy: 0.63804
epoch=5
epoch_loss=446.4596481323242
epoch accuracy: 0.75124
test_epoch_loss=486.4531066417694
test epoch accuracy: 0.675
epoch=6
epoch_loss=427.6187916994095
epoch accuracy: 0.76744
test_epoch_loss=542.3589209318161
test epoch accuracy: 0.65684
```

```
epoch=7
epoch_loss=414.25179597735405
epoch accuracy: 0.77524
test_epoch_loss=460.0263297557831
test epoch accuracy: 0.69612
epoch=8
epoch_loss=394.7843403071165
epoch accuracy: 0.79376
test_epoch_loss=369.7193158864975
test epoch accuracy: 0.71712
epoch=9
epoch_loss=383.44597190618515
epoch accuracy: 0.7986
test_epoch_loss=464.3820433616638
test epoch accuracy: 0.69912
epoch=10
epoch_loss=370.2925351560116
epoch accuracy: 0.80572
test_epoch_loss=387.68890875577927
test epoch accuracy: 0.72644
epoch=11
epoch_loss=358.9472469240427
epoch accuracy: 0.81468
test_epoch_loss=358.85351997613907
test epoch accuracy: 0.71104
epoch=12
epoch_loss=345.9787204861641
epoch accuracy: 0.82468
test_epoch_loss=426.30874586105347
test epoch accuracy: 0.73864
epoch=13
```

```
epoch_loss=333.1944961845875
epoch accuracy: 0.82964
test_epoch_loss=313.9570151567459
test epoch accuracy: 0.73576
epoch=14
epoch_loss=323.7126570418477
epoch accuracy: 0.84012
test_epoch_loss=326.1853696703911
test epoch accuracy: 0.72928
epoch=15
epoch_loss=309.87681122124195
epoch accuracy: 0.84868
test_epoch_loss=326.0149628520012
test epoch accuracy: 0.75076
epoch=16
epoch_loss=301.1800754368305
epoch accuracy: 0.8556
test_epoch_loss=309.1706365942955
test epoch accuracy: 0.7626
epoch=17
epoch_loss=289.21689750254154
epoch accuracy: 0.86324
test_epoch_loss=193.38222339749336
test epoch accuracy: 0.76488
epoch=18
epoch_loss=282.7087580934167
epoch accuracy: 0.86676
test_epoch_loss=264.42927753925323
test epoch accuracy: 0.76844
epoch=19
epoch_loss=272.87283693253994
```

```
epoch accuracy: 0.87008
test_epoch_loss=202.70429161190987
test epoch accuracy: 0.74892
epoch=20
epoch_loss=263.15157068520784
epoch accuracy: 0.87596
test_epoch_loss=190.99209129810333
test epoch accuracy: 0.7798
epoch=21
epoch_loss=252.78857025504112
epoch accuracy: 0.88224
test_epoch_loss=178.93173265457153
test epoch accuracy: 0.74548
epoch=22
epoch_loss=243.34817136079073
epoch accuracy: 0.88736
test_epoch_loss=303.81853026151657
test epoch accuracy: 0.7712
epoch=23
epoch_loss=235.68663988262415
epoch accuracy: 0.89184
test_epoch_loss=166.50022467970848
test epoch accuracy: 0.77032
epoch=24
epoch_loss=225.7369863986969
epoch accuracy: 0.89628
test_epoch_loss=178.69956082105637
test epoch accuracy: 0.78832
epoch=25
epoch_loss=214.11821631900966
epoch accuracy: 0.90536
```

```
test_epoch_loss=185.3488194644451
test epoch accuracy: 0.78296
epoch=26
epoch_loss=205.7373289577663
epoch accuracy: 0.91028
test_epoch_loss=253.90901774168015
test epoch accuracy: 0.79712
epoch=27
epoch_loss=196.76159628480673
epoch accuracy: 0.9154
test_epoch_loss=178.60953551530838
test epoch accuracy: 0.79608
epoch=28
epoch_loss=189.15391789004207
epoch accuracy: 0.91832
test_epoch_loss=162.34352615475655
test epoch accuracy: 0.77512
epoch=29
epoch_loss=180.43747867643833
epoch accuracy: 0.922
test_epoch_loss=228.25893902778625
test epoch accuracy: 0.79712
epoch=30
epoch_loss=172.36834182962775
epoch accuracy: 0.92848
test_epoch_loss=178.21085911989212
test epoch accuracy: 0.80576
epoch=31
epoch_loss=164.76151644438505
epoch accuracy: 0.93072
test_epoch_loss=134.71386006474495
```

```
test epoch accuracy: 0.78644
epoch=32
epoch_loss=157.83770682290196
epoch accuracy: 0.93368
test_epoch_loss=160.06551557779312
test epoch accuracy: 0.80004
epoch=33
epoch_loss=154.58212885446846
epoch accuracy: 0.93652
test_epoch_loss=156.32615965604782
test epoch accuracy: 0.786
epoch=34
epoch_loss=143.92170652374625
epoch accuracy: 0.94096
test_epoch_loss=128.87077778577805
test epoch accuracy: 0.79132
epoch=35
epoch_loss=136.34926942829043
epoch accuracy: 0.94276
test_epoch_loss=115.62988713383675
test epoch accuracy: 0.81212
epoch=36
epoch_loss=129.86934165493585
epoch accuracy: 0.94848
test_epoch_loss=186.56720584630966
test epoch accuracy: 0.80188
epoch=37
epoch_loss=125.99904792709276
epoch accuracy: 0.94924
test_epoch_loss=196.3344544172287
test epoch accuracy: 0.80436
```

