# Image Classification Using PyTorch Lightning and Weights & Biases

This article provides a practical introduction on how to use PyTorch Lightning to improve the readability and reproducibility of your PyTorch code.

Ayush Thakur

Last Updated: Jan 6, 2023

In this article, we'll build an image classification pipeline using PyTorch Lightning. We will follow this style guide to increase the readability and reproducibility of our code.

Open in Colab

## Table of Contents

:ted

## ⚡What is PyTorch Lightning?

PyTorch is an extremely powerful framework for your deep learning research. But once the research gets complicated and things like 16-bit precision, multi-GPU training, and TPU training get mixed in, users are likely to introduce bugs. **PyTorch Lightning lets you decouple research from engineering.**

Let's build an image classification pipeline using PyTorch Lightning. Think this to be a starting guide to getting familiar with the nuisances of PyTorch Lightning.

**PyTorch Lightning ⚡ is not another framework but a style guide for PyTorch.**

## ⏳ Installation and Imports

For this tutorial, we need PyTorch Lightning(ain't that obvious!) and Weights & Biases.

```
# install pytorch lighting
! pip install pytorch-lightning --quiet
# install weights and biases
!pip install wandb --quiet
```

Besides your regular PyTorch imports, you need these ⚡ imports.

```
import pytorch_lightning as pl
```

```
# your favorite machine learning tracking tool
from pytorch_lightning.loggers import WandbLogger
```

We'll use WandbLogger to track our experiment results and log them directly to W&B.

## 🔧 DataModule - The Data Pipeline we Deserve

DataModules are a way of decoupling data-related hooks from the `LightningModule` so you can develop dataset-agnostic models.

It organizes the data pipeline into one shareable and reusable class. A data module encapsulates the five steps involved in data processing in PyTorch:

- Download / tokenize / process.
- Clean and (maybe) save to disk.
- Load inside Dataset.
- Apply transforms (rotate, tokenize, etc…).
- Wrap inside a DataLoader.

Learn more about datamodules here. Let's build a datamodule for the CIFAR-10 dataset.

### 1. Init

The `CIFAR10DataModule` subclasses from PyTorch Lightning's `LightningDataModule`. We will pass in the hyperparameters required for our data pipeline using the `__init__` method. We will also define the data transform pipeline here.

```
class CIFAR10DataModule(pl.LightningDataModule):
    def __init__(self, batch_size, data_dir: str = './'):
        super().__init__()
        self.data_dir = data_dir
        self.batch_size = batch_size

        self.transform = transforms.Compose([
            transforms.ToTensor(),
            transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))
        ])

        self.dims = (3, 32, 32)
        self.num_classes = 10
```

## 2. Perpare_data

This is where we will define the logic to download our dataset. We are using torchvision's CIFAR10 dataset class to download. Use this method to do things that might write to disk or that need to be done only from a single GPU in distributed settings. Do not make any state assignments in this function (i.e. `self.something = ...`).

```
    def prepare_data(self):
        # download
        CIFAR10(self.data_dir, train=True, download=True)
        CIFAR10(self.data_dir, train=False, download=True)
```

## 3. Setup_data

This is where we will load in data from the file and prepare PyTorch tensor datasets for each split. the data split is thus reproducible. This method expects a `stage` arg which is used to separate logic for 'fit' and 'test'. This is helpful if we don't want to load the entire dataset at once. The data operations that we want to perform on every GPU are defined here. This includes applying transform to the PyTorch tensor dataset.

```
    def setup(self, stage=None):
        # Assign train/val datasets for use in dataloaders
        if stage == 'fit' or stage is None:
            cifar_full = CIFAR10(self.data_dir, train=True, transfo
            self.cifar_train, self.cifar_val = random_split(cifar_fu

        # Assign test dataset for use in dataloader(s)
        if stage == 'test' or stage is None:
            self.cifar_test = CIFAR10(self.data_dir, train=False, tr
```

### 4. X_dataloader

`train_dataloader()`, `val_dataloader()`, and `test_dataloader()` all return PyTorch `DataLoader` instances that are created by wrapping their respective datasets that we prepared in `setup()`

```
    def train_dataloader(self):
        return DataLoader(self.cifar_train, batch_size=self.batch_s

    def val_dataloader(self):
        return DataLoader(self.cifar_val, batch_size=self.batch_size

    def test_dataloader(self):
        return DataLoader(self.cifar_test, batch_size=self.batch_si
```

Open in Colab

## 📱 Callbacks

A callback is a self-contained program that can be reused across projects. PyTorch Lightning comes with a few built-in callbacks which are regularly used.

