

# Session 13 - PyTorch Lightning and AI Application Development

- Due Saturday by 9am
- Points 0
- Available after Apr 20 at 11:30am

## Session 13 - PyTorch Lightning, Fabrik, and Spaces

Lightning is a PyTorch wrapper for high-performance AI research, scaling the models and disentangling PyTorch code to decouple the science from engineering.

Lightning structures PyTorch code with these principles:

### Maximal flexibility

```
def training_step(self, batch, batch_nb):
    x, y = batch
    z = self.encoder(x)
    x_hat = self.decoder(z)
    mse = F.mse_loss(x_hat, x)
    gan_regularizer = self.discriminator(x_hat)
    loss = mse + gan_regularizer
    return loss
```

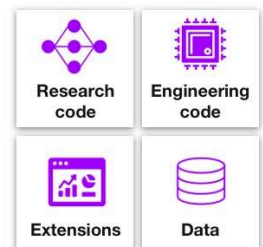
### No boilerplate Maximal flexibility

```
if gpu:
    x = x.cuda(0)
    z = encoder(x)
    x_hat = decoder(z)
    backward()
```

### Self contained models



### Modular



Lightning forces the following structure to our code which makes it reusable and shareable:

- Research code (the lightning module)
- Engineering code (handled by the Trainer)
- Non-essential research code (logging, etc.. goes to the callbacks)
- Data (use PyTorch DataLoaders or organize them into a LightningDataModule)

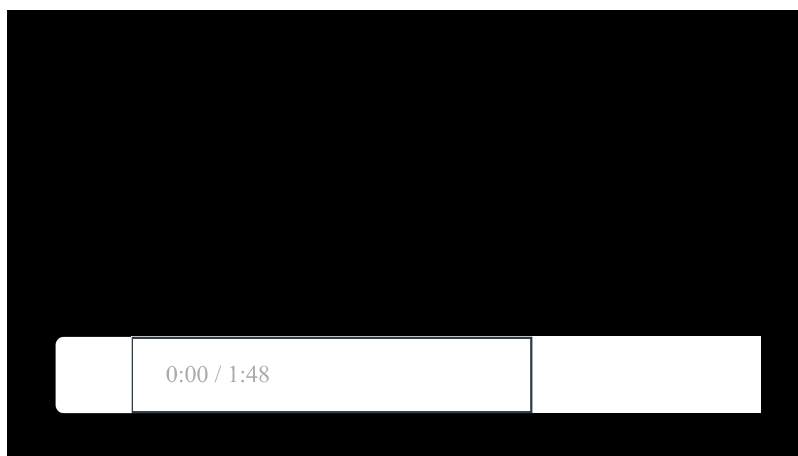
Once we do this, we can train on multiple GPUs, TPUs, CPUs, and even in 16-bit precision without changing our code!

## Code Organization

Organizing our code with Lightning makes our code:

- keep all the flexibility (this is all pure PyTorch), but remove a ton of boilerplate
- More readable by decoupling the research code from the engineering
- Easier to reproduce
- Less error-prone by automating most of the training loop and tricky engineering
- Scalable to any hardware without changing the model

Let's look at a short video to see how it's done:

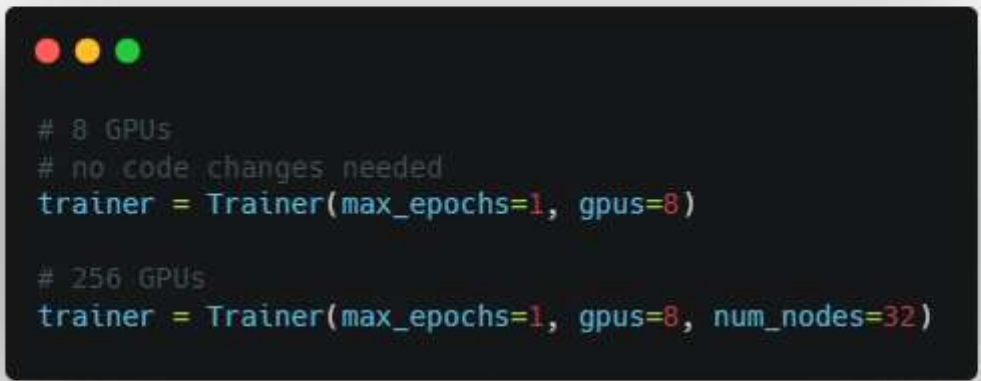


## HandsOn

Let's do some [hands-on](#) 

