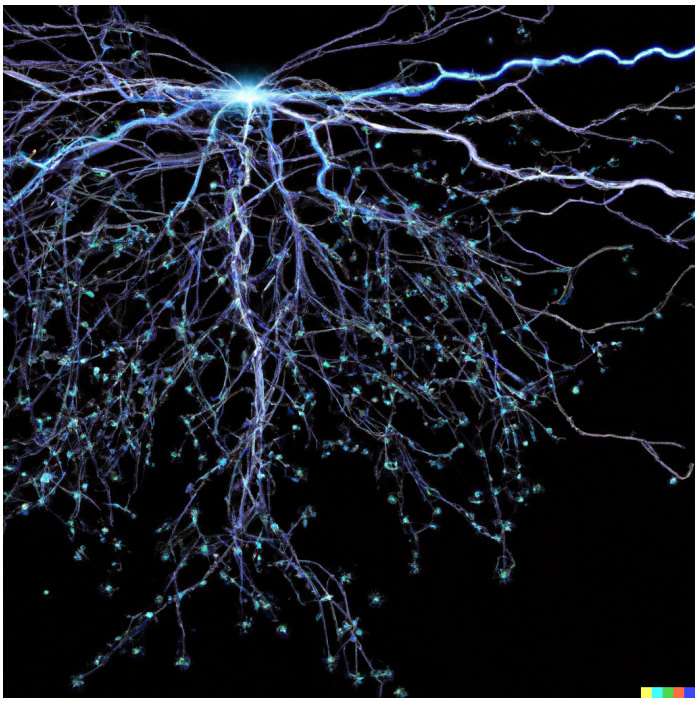
CS197 Harvard: AI Research Experiences

Fall 2022: Lecture 5 – "Lightning McTorch"
Fine-tuning a Vision Transformer using Lightning

Instructed by Pranav Rajpurkar. Website https://cs197.seas.harvard.edu/

Abstract

Reading code is often an effective way of learning. Today we will step through an image classification workflow with Vision transformers. We will parse code to process a computer vision dataset, tokenize inputs for vision transformers, and build a training workflow using the Lightning (PyTorch Lightning) framework. You might be used to learning about a new AI framework with simple tutorials first that build in complexity. However, in research settings, you'll often be faced with using codebases that use unfamiliar frameworks. Our lecture today reflects this very setting, and is thus structured as a walkthrough where you will be exposed to code that uses Pytorch Lightning and then proceed to understand parts of it.



DALL-E Generation: "A bolt of lightning strikes a neural network"

Learning outcomes:

- Interact with code to explore data loading and tokenization of images for Vision Transformers.
- Parse code for PyTorch architecture and modules for building a Vision Transformer.
- Get acquainted with an example training workflow with PyTorch Lightning.

Fine-Tuning A Vision Transformer

In lecture 4, we fine-tuned a GPT-2 language model to auto-complete text. Today we are switching domains from natural language processing to computer vision, which will give you a sense of how data processing and tokenization vary between text and images. We are going to focus in particular on image classification: given an image, which of the following 10 classes is it an image of: airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks.

We will follow the vision transformer tutorial, but with some of my own spin on it. You can find my final notebook here.

Lightning

We are going to use the Lightning library (formerly PyTorch Lightning). PyTorch Lightning is described as 'the deep learning framework with "batteries included".' Lightning is a layer on top of PyTorch to organize code to remove boilerplate; it also abstracts away all the engineering complexity needed for scale.

You may remembers that in the last lecture, we used Huggingface's transformers library which too had a Trainer class. How is the transformers library different from lightning? One answer:

> *"The HuggingFace Trainer API can be seen as a framework similar to PyTorch Lightning in the sense that it also abstracts the training away using a Trainer object. However, contrary to PyTorch Lightning, it is not meant to be a general framework. Rather, it is made especially for fine-tuning Transformer-based models available in the HuggingFace Transformers library."*

I'd recommend going through the basic skills in Lightning.