Learn more about callbacks in PyTorch Lightning here.

### Built-In Callback

In this tutorial, we will use Early Stopping and Model Checkpoint built-in callbacks. They can be passed to the `Trainer`.

### Custom Callback

If you are familiar with Custom Keras callback, the ability to do the same in your PyTorch pipeline is just a cherry on the cake.

Since we are performing image classification, the ability to visualize the model's predictions on some samples of images can be helpful.

This in the form of a callback can help debug the model at an early stage.

## 1.__Init__

The `ImagePredictionLogger` subclasses from the PyTorch Lightning's `Callback` class. Here we will pass `val_samples` which is a tuple of images and labels. The `num_samples` is the number of images to be logged to the W&B dashboard.

```
class ImagePredictionLogger(Callback):
    def __init__(self, val_samples, num_samples=32):
        super().__init__()
        self.num_samples = num_samples
        self.val_imgs, self.val_labels = val_samples
```

## 2. The Callback Hooks

You can find all the available callback hooks here.

The `on_validation_epoch_end` method is called when the validation epoch ends. It takes two arguments - `trainer` and `pl_module` which are automatically passed by the `Trainer`.

By using `trainer.logger.experimental` we can use all the features available by Weights & Biases.

```
    def on_validation_epoch_end(self, trainer, pl_module):
        # Bring the tensors to CPU
        val_imgs = self.val_imgs.to(device=pl_module.device)
        val_labels = self.val_labels.to(device=pl_module.device)
        # Get model prediction
        logits = pl_module(val_imgs)
        preds = torch.argmax(logits, -1)
        # Log the images as wandb Image
        trainer.logger.experiment.log({
            "examples":[wandb.Image(x, caption=f"Pred:{pred}, Label
                            for x, pred, y in zip(val_imgs[:self.num_
                                                  preds[:self.num_sar
                                                  val_labels[:self.nu
        })
```

Open in Colab

We will see see the results of this callback.

## LightningModule - Define the System

The `LightningModule` defines a system and not a model. Here a system groups all the research code into a single class to make it self-contained. `LightningModule` organizes your PyTorch code into 5 sections:

- Computations (`__init__`).
- Train loop (`training_step`)
- Validation loop (`validation_step`)
- Test loop (`test_step`)
- Optimizers (`configure_optimizers`)

One can thus build a dataset agnostic model that can be easily shared. Let's build a system for Cifar-10 classification.

### 1. Computations

This component of the `LightningModule` encompasses the model architecture and the forward pass. This code snippet might look familiar to your normal PyTorch code.

You can pass all the required hyperparameters required by the model through `__init__`. Often times we train many versions of a model with different hyperparameters. By calling `save_hyperparameters` we can ask lightning to save the values of anything in the `__init__` for us to the checkpoint. This is a useful feature.

You will notice two methods `_get_conv_output` and `_forward_features`. They are used to automatically compute the tensor size of the output of the convolutional block. Learn about it here.

The `forward` method might look familiar to the normal PyTorch code. However, in Lightning `forward` is used only to define the inference actions. The `training_step` defines the training loop.

```
class LitModel(pl.LightningModule):
    def __init__(self, input_shape, num_classes, learning_rate=2e-4)
        super().__init__()

        # log hyperparameters
        self.save_hyperparameters()
        self.learning_rate = learning_rate

        self.conv1 = nn.Conv2d(3, 32, 3, 1)
        self.conv2 = nn.Conv2d(32, 32, 3, 1)
        self.conv3 = nn.Conv2d(32, 64, 3, 1)
        self.conv4 = nn.Conv2d(64, 64, 3, 1)

        self.pool1 = torch.nn.MaxPool2d(2)
        self.pool2 = torch.nn.MaxPool2d(2)

        n_sizes = self._get_conv_output(input_shape)

        self.fc1 = nn.Linear(n_sizes, 512)
        self.fc2 = nn.Linear(512, 128)
        self.fc3 = nn.Linear(128, num_classes)

        self.accuracy = torchmetrics.Accuracy()

    # returns the size of the output tensor going into Linear layer
    def _get_conv_output(self, shape):
        batch_size = 1
        input = torch.autograd.Variable(torch.rand(batch_size, *shap

        output_feat = self._forward_features(input)
        n_size = output_feat.data.view(batch_size, -1).size(1)
        return n_size
    # returns the feature tensor from the conv block
    def _forward_features(self, x):
        x = F.relu(self.conv1(x))
        x = self.pool1(F.relu(self.conv2(x)))
        x = F.relu(self.conv3(x))
        x = self.pool2(F.relu(self.conv4(x)))
        return x

    # will be used during inference
    def forward(self, x):
       x = self._forward_features(x)
       x = x.view(x.size(0), -1)
       x = F.relu(self.fc1(x))
       x = F.relu(self.fc2(x))
       x = F.log_softmax(self.fc3(x), dim=1)

       return x
```