(<https://colab.research.google.com/drive/1onWkJkeLRHmXfK79AcMyL0gwHNuu-je1?usp=sharing>)\_.


## Some Code-notes



```
# 8 GPUs
# no code changes needed
trainer = Trainer(max_epochs=1, gpus=8)

# 256 GPUs
trainer = Trainer(max_epochs=1, gpus=8, num_nodes=32)
```

you can also write `trainer = Trainer(accelerator="auto")` mode, where it can find the system you're on and select the appropriate Accelerator.



```
# no code changes needed to move to TPU
trainer = Trainer(tpu_cores=8)
```

Accumulated gradients run K small batches of size N before doing a backward pass. The effect is a large effective batch size of size  $K \times N$ .



```
# DEFAULT (ie: no accumulated grads)
trainer = Trainer(accumulate_grad_batches=1)
```

Stochastic Weight Averaging (SWA) can make your models generalize better at virtually no additional cost. This can be used with both non-trained and trained models. The SWA procedure smooths the loss landscape thus making it harder to end up in a local minimum during optimization.

For a more detailed explanation of SWA and how it works, read [this post](https://pytorch.org/blog/pytorch-1.6-now-includes-stochastic-weight-averaging) (<https://pytorch.org/blog/pytorch-1.6-now-includes-stochastic-weight-averaging>) by the PyTorch team.

```
# Enable Stochastic Weight Averaging using the callback  
trainer = Trainer(callbacks=[StochasticWeightAveraging(...)])
```

Auto-scaling of batch size may be enabled to find the largest batch size that fits into memory.

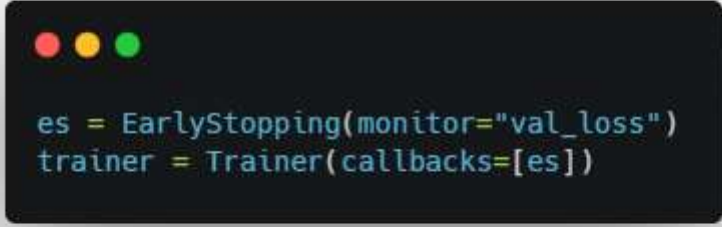
```
# DEFAULT (ie: don't scale batch size automatically)  
trainer = Trainer(auto_scale_batch_size=None)  
  
# Autoscale batch size  
trainer = Trainer(auto_scale_batch_size=None | "power" | "binsearch")  
  
# find the batch size  
trainer.tune(model)
```

```
# no code changes needed  
trainer = Trainer(precision=16)
```

*Trainer(gpus=1, precision="bf16") is also available*

```
from pytorch_lightning import loggers  
  
# tensorboard  
trainer = Trainer(logger=TensorBoardLogger("logs/"))  
  
# weights and biases  
trainer = Trainer(logger=loggers.WandbLogger())  
  
# comet  
trainer = Trainer(logger=loggers.CometLogger())  
  
# mlflow  
trainer = Trainer(logger=loggers.MLFlowLogger())  
  
# neptune  
trainer = Trainer(logger=loggers.NeptuneLogger())
```

**WANDB**  (<https://docs.wandb.ai/guides/integrations/lightning>) and others.



```
es = EarlyStopping(monitor="val_loss")
trainer = Trainer(callbacks=[es])
```

The `EarlyStopping` [callback](https://pytorch-lightning.readthedocs.io/en/latest/api/pytorch_lightning.callbacks.early_stopping.html#pytorch_lightning.callbacks.early_stopping) can be used to monitor a validation metric and stop the training when no improvement is observed.