```
epoch=38
epoch_loss=122.06267203763127
epoch accuracy: 0.95044
test_epoch_loss=116.10253241658211
test epoch accuracy: 0.79604
epoch=39
epoch_loss=117.98248126823455
epoch accuracy: 0.95332
test_epoch_loss=57.05914856493473
test epoch accuracy: 0.79232
epoch=40
epoch_loss=112.34399167913944
epoch accuracy: 0.95552
test_epoch_loss=50.75840865075588
test epoch accuracy: 0.80796
epoch=41
epoch_loss=106.57531978655607
epoch accuracy: 0.95824
test_epoch_loss=97.46047139167786
test epoch accuracy: 0.793
epoch=42
epoch_loss=102.25075506605208
epoch accuracy: 0.9604
test_epoch_loss=89.35435375571251
test epoch accuracy: 0.79384
epoch=43
epoch_loss=98.55007935967296
epoch accuracy: 0.96164
test_epoch_loss=14.187656991183758
test epoch accuracy: 0.7746
epoch=44
```

```
epoch_loss=92.84837522450835
epoch accuracy: 0.96504
test_epoch_loss=49.28605869412422
test epoch accuracy: 0.81676
epoch=45
epoch_loss=89.9667935622856
epoch accuracy: 0.9658
test_epoch_loss=188.05640137195587
test epoch accuracy: 0.76976
epoch=46
epoch_loss=85.55727924080566
epoch accuracy: 0.96784
test_epoch_loss=150.1174458861351
test epoch accuracy: 0.79304
epoch=47
epoch_loss=83.74535868922248
epoch accuracy: 0.96908
test_epoch_loss=108.49142968654633
test epoch accuracy: 0.81592
epoch=48
epoch_loss=78.1254064507084
epoch accuracy: 0.97164
test_epoch_loss=91.07044561207294
test epoch accuracy: 0.79256
epoch=49
epoch_loss=75.26999314967543
epoch accuracy: 0.97184
test_epoch_loss=46.23461839556694
test epoch accuracy: 0.77744
```

There are *lots* of ways to improve and go from here, and relying on the PyTorch-provided `TransformerEncoder` and `PositionalEncoding` modules makes it anything but "from scratch," but I was glad to create a basic architecture in pure PyTorch that could learn a simple NLP classification task.

Addendum:

I'm getting a few questions about my use of `vocab_size` instead of `max_len` in `PositionalEncoding`.

Here is the archived version of the pytorch tutorial I was using at the time of writing this code. I have updated the link above: https://web.archive.org/web/20200506194819/https://pytorch.org/tutorials/beginner/tı

It looks like I just used a different variable name that I found more descriptive; I use `def __init__(self, d_model, vocab_size=5000, dropout=0.1):`, they use `def __init__(self, d_model, dropout=0.1, max_len=5000):`.

As you can see in the archived link:

> *The positional encodings have the same dimension as the embeddings so that the two can be summed*

If you search the page, you'll see that they call the vocab size `ntokens` initially, which gets passed to `TransformerModel` as `ntoken`, then they initialize `nn.Embedding(ntoken, ninp)` (where `ninp` is `emsize`), and pass `ninp` to `PositionalEncoding` as the first positional argument (`d_model`).

They then set `pe = torch.zeros(max_len, d_model)`. So if the positional embedding and the word embeddings need to have the same dimensions, then the shapes of `nn.Embedding(ntoken, ninp)` and `torch.zeros(max_len, d_model)` will need to be the same (we already know that `ninp` and `d_model` are the same).

I'm not sure why the PyTorch example uses `max_len=5000` and then does not override this default argument; for me, it made more sense to give it a name that told me where that dimension was coming from.

Hopefully this helps provide more context. Feel free to comment below if I'm thinking about this wrong, or if there's a better approach I should consider.

SHIN • 9 months ago

Thank you very much for your post. It is really helpful to me.

I am also suffering the same problem when using transformer-encoder for binary classification. My model cannot learn anything because the training loss equals 0.69.
I am trying to solve the problem by referring to your post. I have a question:
Your PositionalEncoding class initializes the pe=torch.zeros(vocab_size, d_model).
However, other posts always use pe=torch.zeros(max_len, d_model).
I am also using the latter one in my model. Unfortunately, the training loss and the validation loss are all about 0.69. The training acc and val_acc are about 0.5.
Actually, I have tried many hyperparameters, but it does not work well.

Actually, I have tried many hyperparameters, but it does not work well...

If you could give me some suggestions, I will appreciate it very much.

^ | v • Reply • Share ›

Nathan Henrie Mod ➦ SHIN • 9 months ago

Please see my explanatory addendum above, hopefully that sheds some light.

^ | v • Reply • Share ›

SHIN ➦ Nathan Henrie • 9 months ago

@Nathan Henrie Thank you for your comment. I have found the reason why the model can't learn anything. It is because the binary classified dataset confused the classifier. When I changed the dataset, the model can reach 99% accuracy.
Thanks again.

^ | v • Reply • Share ›

SHIN • 9 months ago

Thanks for your post. It is really helpful!
I have an questions. I found that most of other posts use the position encoding(d_model, max_len). However, in your case, you use (d_model, vocab_size).
Could you explain the reason for that?
Thanks

^ | v • Reply • Share ›

Nathan Henrie Mod ➦ SHIN • 9 months ago

I think you have those reversed compared to my code above. Please see my explanatory addendum above.

^ | v • Reply • Share ›

Subscribe **Add Disqus to your siteAdd DisqusAdd** **Do Not Sell My Data**

n8henrie.com

n8henrie
n8henrie

Technology, medicine, science, superstition... and having fun.
Brought to you by Nathan Henrie.