model_133_beta=1_(d_k)^0_5_d_k=20_without_using_the_alter_algo

October 18, 2023

[ ]:
```
# Author: William Chuang
# Last modified: Oct 18, 2023
# This notebook is built on the code written by Dr. Phillip Lippe.
#
## Standard libraries
import os
import numpy as np
import random
import math
import json
from functools import partial
import statistics as stat

## PyTorch
import torch
import torch.nn as nn
import torch.nn.functional as F
import torch.utils.data as data
import torch.optim as optim


# PyTorch Lightning
try:
    import pytorch_lightning as pl
except ModuleNotFoundError: # Google Colab does not have PyTorch Lightning␣
↪installed by default. Hence, we do it here if necessary
    !pip install --quiet pytorch-lightning>=1.4
    import pytorch_lightning as pl
from pytorch_lightning.callbacks import LearningRateMonitor, ModelCheckpoint

# Path to the folder where the datasets are/should be downloaded (e.g. CIFAR10)
DATASET_PATH = "./data"
# Path to the folder where the pretrained models are saved
CHECKPOINT_PATH = "./saved_models/tutorial6"
```

```
# Setting the seed
pl.seed_everything(42)

# Ensure that all operations are deterministic on GPU (if used) for␣
↪reproducibility
torch.backends.cudnn.deterministic = True
torch.backends.cudnn.benchmark = False

device = torch.device("cuda:0") if torch.cuda.is_available() else torch.
↪device("cpu")
print("Device:", device)


def scaled_dot_product(q, k, v, mask=None):
    d_k = q.size()[-1]
    PATH = "./tmp.pth"

    #torch.save(reverse_model.state_dict(), PATH)
    #w=torch.load(PATH)
    #d=sigma=torch.std((w["transformer.layers.0.self_attn.qkv_proj.weight"])).
↪item()
    #print(d_k)
    attn_logits = torch.matmul(q, k.transpose(-2, -1))
    attn_logits = attn_logits / (math.sqrt(d_k)) #*0.15 #math.sqrt(0.005*d_k)␣
↪#0.005 #10 #3 #1.414 #(0.00001*d_k) #(d_k)**(1/100) #math.sqrt(d_k*2)
    #print(attn_logits)
    if mask is not None:
        attn_logits = attn_logits.masked_fill(mask == 0, -9e15)
    attention = F.softmax(attn_logits, dim=-1)
    values = torch.matmul(attention, v)
    return values, attention
class MultiheadAttention(nn.Module):

    def __init__(self, input_dim, embed_dim, num_heads):
        super().__init__()
        assert embed_dim % num_heads == 0, "Embedding dimension must be 0␣
↪modulo number of heads."

        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads

        # Stack all weight matrices 1...h together for efficiency
        # Note that in many implementations you see "bias=False" which is␣
↪optional
        self.qkv_proj = nn.Linear(input_dim, 3*embed_dim)
```

```python
        self.o_proj = nn.Linear(embed_dim, embed_dim)

        self._reset_parameters()

    def _reset_parameters(self):
        # Original Transformer initialization, see PyTorch documentation
        nn.init.xavier_uniform_(self.qkv_proj.weight)
        self.qkv_proj.bias.data.fill_(0)
        nn.init.xavier_uniform_(self.o_proj.weight)
        self.o_proj.bias.data.fill_(0)

    def forward(self, x, mask=None, return_attention=False):
        batch_size, seq_length, _ = x.size()
        if mask is not None:
            mask = expand_mask(mask)
        qkv = self.qkv_proj(x)

        # Separate Q, K, V from linear output
        qkv = qkv.reshape(batch_size, seq_length, self.num_heads, 3*self.