To enable it:

- Import `EarlyStopping` [callback](https://pytorch-lightning.readthedocs.io/en/latest/api/pytorch_lightning.callbacks.early_stopping.html#pytorch_lightning.callbacks.early_stopping).
- Log the metric you want to monitor using `log()` [method](https://pytorch-lightning.readthedocs.io/en/latest/api/pytorch_lightning.core.lightning.html#pytorch_lightning.core.LightningModule.log).
- Init the callback, and set monitor to the logged metric of your choice.
- Pass the `EarlyStopping` [callback](https://pytorch-lightning.readthedocs.io/en/latest/api/pytorch_lightning.callbacks.early_stopping.html#pytorch_lightning.callbacks.early_stopping) to the `Trainer` [callbacks](https://pytorch-lightning.readthedocs.io/en/latest/api/pytorch_lightning.trainer.trainer.html#pytorch_lightning.trainer.Trainer) flag.

```
from pytorch_lightning.callbacks.early_stopping import EarlyStopping

def validation_step(self):
    self.log("val_loss", loss)

trainer = Trainer(callbacks=[EarlyStopping(monitor="val_loss")])

# OR You can customize the callbacks behaviour by changing its parameters.

early_stop_callback = EarlyStopping(monitor="val_accuracy", min_delta=0.00, patience=3, verbose=False,
mode="max")
trainer = Trainer(callbacks=[early_stop_callback])
```

```
checkpointing = ModelCheckpoint(monitor="val_loss")
trainer = Trainer(callbacks=[checkpointing])
```

```
# onnx
with tempfile.NamedTemporaryFile(suffix=".onnx", delete=False) as tmpfile:
    autoencoder = LitAutoEncoder()
    input_sample = torch.randn((1, 64))
    autoencoder.to_onnx(tmpfile.name, input_sample, export_params=True)
    os.path.isfile(tmpfile.name)
```



This flag runs a “unit test” by running `n` if set to `n` (int) else 1 if set to `True` training and validation batch(es). The point is to detect any bugs in the training/validation loop without having to wait for a full epoch to crash.



```
# runs 1 train, val, test batch and program ends
trainer = Trainer(fast_dev_run=True)

# runs 7 train, val, test batches and program ends
trainer = Trainer(fast_dev_run=7)
```

A good debugging technique is to take a tiny portion of your data (say 2 samples per class), and try to get your model to overfit. If it can't, it's a sign it won't work with large datasets.



```
# use only 1% of training data (and turn off validation)
trainer = Trainer(overfit_batches=0.01)

# similar, but with a fixed 10 batches
trainer = Trainer(overfit_batches=10)
```