Installation

We're going to use:

```
conda create --name lec5 python=3.9
conda activate lec5
pip install --quiet "setuptools==59.5.0" "pytorch-lightning>=1.4"
"matplotlib" "torch>=1.8" "ipython[notebook]" "torchmetrics>=0.7"
"torchvision" "seaborn"
```

Code Walkthrough

In the following sections, we will look through parts of a notebook, and line-by-line try to understand what it is trying to do.

Data Loading

Let's begin:

```
1  from torchvision import transforms
2  from torchvision.datasets import CIFAR10
3  import pytorch_lightning as pl
4  import os
5
6  DATASET_PATH = os.environ.get("PATH_DATASETS", "data/")
```

L1-4: We're importing libraries. Remember that Python code in one module gains access to the code in another module by the process of importing it. You can import a resource directly, as in line 4. You can import the resource from another package or module, as in lines 1 and 2. You can also choose to rename an imported resource, like in line 3.

L6: We can use the get() method to return a default value instead of None if the key-value pair is not present for the key specified by providing the default value as a second argument to the get() method. Here, we're setting the DATASET_PATH to use the environment variable if it exists, and if not use "data/".

```
8  train_transform = transforms.Compose(
9      [
10         transforms.RandomHorizontalFlip(),
11         transforms.RandomResizedCrop(
12             (32, 32), scale=(0.8, 1.0), ratio=(0.9, 1.1)),
13         transforms.ToTensor(),
14         transforms.Normalize([0.49139968, 0.48215841, 0.44653091],
15             [0.24703223, 0.24348513, 0.26158784]),
16     ]
17 )
18
19 test_transform = transforms.Compose(
20     [
21         transforms.ToTensor(),
22         transforms.Normalize([0.49139968, 0.48215841, 0.44653091],
23         [0.24703223, 0.24348513, 0.26158784]),
24     ]
25 )
```

L8-17: We're composing transformations. The compositions are performed in sequence. We can read details on these in the docs, but I'm going to highlight a few things:

- L10: we horizontally flip the given image randomly with a given probability. The probability of the image being flipped is at default 0.5.
- L11-12: A crop of the original image is made: the crop has a random area (H * W) and a random aspect ratio. This crop is finally resized to the given size. scale (tuple of float)

specifies the lower and upper bounds for the random area of the crop before resizing; while ratio (tuple of float) specifies the lower and upper bounds for the random aspect ratio of the crop, before resizing.

- L13: Converts a PIL Image (common image format) or numpy.ndarray (H x W x C) in the range [0, 255] to a torch.FloatTensor of shape (C x H x W) in the range [0.0, 1.0].
- L14-15: Normalizes a tensor image with mean and standard deviation. This transform will normalize each channel of the input using the precomputed means and standard deviation for the CIFAR dataset that we will use. The constants correspond to the values that scale and shift the data to a zero mean and standard deviation of one.
-

L19-24:

- What we should be initially surprised by here is that we are applying a different set of transforms than the train_transforms. Think about why this might be! Answer: the train transforms help augment the data to give the dataset more examples, but in test time, we don't want to corrupt the examples by performing augmentations like cropping them. Pro tip: there are competition strategies to apply what's called test time augmentations, where multiple augmented images are passed through the network and their outputs averaged to get a more performant model.

```
26
27  train_dataset = CIFAR10(
28      root=DATASET_PATH, train=True, transform=train_transform,download=True)
29  val_dataset = CIFAR10(
30      root=DATASET_PATH, train=True, transform=test_transform, download=True)
31  test_set = CIFAR10(
32      root=DATASET_PATH, train=False, transform=test_transform,download=True)
```

L27-L28: We are loading up the CIFAR10 dataset by instantiating it. We can see the documentation here. We can specify the root directory of the dataset where the download will be saved, whether or not we load the train (vs test) set, and what transform we apply.
L29-30: Do you see anything weird? Note that we're loading the same dataset as in the train dataset, but applying a different transformation (the test transform). Something seems wrong here (we'll see how this works out in the future).
L31-32: Note that we're applying the same transform to the test set as we do to the validation set because we want the validation set to help us pick a model that will perform well on the test set.

```python
import torch
import torch.utils.data as data

pl.seed_everything(42)
train_set, _ = torch.utils.data.random_split(train_dataset, [45000, 5000])
pl.seed_everything(42)
_, val_set = torch.utils.data.random_split(val_dataset, [45000, 5000])