  ↪head_dim)
        qkv = qkv.permute(0, 2, 1, 3) # [Batch, Head, SeqLen, Dims]
        q, k, v = qkv.chunk(3, dim=-1)


        # Determine value outputs
        values, attention = scaled_dot_product(q, k, v, mask=mask)
        values = values.permute(0, 2, 1, 3) # [Batch, SeqLen, Head, Dims]
        values = values.reshape(batch_size, seq_length, self.embed_dim)
        o = self.o_proj(values)

        if return_attention:
            return o, attention
        else:
            return o
class EncoderBlock(nn.Module):

    def __init__(self, input_dim, num_heads, dim_feedforward, dropout=0.0):
        """
        Inputs:
            input_dim - Dimensionality of the input
            num_heads - Number of heads to use in the attention block
            dim_feedforward - Dimensionality of the hidden layer in the MLP
            dropout - Dropout probability to use in the dropout layers
        """
        super().__init__()

        # Attention layer
```

```python
        self.self_attn = MultiheadAttention(input_dim, input_dim, num_heads)

        # Two-layer MLP
        self.linear_net = nn.Sequential(
            nn.Linear(input_dim, dim_feedforward),
            nn.Dropout(dropout),
            nn.ReLU(inplace=True),
            nn.Linear(dim_feedforward, input_dim)
        )

        # Layers to apply in between the main layers
        self.norm1 = nn.LayerNorm(input_dim)
        self.norm2 = nn.LayerNorm(input_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask=None):
        # Attention part
        attn_out = self.self_attn(x, mask=mask)
        x = x + self.dropout(attn_out)
        x = self.norm1(x)

        # MLP part
        linear_out = self.linear_net(x)
        x = x + self.dropout(linear_out)
        x = self.norm2(x)

        return x
class TransformerEncoder(nn.Module):

    def __init__(self, num_layers, **block_args):
        super().__init__()
        self.layers = nn.ModuleList([EncoderBlock(**block_args) for _ in␣
 ↪range(num_layers)])

    def forward(self, x, mask=None):
        for l in self.layers:
            x = l(x, mask=mask)
        return x

    def get_attention_maps(self, x, mask=None):
        attention_maps = []
        for l in self.layers:
            _, attn_map = l.self_attn(x, mask=mask, return_attention=True)
            attention_maps.append(attn_map)
            x = l(x)
        return attention_maps
class PositionalEncoding(nn.Module):
```

```
    def __init__(self, d_model, max_len=5000):
        """
        Inputs
            d_model - Hidden dimensionality of the input.
            max_len - Maximum length of a sequence to expect.
        """
        super().__init__()

        # Create matrix of [SeqLen, HiddenDim] representing the positional␣
↪encoding for max_len inputs
        pe = torch.zeros(max_len, d_model)
        position = torch.arange(0, max_len, 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)

        # register_buffer => Tensor which is not a parameter, but should be␣
↪part of the modules state.
        # Used for tensors that need to be on the same device as the module.
        # persistent=False tells PyTorch to not add the buffer to the state␣
↪dict (e.g. when we save the model)
        self.register_buffer('pe', pe, persistent=False)

    def forward(self, x):
        x = x + self.pe[:, :x.size(1)]
        return x
class TransformerPredictor(pl.LightningModule):

    def __init__(self, input_dim, model_dim, num_classes, num_heads,␣
↪num_layers, lr, warmup, max_iters, dropout=0.0, input_dropout=0.0):
        """
        Inputs:
            input_dim - Hidden dimensionality of the input
            model_dim - Hidden dimensionality to use inside the Transformer
            num_classes - Number of classes to predict per sequence element
            num_heads - Number of heads to use in the Multi-Head Attention␣
↪blocks
            num_layers - Number of encoder blocks to use.
            lr - Learning rate in the optimizer
            warmup - Number of warmup steps. Usually between 50 and 500
            max_iters - Number of maximum iterations the model is trained for.␣
↪This is needed for the CosineWarmup scheduler
            dropout - Dropout to apply inside the model
```

```
            input_dropout - Dropout to apply on the input features
        """
        super().__init__()
        self.save_hyperparameters()
        self._create_model()

    def _create_model(self):
        # Input dim -> Model dim
        self.input_net = nn.Sequential(
            nn.Dropout(self.hparams.input_dropout),
            nn.Linear(self.hparams.input_dim, self.hparams.model_dim)
        )
        # Positional encoding for sequences
        self.positional_encoding = PositionalEncoding(d_model=self.hparams.
↪model_dim)
        # Transformer
        self.transformer = TransformerEncoder(num_layers=self.hparams.
↪num_layers,
                                              input_dim=self.hparams.model_dim,
                                              dim_feedforward=2*self.hparams.
↪model_dim,
                                              num_heads=self.hparams.num_heads,
                                              dropout=self.hparams.dropout)
        # Output classifier per sequence lement
        self.output_net = nn.Sequential(
            nn.Linear(self.hparams.model_dim, self.hparams.model_dim),
            nn.LayerNorm(self.hparams.model_dim),
            nn.ReLU(inplace=True),
            nn.Dropout(self.hparams.dropout),
            nn.Linear(self.hparams.model_dim, self.hparams.num_classes)
        )

    def forward(self, x, mask=None, add_positional_encoding=True):
        """
        Inputs:
            x - Input features of shape [Batch, SeqLen, input_dim]
            mask - Mask to apply on the attention outputs (optional)
            add_positional_encoding - If True, we add the positional encoding␣
↪to the input.