**Hyperparameters**  (<https://pytorch-lightning.readthedocs.io/en/latest/common/hyperparameters.html>)

## Note on Distributed Training

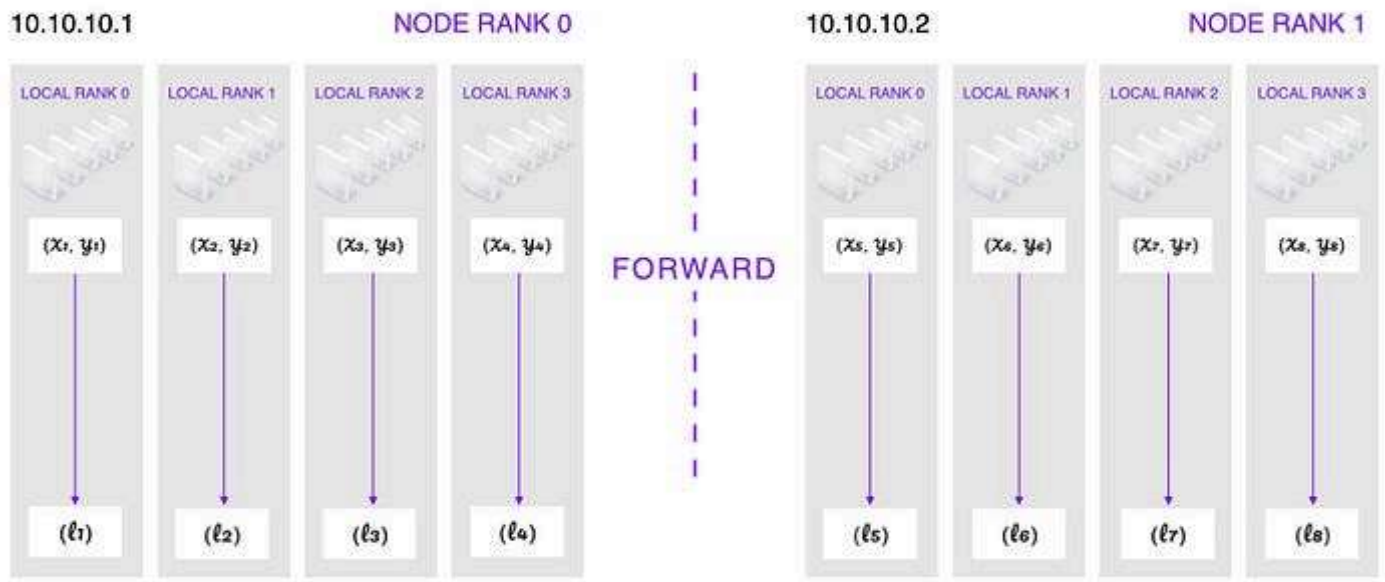
It is important to understand that the optimization in a distributed setting does not change at all when compared to the single-device setting, that is, we still minimize the same cost function with the same model and the same optimizer.

The real difference is that the gradient computation gets split into multiple devices and runs in parallel. This works simply because of the linearity of the gradient operator, i.e., computing the gradient for individual data samples and then averaging them is the same as computing the gradient using the whole batch of data at once.

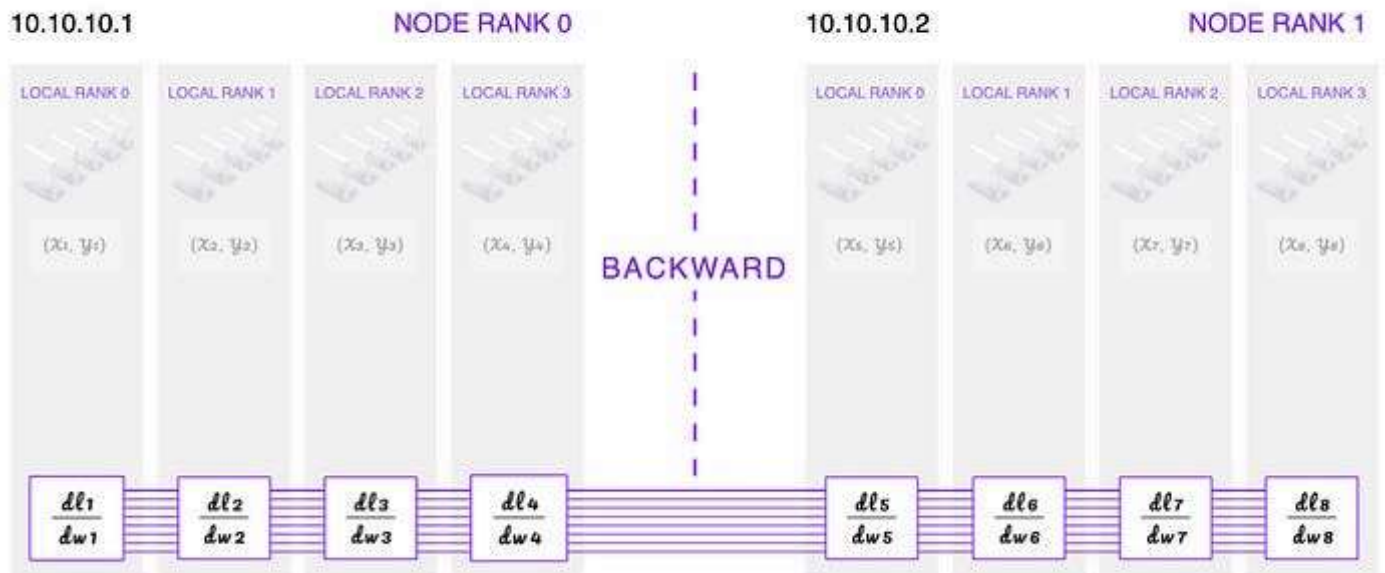
$$\frac{\partial l_0}{\partial w} + \frac{\partial l_1}{\partial w} = \frac{\partial (l_0 + l_1)}{\partial w}$$