Open in Colab

## 2. Training Loop

Lightning automates most of the training for us, the epoch and batch iterations, all we need to keep is the training step logic. The `training_step` method requires `batch` and `batch_idx` args which are automatically passed by the `Trainer`. Learn more about training loop here

To compute epoch wise metrics pass `on_epoch=True` to the `.log` method. The step-wise metrics are automatically logged. To turn it off pass `on_step=False`.

```
    def training_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = F.nll_loss(logits, y)

        # training metrics
        preds = torch.argmax(logits, dim=1)
        acc = self.accuracy(preds, y)
        self.log('train_loss', loss, on_step=True, on_epoch=True, lo
        self.log('train_acc', acc, on_step=True, on_epoch=True, logg

        return loss
```

## 3. Validation Loop

Similar to the training loop, the validation loop can be implemented by overwriting the `validation_step` method of the `LightningModule`. Learn about the validation loop here.

The metrics are automatically logged epoch-wise.

```
    def validation_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = F.nll_loss(logits, y)

        # validation metrics
        preds = torch.argmax(logits, dim=1)
        acc = self.accuracy(preds, y)
        self.log('val_loss', loss, prog_bar=True)
        self.log('val_acc', acc, prog_bar=True)
        return loss
```

## 4. Test Loop

The test loop is similar to the validation loop. The only difference is that the test loop is only called when `trainer.test()` is used. Learn about the testing loop here.

The metrics are automatically logged epoch-wise.

```
    def test_step(self, batch, batch_idx):
        x, y = batch
        logits = self(x)
        loss = F.nll_loss(logits, y)

        # validation metrics
        preds = torch.argmax(logits, dim=1)
        acc = self.accuracy(preds, y)
        self.log('test_loss', loss, prog_bar=True)
        self.log('test_acc', acc, prog_bar=True)
        return loss
```

### 5. Optimizer

We can define our optimizer and learning rate schedulers using the `configure_optimizer` method. One can even define multiple optimizers like in the case of GANs.

Learn more about this method here.

```
    def configure_optimizers(self):
        optimizer = torch.optim.Adam(self.parameters(), lr=self.lea
        return optimizer
```

**Note:** If you are refactoring your PyTorch code using Lightning remove `.cuda()` and `.to()` from the `LightningModule`.

## 🚋 Train and Evaluate

Now that we have organized our data pipeline using `DataModule` and model architecture+training loop using `LightningModule`, the PyTorch Lightning Trainer automates everything else for us.

The Trainer automates:

- Epoch and batch iteration
- Calling of `optimizer.step()`, `backward`, `zero_grad()`
- Calling of `.eval()`, enabling/disabling grads
- Saving and loading weights
- Weights & Biases logging
- Multi-GPU training support
- TPU support
- 16-bit training support

Learn more about Trainer here. Let's use this to finally train our model.

We will first initialize our data pipeline. The Trainer just needs a PyTorch `DataLoader` for the train/val/test splits. We can directly pass the `dm` object that we have created to the Trainer. But since we need some samples for our `ImagePredictionLogger`, we will manually call the `prepare_data` and `setup` methods.

```
# Init our data pipeline
dm = CIFAR10DataModule(batch_size=32)
# To access the x_dataloader we need to call prepare_data and setup.
dm.prepare_data()
dm.setup()

# Samples required by the custom ImagePredictionLogger callback to l
val_samples = next(iter(dm.val_dataloader()))
val_imgs, val_labels = val_samples[0], val_samples[1]
val_imgs.shape, val_labels.shape
```