                                      Might not be desired for some tasks.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        x = self.transformer(x, mask=mask)
        x = self.output_net(x)
        return x
```

```python
    @torch.no_grad()
    def get_attention_maps(self, x, mask=None, add_positional_encoding=True):
        """
        Function for extracting the attention matrices of the whole Transformer␣
↪for a single batch.
        Input arguments same as the forward pass.
        """
        x = self.input_net(x)
        if add_positional_encoding:
            x = self.positional_encoding(x)
        attention_maps = self.transformer.get_attention_maps(x, mask=mask)
        return attention_maps

    def configure_optimizers(self):
        optimizer = optim.Adam(self.parameters(), lr=self.hparams.lr)

        # Apply lr scheduler per step
        lr_scheduler = CosineWarmupScheduler(optimizer,
                                             warmup=self.hparams.warmup,
                                             max_iters=self.hparams.max_iters)
        return [optimizer], [{'scheduler': lr_scheduler, 'interval': 'step'}]

    def training_step(self, batch, batch_idx):
        raise NotImplementedError

    def validation_step(self, batch, batch_idx):
        raise NotImplementedError

    def test_step(self, batch, batch_idx):
        raise NotImplementedError
class ReverseDataset(data.Dataset):

    def __init__(self, num_categories, seq_len, size):
        super().__init__()
        self.num_categories = num_categories

        self.seq_len = seq_len
        self.size = size

        self.data = torch.randint(self.num_categories, size=(self.size, self.
↪seq_len))
        # self.data = torch.abs(torch.normal(0, 1, size=(self.size, self.
↪seq_len)).long())
        # torch.randint(low=0, high, size, \*, generator=None, out=None,␣
↪dtype=None,
        # layout=torch.strided, device=None, requires_grad=False) → Tensor
```

```
        print(self.num_categories)
        print(self.seq_len)
        print(self.size)
        print(self.data)

    def __len__(self):
        return self.size

    def __getitem__(self, idx):
        inp_data = self.data[idx]
        labels = torch.flip(inp_data, dims=(0,))
        return inp_data, labels
#''' Examples of torch.randint
#>>> torch.randint(3, 5, (3,))
#tensor([4, 3, 4])


#>>> torch.randint(10, (2, 2))
#tensor([[0, 2],
#        [5, 5]])


#>>> torch.randint(3, 10, (2, 2))
#tensor([[4, 5],
#        [6, 7]])'''

#'''>>> torch.normal(mean=0.5, std=torch.arange(1., 6.))
#tensor([-1.2793, -1.0732, -2.0687,  5.1177, -1.2303])'''

class ReversePredictor(TransformerPredictor):

    def _calculate_loss(self, batch, mode="train"):
        # Fetch data and transform categories to one-hot vectors
        inp_data, labels = batch
        inp_data = F.one_hot(inp_data, num_classes=self.hparams.num_classes).
 ↪float()

        # Perform prediction and calculate loss and accuracy
        preds = self.forward(inp_data, add_positional_encoding=True)
        loss = F.cross_entropy(preds.view(-1,preds.size(-1)), labels.view(-1))
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        # Logging
        self.log(f"{mode}_loss", loss)
        self.log(f"{mode}_acc", acc)
        return loss, acc
```

```python
    def training_step(self, batch, batch_idx):
        loss, _ = self._calculate_loss(batch, mode="train")
        return loss

    def validation_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="val")

    def test_step(self, batch, batch_idx):
        _ = self._calculate_loss(batch, mode="test")
def train_reverse(**kwargs):

    # Create a PyTorch Lightning trainer with the generation callback
    root_dir = os.path.join(CHECKPOINT_PATH, "ReverseTask")
    os.makedirs(root_dir, exist_ok=True)
    trainer = pl.Trainer(default_root_dir=root_dir,
                         callbacks=[ModelCheckpoint(save_weights_only=True,␣
 ↪mode="max", monitor="val_acc")],
                         accelerator="gpu" if str(device).startswith("cuda")␣
 ↪else "cpu",
                         devices=1,
                         max_epochs=10,
                         gradient_clip_val=5)
    trainer.logger._default_hp_metric = None # Optional logging argument that␣
 ↪we don't need
    trainer.callbacks
    # Check whether pretrained model exists. If yes, load it and skip training
    pretrained_filename = os.path.join(CHECKPOINT_PATH, "ReverseTask.ckpt")
    if os.path.isfile(pretrained_filename):
        print("Found pretrained model, loading...")
        model = ReversePredictor.load_from_checkpoint(pretrained_filename)
    else:
        print("Found pretrained model does not exist, generating...")
        model = ReversePredictor(max_iters=trainer.