In summary, there are four main steps involved in a single training step

STEP1: We start off with the same model weights on all devices. Each device gets its own split of the data batch and performs a forward pass. This yields a different loss value per device.



STEP2: Given the loss value, we can perform the backward pass which computes the gradients of the loss w.r.t. the model weights. We now have a different gradient per GPU device.



STEP3: We synchronize the gradients by summing them up and dividing by the number of GPU devices involved. At the end of this process, each GPU now has the same averaged gradients.

STEP4: Finally, all models can update their weights with the synchronized gradient. Because the gradient is the same on all GPUs, we again end up with the same model weights on all devices and the next training step can begin.


Leveraging multiple GPUs in vanilla PyTorch can be [overwhelming](#) 

(<https://github.com/pytorch/examples/blob/01539f9eada34aef67ae7b3d674f30634c35f468/ima>

, and to implement steps 1–4 from the theory above, a significant amount of code changes are required to “refactor” the codebase. With PyTorch Lightning, single-node training with multiple GPUs is as trivial as adding two arguments to the Trainer:



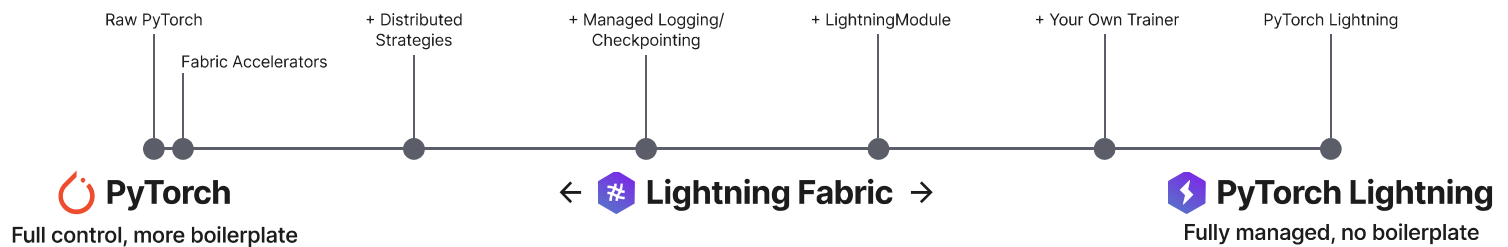
```
trainer = Trainer(gpus=8, accelerator="ddp")
```

DPP stands for Distributed Data-Parallel. It is useful when you need to speed up training because you have a large amount of data, or work with a large batch size that cannot fit into the memory of a single GPU. 

Let's revise some of our PT understanding via [code](#) 

([https://colab.research.google.com/github/PytorchLightning/lightning-tutorials/blob/publication/.notebooks/lightning\\_examples/cifar10-baseline.ipynb](https://colab.research.google.com/github/PytorchLightning/lightning-tutorials/blob/publication/.notebooks/lightning_examples/cifar10-baseline.ipynb)).