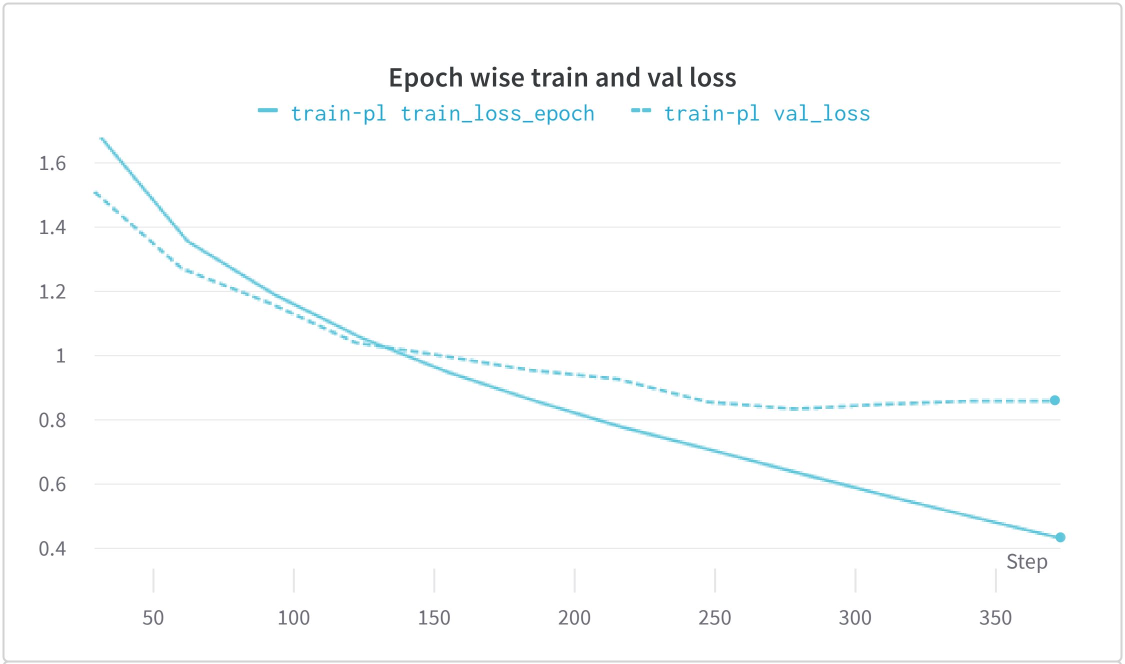
Training the model was never this easy. We just need to initialize the model and our favorite logger. Notice that we have passed `checkpoint_callback` separately.

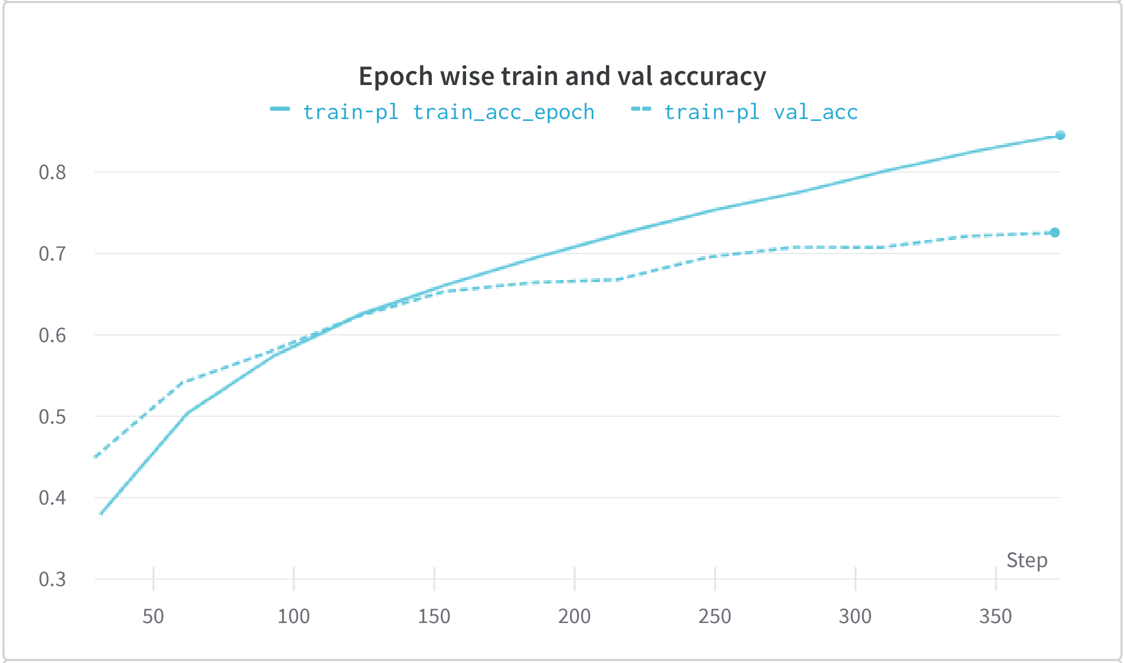
```
# Init our model
model = LitModel(dm.size(), dm.num_classes)

# Initialize wandb logger
wandb_logger = WandbLogger(project='wandb-lightning', job_type='trai

# Initialize a trainer
trainer = pl.Trainer(max_epochs=50,
                     progress_bar_refresh_rate=20,
                     gpus=1,
                     logger=wandb_logger,
                     callbacks=[early_stop_callback,
                                ImagePredictionLogger(val_samples)],
                     checkpoint_callback=checkpoint_callback)

# Train the model ⚡🚅⚡
trainer.fit(model, dm)
```

```
# Evaluate the model on the held-out test set ⚡⚡
trainer.test()

# Close wandb run
wandb.finish()
```