 ↪max_epochs*len(train_loader), **kwargs)
        trainer.fit(model, train_loader, val_loader)

    # Test best model on validation and test set
    val_result = trainer.test(model, val_loader, verbose=False)
    test_result = trainer.test(model, test_loader, verbose=False)
    result = {"test_acc": test_result[0]["test_acc"], "val_acc":␣
 ↪val_result[0]["test_acc"]}

    model = model.to(device)
    return model, result
class CosineWarmupScheduler(optim.lr_scheduler._LRScheduler):

    def __init__(self, optimizer, warmup, max_iters):
```

```
        self.warmup = warmup
        self.max_num_iters = max_iters
        super().__init__(optimizer)

    def get_lr(self):
        lr_factor = self.get_lr_factor(epoch=self.last_epoch)
        return [base_lr * lr_factor for base_lr in self.base_lrs]

    def get_lr_factor(self, epoch):
        lr_factor = 0.5 * (1 + np.cos(np.pi * epoch / self.max_num_iters))
        if epoch <= self.warmup:
            lr_factor *= epoch * 1.0 / self.warmup
        return lr_factor

dataset = partial(ReverseDataset, 100, 200)
train_loader = data.DataLoader(dataset(19300), batch_size=128, shuffle=True,␣
 ↪drop_last=True, pin_memory=True)
val_loader   = data.DataLoader(dataset(1000), batch_size=128)
test_loader  = data.DataLoader(dataset(100000), batch_size=128)
inp_data, labels = train_loader.dataset[0]
print("Input data:", inp_data)
print("Labels:     ", labels)
```

```
INFO:lightning_fabric.utilities.seed:Seed set to 42

Device: cuda:0
100
200
19300
tensor([[42, 67, 76,  …, 31, 56, 94],
        [26, 59, 94,  …, 73, 59, 20],
        [19, 75, 78,  …, 18, 47, 73],
        …,
        [47, 89, 77,  …, 53,  0, 60],
        [65, 27, 71,  …, 89, 30, 48],
        [11, 35, 96,  …,  8, 29, 83]])
100
200
1000
tensor([[90, 10, 66,  …, 42, 91, 69],
        [96,  7, 50,  …, 24, 25, 50],
        [84, 87, 85,  …, 15, 89, 11],
        …,
        [86, 44, 58,  …, 33, 76, 63],
        [87, 45, 82,  …, 27, 88, 63],
        [77, 81, 42,  …, 17, 68, 29]])
100
200
```

```
100000
tensor([[90, 45, 29,  …, 52, 33,  1],
        [28, 80, 98,  …, 84, 98, 30],
        [35, 97, 44,  …, 14, 15, 96],
        …,
        [44, 60, 53,  …, 97, 31, 39],
        [64, 38, 89,  …, 17, 81, 29],
        [91, 55, 53,  …, 33, 44, 62]])
Input data: tensor([42, 67, 76, 14, 26, 35, 20, 24, 50, 13, 78, 14, 10, 54, 31,
72, 15, 95,
        67,  6, 49, 76, 73, 11, 99, 13, 41, 69, 87, 19, 72, 80, 75, 29, 33, 64,
        39, 76, 32, 10, 86, 22, 77, 19,  7, 23, 43, 94, 93, 77, 70,  9, 70, 39,
        86, 99, 15, 84, 78,  8, 66, 30, 40, 60, 70, 61, 23, 20, 11, 61, 77, 89,
        84, 53, 48,  9, 83,  7, 58, 91, 14, 91, 36,  3, 82, 90, 89, 28, 55, 33,
        27, 47, 65, 89, 41, 45, 61, 39, 61, 64, 70, 53, 37, 75, 17, 21,  5,  7,
        45, 38, 95, 34, 91, 21, 90, 39, 20, 29, 11, 68, 39, 36, 17, 46,  0, 79,
        65, 92, 39, 51, 47, 68, 36, 20,  6,  8, 76, 48, 10, 96, 39,  0, 55, 19,
        35, 82, 70, 18, 68,  7, 22, 82, 96, 98, 32, 77, 66, 53,  0, 83, 66, 80,
        63, 47, 93, 28, 39, 78,  2, 25, 35, 24, 58, 51, 87,  9, 22, 46, 48,  2,
        48, 62, 21, 90, 80, 76, 25, 10, 46, 69, 42, 54, 92, 70,  2, 73, 13, 31,
        56, 94])
Labels:     tensor([94, 56, 31, 13, 73,  2, 70, 92, 54, 42, 69, 46, 10, 25, 76,
80, 90, 21,
        62, 48,  2, 48, 46, 22,  9, 87, 51, 58, 24, 35, 25,  2, 78, 39, 28, 93,
        47, 63, 80, 66, 83,  0, 53, 66, 77, 32, 98, 96, 82, 22,  7, 68, 18, 70,
        82, 35, 19, 55,  0, 39, 96, 10, 48, 76,  8,  6, 20, 36, 68, 47, 51, 39,
        92, 65, 79,  0, 46, 17, 36, 39, 68, 11, 29, 20, 39, 90, 21, 91, 34, 95,
        38, 45,  7,  5, 21, 17, 75, 37, 53, 70, 64, 61, 39, 61, 45, 41, 89, 65,
        47, 27, 33, 55, 28, 89, 90, 82,  3, 36, 91, 14, 91, 58,  7, 83,  9, 48,
        53, 84, 89, 77, 61, 11, 20, 23, 61, 70, 60, 40, 30, 66,  8, 78, 84, 15,
        99, 86, 39, 70,  9, 70, 77, 93, 94, 43, 23,  7, 19, 77, 22, 86, 10, 32,
        76, 39, 64, 33, 29, 75, 80, 72, 19, 87, 69, 41, 13, 99, 11, 73, 76, 49,
         6, 67, 95, 15, 72, 31, 54, 10, 14, 78, 13, 50, 24, 20, 35, 26, 14, 76,
        67, 42])
```

[ ]:
```
reverse_model, reverse_result = train_reverse(input_dim=train_loader.dataset.