Fabrik



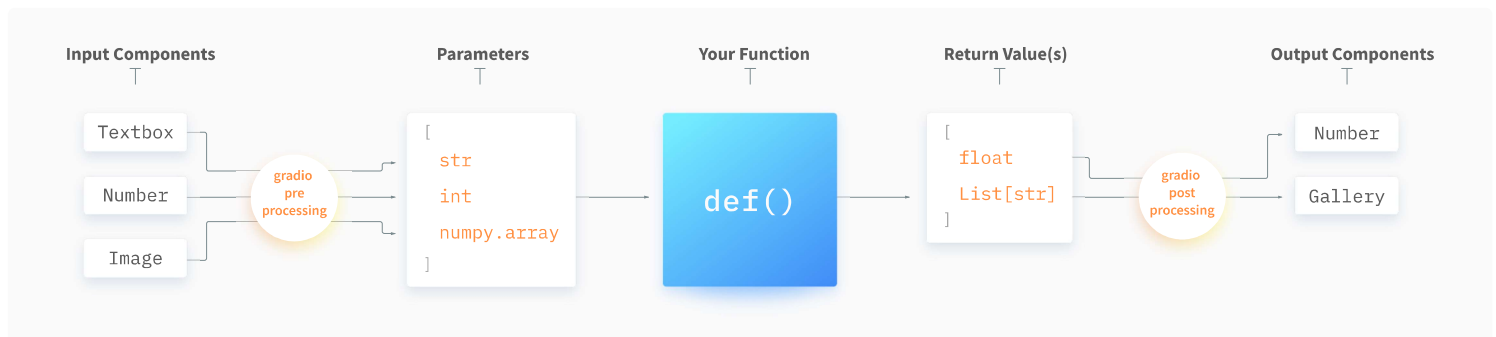
- Fast to implement: no need to restructure your code.
- Flexible: can write your own trainer or inference logic
- Control: Everything is opt-in.

At TSAI, we'll use Pytorch and Lightning, and skip Fabrik. However you're free to use Pytorch+ Fabrik AND Lightning for your assignments (basically integrate Fabrik if you want with your Pytroch Code, but Lightning assignments will be purely lightning based).

Let's check out some [docs](https://lightning.ai/docs/fabric/stable/) → [\(https://lightning.ai/docs/fabric/stable/\)](https://lightning.ai/docs/fabric/stable/).

## Gradio

It is the fastest way to demo your models with a friendly web interface so anyone can use it!