```

✓ 0.3s Python

L4: We're using the function seed_everything (documentation here). We can see that this function sets the seed for pseudo-random number generators in: pytorch, numpy, python.random, and in addition, sets a couple of environment variables.
L5,L7: The random_split method randomly splits a dataset into non-overlapping new datasets of given lengths (source). However, this seems like a weird hack: remember that the train_dataset and val_dataset loaded the same data and transformed it in two different ways. Here it looks like we're able to make the train_set and val_set use different sets of images, which is what we'd like to evaluate generalization.

Exercise: change the setup such that you first split the training dataset once into a train set and val set before applying the train and test transforms.
Let's continue:

```python
import matplotlib.pyplot as plt
import torchvision

# Visualize some examples
NUM_IMAGES = 4
CIFAR_images = torch.stack(
    [val_set[idx][0] for idx in range(NUM_IMAGES)], dim=0)
img_grid = torchvision.utils.make_grid(
    CIFAR_images, nrow=4, normalize=True, pad_value=0.9)
img_grid = img_grid.permute(1, 2, 0)

plt.figure(figsize=(8, 8))
plt.title("Image examples of the CIFAR10 dataset")
plt.imshow(img_grid)
plt.axis("off")
plt.show()
plt.close()

```

✓ 0.9s Python

L6-7: This is the interesting bit. We're sampling the first 4 images in the val_set to show as examples. If we print val_set[0], we'll see it's a tuple of (image, label), so val_set[0][0] gets us to the image. We're using stack (documentation) on the 0th dimension, which will concatenate the sequence of tensors passed to it in the 0th dimension, giving us CIFAR_images, a torch tensor of the shape (4, 3, 32, 32). Quiz: what is each of the dimensions?
L1-5,L8-18: This is visualization code which I don't find critical for us at this point; so we will skip it. Note that like reading a paper, there are some details which we will have to skip in our first pass through code, like there are details we skip in our first pass of a paper.

```
1  train_loader = data.DataLoader(
2      train_set, batch_size=128,
3      shuffle=True, drop_last=True, pin_memory=True, num_workers=4)
4  val_loader = data.DataLoader(
5      val_set, batch_size=128,
6      shuffle=False, drop_last=False, num_workers=4)
7  test_loader = data.DataLoader(
8      test_set, batch_size=128,
9      shuffle=False, drop_last=False, num_workers=4)
```

.] ✓ 0.5s Python

L1-3: We're using a DataLoader now. The dataloader documentation specifies that it "combines a dataset and a sampler, and provides an iterable over the given dataset". In simpler terms, it allows us to iterate over a dataset in batches given by the batch size. Shuffle=true makes sure to have the data reshuffled at every epoch; this improves performance. This is because gradient descent relies on randomization to get out of local minimas. We set drop_last Tru` to drop the last incomplete batch if the dataset size is not divisible by the batch size. num_workers specifies how many subprocesses to use for data loading.

How do you set num_workers? As Lightning docs say:

- *The question of how many workers to specify in num_workers is tricky. Here's a summary of some references, and our suggestions:*
  - *num_workers=0 means ONLY the main process will load batches (that can be a bottleneck).*
  - *num_workers=1 means ONLY one worker (just not the main process) will load data, but it will still be slow.*
  - *The performance of high num_workers depends on the batch size and your machine.*
  - *A general place to start is to set num_workers equal to the number of CPU cores on that machine. You can get the number of CPU cores in*

python using os.cpu_count(), but note that depending on your batch size, you may overflow RAM memory.

- *WARNING: Increasing num_workers will ALSO increase your CPU memory consumption.*
- *Best practice: The best thing to do is to increase the num_workers slowly and stop once there is no more improvement in your training speed. For debugging purposes or for dataloaders that load very small datasets, it is desirable to set num_workers=0.*

Finally, the pin_memory argument is tricky too. We will ignore it for now, but if you're curious, check out the following.

Let's keep going!