↪num_categories,
                                              model_dim=200,
                                              num_heads=1,
                                              num_classes=train_loader.dataset.
↪num_categories,
                                              num_layers=1,
                                              dropout=0.0,
                                              lr=5e-4,
                                              warmup=50)
```

```
INFO:pytorch_lightning.utilities.rank_zero:GPU available: True (cuda), used:
True
```

```
INFO:pytorch_lightning.utilities.rank_zero:TPU available: False, using: 0 TPU
cores
INFO:pytorch_lightning.utilities.rank_zero:IPU available: False, using: 0 IPUs
INFO:pytorch_lightning.utilities.rank_zero:HPU available: False, using: 0 HPUs
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES:
[0]
INFO:pytorch_lightning.callbacks.model_summary:
  | Name                | Type               | Params
-----------------------------------------------------------
0 | input_net           | Sequential         | 20.2 K
1 | positional_encoding | PositionalEncoding | 0
2 | transformer         | TransformerEncoder | 322 K
3 | output_net          | Sequential         | 60.7 K
-----------------------------------------------------------
403 K     Trainable params
0         Non-trainable params
403 K     Total params
1.612     Total estimated model params size (MB)

Found pretrained model does not exist, generating…

Sanity Checking: |          | 0/? [00:00<?, ?it/s]

/usr/local/lib/python3.10/dist-
packages/pytorch_lightning/trainer/connectors/data_connector.py:441: The
'val_dataloader' does not have many workers which may be a bottleneck. Consider
increasing the value of the `num_workers` argument` to `num_workers=1` in the
`DataLoader` to improve performance.
/usr/local/lib/python3.10/dist-
packages/pytorch_lightning/trainer/connectors/data_connector.py:441: The
'train_dataloader' does not have many workers which may be a bottleneck.
Consider increasing the value of the `num_workers` argument` to `num_workers=1`
in the `DataLoader` to improve performance.

Training: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]

Validation: |          | 0/? [00:00<?, ?it/s]
```

```
INFO:pytorch_lightning.utilities.rank_zero:`Trainer.fit` stopped:
`max_epochs=10` reached.
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES:
[0]
/usr/local/lib/python3.10/dist-
packages/pytorch_lightning/trainer/connectors/data_connector.py:441: The
'test_dataloader' does not have many workers which may be a bottleneck. Consider
increasing the value of the `num_workers` argument` to `num_workers=1` in the
`DataLoader` to improve performance.
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

```
INFO:pytorch_lightning.accelerators.cuda:LOCAL_RANK: 0 - CUDA_VISIBLE_DEVICES:
[0]
```

```
Testing: |          | 0/? [00:00<?, ?it/s]
```

[ ]:
```python
# @title the scaling factor, beta, is 1/(d_k)^0.5, where d_k = 20, i.e. without
↪using the alternative algorithm to obtain the scaling factor first
# model_133_beta=1/(d_k)^0.5_d_k=20_without_using_the_alter_algo.ipynb
print(f"Val accuracy:  {(100.0 * reverse_result['val_acc']):4.2f}%")
print(f"Test accuracy: {(100.0 * reverse_result['test_acc']):4.2f}%")
```

```
Val accuracy:  1.22%
Test accuracy: 1.23%
```

[ ]:
```python

```