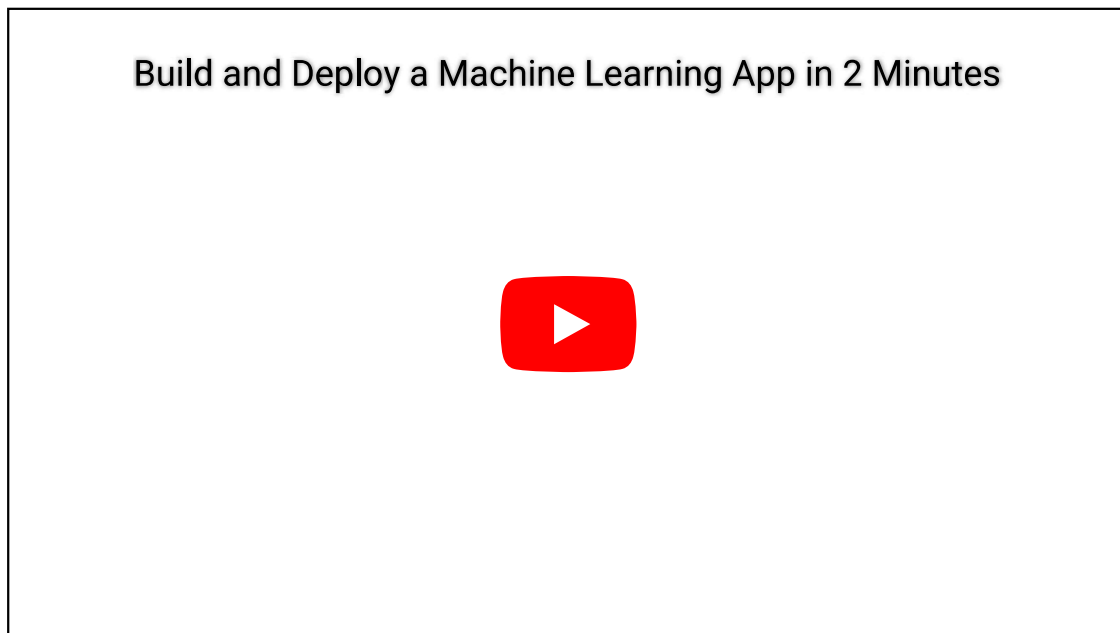


Let's just directly head over to the [documentation](https://www.gradio.app/guides/quickstart) →  [\(https://www.gradio.app/guides/quickstart\)](https://www.gradio.app/guides/quickstart).

Let's go back to our code and write some gradio code.

## Huggingface Spaces

Spaces offer a simple way to host ML demo apps directly on your profile (for free).



Here's the [documentation](https://huggingface.co/docs/hub/spaces-overview)  [\\_\(https://huggingface.co/docs/hub/spaces-overview\)\\_](https://huggingface.co/docs/hub/spaces-overview).

Let's build an [app](https://huggingface.co/spaces/theschoolofai/demo)  [\\_\(https://huggingface.co/spaces/theschoolofai/demo\)\\_](https://huggingface.co/spaces/theschoolofai/demo), it's simple.

[S11 Reference](https://canvas.instructure.com/courses/8491182/files/252602488?wrap=1) [\\_\(https://canvas.instructure.com/courses/8491182/files/252602488?wrap=1\)\\_](https://canvas.instructure.com/courses/8491182/files/252602488?wrap=1)   
[\\_\(https://canvas.instructure.com/courses/8491182/files/252602488/download?download\\_frd=1\)](https://canvas.instructure.com/courses/8491182/files/252602488/download?download_frd=1)

[S11 Other files \(extract files\)](https://canvas.instructure.com/courses/8491182/files/252602732?wrap=1) [\\_\(https://canvas.instructure.com/courses/8491182/files/252602732?wrap=1\)\\_](https://canvas.instructure.com/courses/8491182/files/252602732?wrap=1)  
 [\\_\(https://canvas.instructure.com/courses/8491182/files/252602732/download?download\\_frd=1\)](https://canvas.instructure.com/courses/8491182/files/252602732/download?download_frd=1)

CLASS FILES:

[Hugging Face Code and Demo](https://huggingface.co/spaces/theschoolofai/erfav2demo)  [\\_\(https://huggingface.co/spaces/theschoolofai/erfav2demo\)\\_](https://huggingface.co/spaces/theschoolofai/erfav2demo)

## Assignment

1. Move the S11 reference assignment code shared above to Lightning first and then to Spaces such that:
  1. (You have retrained your model on Lightning)
  2. You are using Gradio
  3. Your spaces app has these features:
    1. ask the user whether he/she wants to see GradCAM images and how many, and from which layer, allow opacity change as well
    2. ask whether he/she wants to view misclassified images, and how many
    3. allow users to upload new images, as well as provide 10 example images
    4. ask how many top classes are to be shown (make sure the user cannot enter more than 10)
  4. Add the full details on what your App is doing to Spaces README
2. Head over to submissions and then:
  1. Submit the Spaces App Link
  2. Submit the Spaces README link (Space must not have a training code)
  3. Submit the GitHub Link where Lightning Code can be found along with detailed README with log, loss function graphs, and 10 misclassified images

## Videos

## Studio

## ERA V2 S13 Studio



## Google Meet

### ERA V2 Session 13 GM