Tokenization

```python
def img_to_patch(x, patch_size, flatten_channels=True):
    """
    Inputs:
        x - Tensor representing the image of shape [B, C, H, W]
        patch_size - Number of pixels per dimension of the
            patches (integer)
        flatten_channels - If True, the patches will be
            returned in a flattened format as a feature
            vector instead of a image grid.
    """
    B, C, H, W = x.shape
    x = x.reshape(
        B,
        C,
        torch.div(H, patch_size, rounding_mode='trunc'),
        patch_size,
        torch.div(W, patch_size, rounding_mode='floor'),
        patch_size,
    )
    x = x.permute(0, 2, 4, 1, 3, 5)  # [B, H', W', C, p_H, p_W]
    x = x.flatten(1, 2)  # [B, H'*W', C, p_H, p_W]
    if flatten_channels:
        x = x.flatten(2, 4)  # [B, H'*W', C*p_H*p_W]
    return x

img_patches = img_to_patch(
    CIFAR_images, patch_size=4, flatten_channels=False)
```

✓ 0.2s Python

L11-19: Okay we're getting into some transformers specific code now. The Vision Transformer is a model for image classification that views images as sequences of smaller patches. So as a preprocessing step, we split an image of, for example, 32 x 32 pixels into a grid of 8 x 8 of size 4 x 4 each. This is exactly what's going on here. Note that the Batch and Channels dimensions are untouched, and we're working to transform the Height and Width into 4 pieces: H' (L15), p_H (L16), W' (L17), p_W (L18).

L20-L21: These permute operations are simply getting us to the point at which we will have H'*W' patches for every image, and we can visualize them by looking at (C, p_H, p_W). The visualization step will happen soon enough.

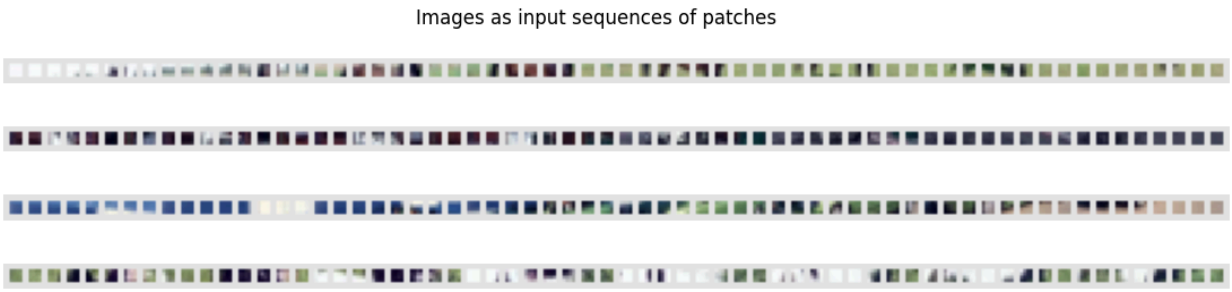
L22-23: We are now getting to a stage at which we are combining (flattening) the height and width dimension so that we have one vector of (C*p_H*p_W) elements for each of the H'*W' patches. Each of those patches is considered to be a "word"/"token".

This idea can be found in Alexey Dosovitskiy et al. https://openreview.net/pdf?id=YicbFdNTTy from the paper "An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale".

```python
fig, ax = plt.subplots(CIFAR_images.shape[0], 1, figsize=(14, 3))
fig.suptitle("Images as input sequences of patches")
for i in range(CIFAR_images.shape[0]):
    img_grid = torchvision.utils.make_grid(
        img_patches[i], nrow=64, normalize=True, pad_value=0.9)
    img_grid = img_grid.permute(1, 2, 0)
    ax[i].imshow(img_grid)
    ax[i].axis("off")
plt.show()
plt.close()
```

✓ 0.2s Python

We can now visualize each of the patches. Notice the make_grid function making a reappearance so we will look at its documentation: it takes in a 4D mini-batch Tensor of shape (B x C x H x W) or a list of images all of the same size; nrow sets the number of images displayed in each row of the grid; normalize shifts the image to the range (0, 1), by the min and max values specified (here by default, the min and max over the input tensor). Finally, pad_value sets the value for the padded pixels.



Exercise: Run the above visualization, noting how these patches compare to the original images. What happens without L6?

Neural Net Module

```python
import torch.nn as nn

class AttentionBlock(nn.Module):
    def __init__(self, embed_dim, hidden_dim, num_heads, dropout=0.0):
        super().__init__()

        self.layer_norm_1 = nn.LayerNorm(embed_dim)
        self.attn = nn.MultiheadAttention(embed_dim, num_heads)
        self.layer_norm_2 = nn.LayerNorm(embed_dim)
        self.linear = nn.Sequential(
            nn.Linear(embed_dim, hidden_dim),
            nn.GELU(),
            nn.Dropout(dropout),
            nn.Linear(hidden_dim, embed_dim),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        inp_x = self.layer_norm_1(x)
        x = x + self.attn(inp_x, inp_x, inp_x)[0]
        x = x + self.linear(self.layer_norm_2(x))
        return x
```

✓ 0.2s Python

We now get into code for the AttentionBlock. Note that nn.Module is the base class for all neural network modules. Our models should also subclass this class.Modules can also contain other Modules, allowing us to nest them in a tree structure using attributes. We typically implement the __init__ and forward methods for the nn.Module subclasses.

L5: As per the example, an ``__init__()`` call to the parent class must be made.
L7-L11: Here, we set the attributes of the AttentionBlock to be other Modules, including LayerNorm, MultiheadAttention, and a Sequential container. In a Sequential container, Modules are added to it in the order they are passed in the constructor. The forward() method of Sequential accepts any input and forwards it to the first module it contains. It then "chains" outputs to inputs sequentially for each subsequent module, finally returning the output of the last module. We have several different chained modules, including a Linear, GELU, & Dropout.

Exercise: Find out the documentation for Linear, GELU, Dropout, LayerNorm and Multihead Attention and describe what they do!

L18-22: Here in the forward method, we compute output Tensors from input Tensors – this computation is often referred to as the forward pass of the network. Here, we see how a bunch of operations are applied to x.
Exercise: Read this documentation and explain what L20 is doing. Describe the concept in 5-10 lines. Hint: you may have to reference the paper linked in the documentation./

```
1  class VisionTransformer(nn.Module):
2      def __init__(
3          self,
4          embed_dim,
5          hidden_dim,
6          num_channels,
7          num_heads,
8          num_layers,
9          num_classes,
10         patch_size,
11         num_patches,
12         dropout=0.0,
13     ):
14         super().__init__()
15         self.patch_size = patch_size
16
17         # Layers/Networks
18         self.input_layer = nn.Linear(
19             num_channels * (patch_size**2), embed_dim)
20         self.transformer = nn.Sequential(
21             *(AttentionBlock(
22                 embed_dim, hidden_dim, num_heads, dropout=dropout)
23                     for _ in range(num_layers))
24         )
25         self.mlp_head = nn.Sequential(
26             nn.LayerNorm(embed_dim), nn.Linear(embed_dim, num_classes))
27         self.dropout = nn.Dropout(dropout)
28
29         # Parameters/Embeddings
30         self.cls_token = nn.Parameter(
31             torch.randn(1, 1, embed_dim))
32         self.pos_embedding = nn.Parameter(
33             torch.randn(1, 1 + num_patches, embed_dim))
```

Notice how the VisionTransformer Module now has now nested the previously seen AttentionBlocks in L20-24. In addition, we have:

- L18-20: A linear projection layer that maps the input patches (each of num_channels * patch_size**2) to a feature vector of larger size (embed dim).

- L25-26: A multi-layer perceptron (MLP head) that takes an output feature vector and maps it to a classification prediction.

Exercise: What do lines 30-33 do? Hint: See https://huggingface.co/docs/transformers/model_doc/vit and read https://arxiv.org/pdf/1706.03762.pdf

```
35      def forward(self, x):
36          x = img_to_patch(x, self.patch_size)
37          B, T, _ = x.shape
38          x = self.input_layer(x)
39
40          cls_token = self.cls_token.repeat(B, 1, 1)
41          x = torch.cat([cls_token, x], dim=1)
42          x = x + self.pos_embedding[:, : T + 1]
43
44          x = self.dropout(x)
45          x = x.transpose(0, 1)
46          x = self.transformer(x)
47
48          cls = x[0]
49          out = self.mlp_head(cls)
50          return out
```

Now, in the forward function, we see the modules come together.
L36-38: Notice that we're calling img_to_patch, then passing through the input_layer.
L40-41: We're prepending the classification token for every sample in the batch.
L42: Here, we're doing a sum of the positional embeddings with our x. Notice how pos_embeddings is of shape [1, 65, 256] and x is of shape [B, 65, 256] and yet we're able to sum them up, applying the pos_embeddings to every sample in the batch. This is called broadcasting.
L44-L50: We're continuing to apply operations in sequence, and finally taking the classification head output.

Exercise: what is the purpose of L46?

Lightning Module

Let's continue our exploration!

```python
import torch.nn.functional as F
import torch.optim as optim

class ViT(pl.LightningModule):
    def __init__(self, model_kwargs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.model = VisionTransformer(**model_kwargs)

    def forward(self, x):
        return self.model(x)

    def configure_optimizers(self):
        optimizer = optim.AdamW(self.parameters(), lr=self.hparams.lr)
        lr_scheduler = optim.lr_scheduler.MultiStepLR(
            optimizer, milestones=[100, 150], gamma=0.1)
        return [optimizer], [lr_scheduler]

    def _calculate_loss(self, batch, mode="train"):
        imgs, labels = batch
        preds = self.model(imgs)
        loss = F.cross_entropy(preds, labels)
        acc = (preds.argmax(dim=-1) == labels).float().mean()

        self.log("%s_loss" % mode, loss, prog_bar=True)
        self.log("%s_acc" % mode, acc, prog_bar=True)
        return loss

    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")
```

✓ 0.2s Python

Here, we see the Lightning Module. A LightningModule organizes your PyTorch code into sections including:

- Computations (L5-9)
- Forward: Used for inference only (separate from training_step, L10-11)
- Optimizer and scheduler (through configure_optimizers, L13-17). The optimizer takes in the parameters and determines how the parameters are updated. The scheduler contains the optimizer as a member and alters its parameters learning rates. We don't need to worry about these for now.
- Training Loop (training_step, L29-31)
- Validation Loop (validation_step, L33-35)
- Test Loop (test_step, L 36-37)

All of the training, validation and test loops use _calculate_loss, which computes the cross_entropy loss for the batch comparing the predictions (preds) of the model with the labels, logging the accuracy in the process. Note how all of the functions receive a batch, which unpacks into the images and the labels.

```python
CHECKPOINT_PATH = os.environ.get(
    "PATH_CHECKPOINT",
    "saved_models/VisionTransformers/")

def train_model(**kwargs):
    trainer = pl.Trainer(
        default_root_dir=os.path.join(CHECKPOINT_PATH, "ViT"),
        fast_dev_run=5,
    )

    pl.seed_everything(42)  # To be reproducible
    model = ViT(**kwargs)
    trainer.fit(model, train_loader, val_loader)
    test_result = trainer.test(
        model, dataloaders=test_loader, verbose=False)
    return model, test_result
```

✓ ✓ ✓ 0.2s Python

Finally, we have a Trainer. Once we have got a LightningModule, the Trainer automates everything else. The basic use of the trainer is to initialize it (L4-7), and then fit the model using the train_loader and the val_loader (L11). We then use the test method on the Trainer using the test loader (L12). Read about what the Trainer does under the hood here.

Exercise: Find out what the default_root_dir and fast_dev_run arguments for constructing the Trainer object do.

```python
model, results = train_model(
    model_kwargs={
        "embed_dim": 256,
        "hidden_dim": 512,
        "num_heads": 8,
        "num_layers": 6,
        "patch_size": 4,
        "num_channels": 3,
        "num_patches": 64,
        "num_classes": 10,
        "dropout": 0.2,
    },
    lr=3e-4,
)
print("Results", results)
```

✓ 3.3s Python

This code finally executes the model training (and evaluation). Congratulations, we've completed our walkthrough.