Open in Colab

The media panels below show the metrics that are logged to W&B.

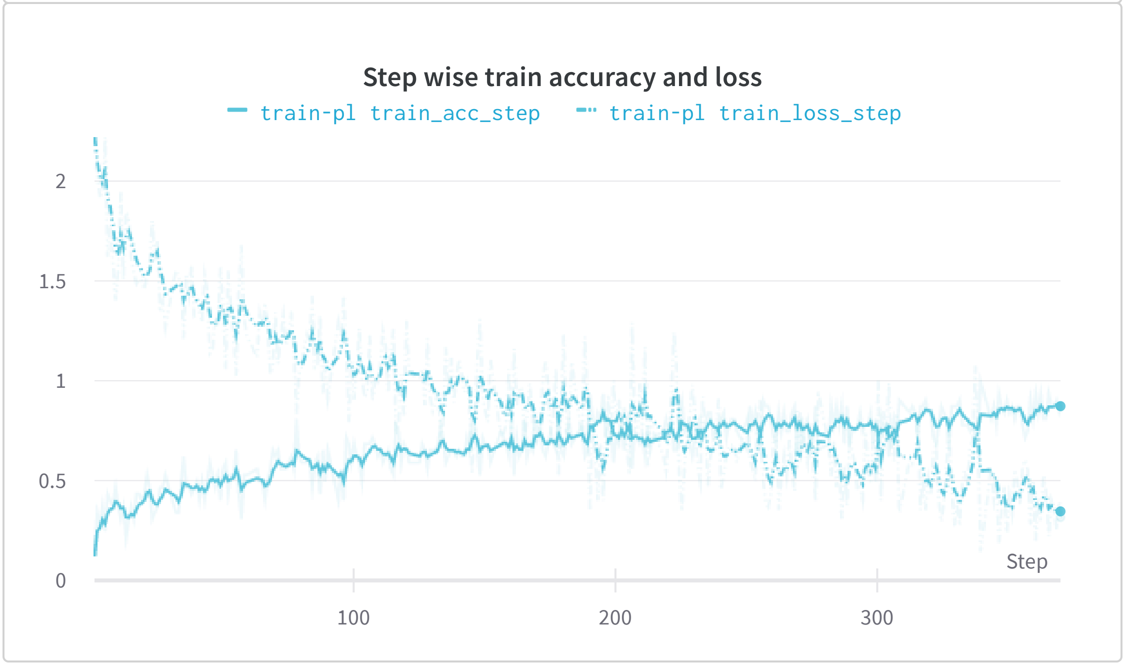




Test Accuracy

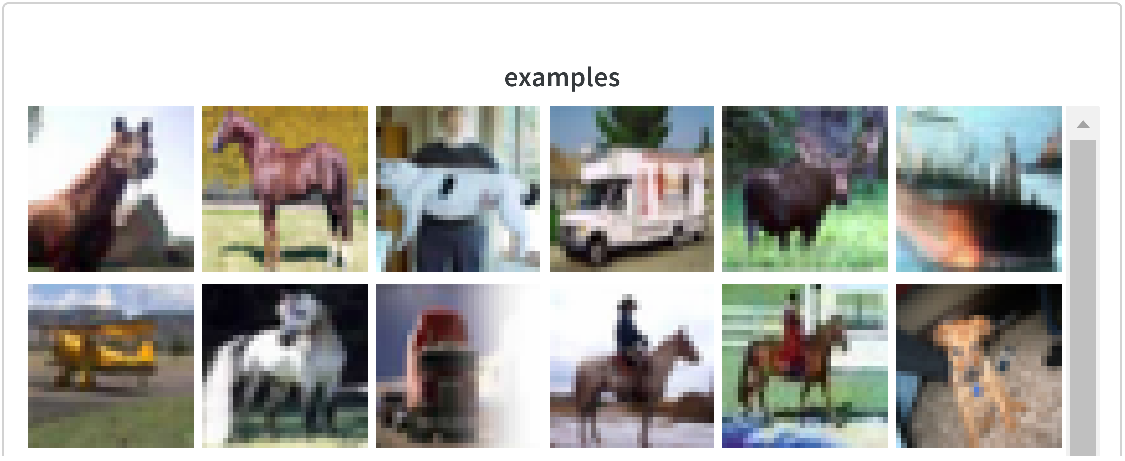
train-pl

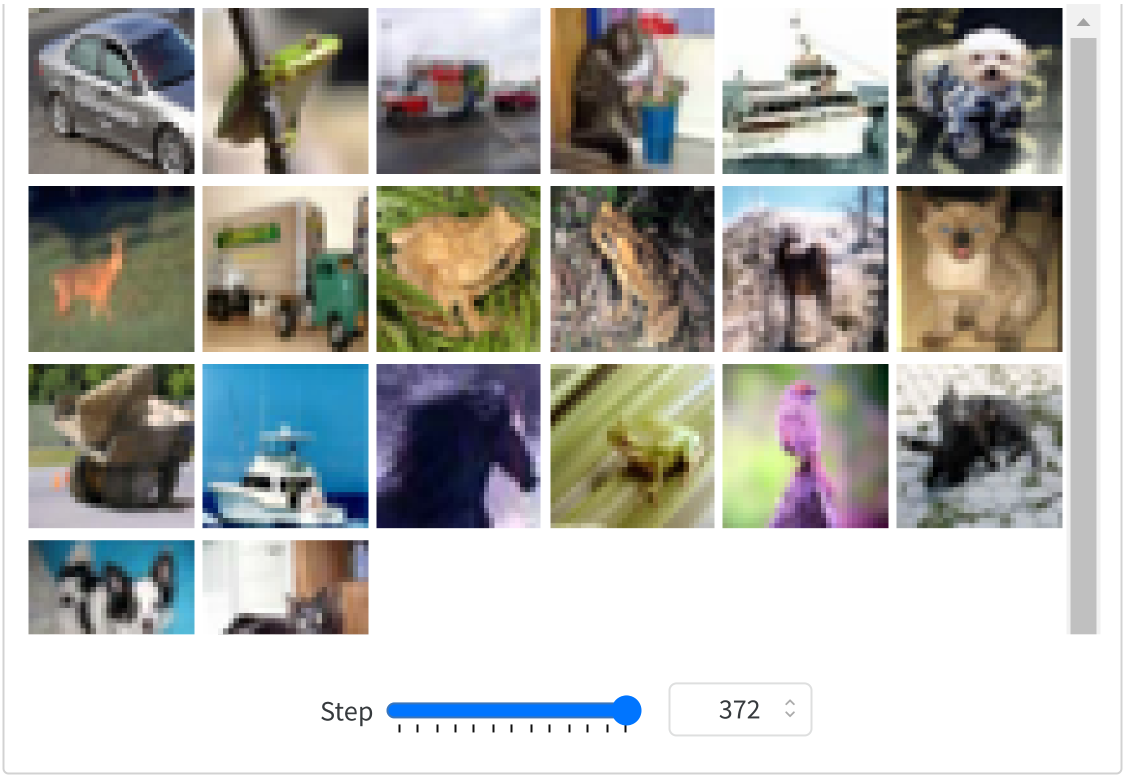
0.7097



The media chart below is the result of the `ImagePredictionLogger` custom callback. You can see the prediction and the ground truth label of each image.

Click on the ⚙ icon and move the slider to look at the predictions of the model at every epoch. 🔥





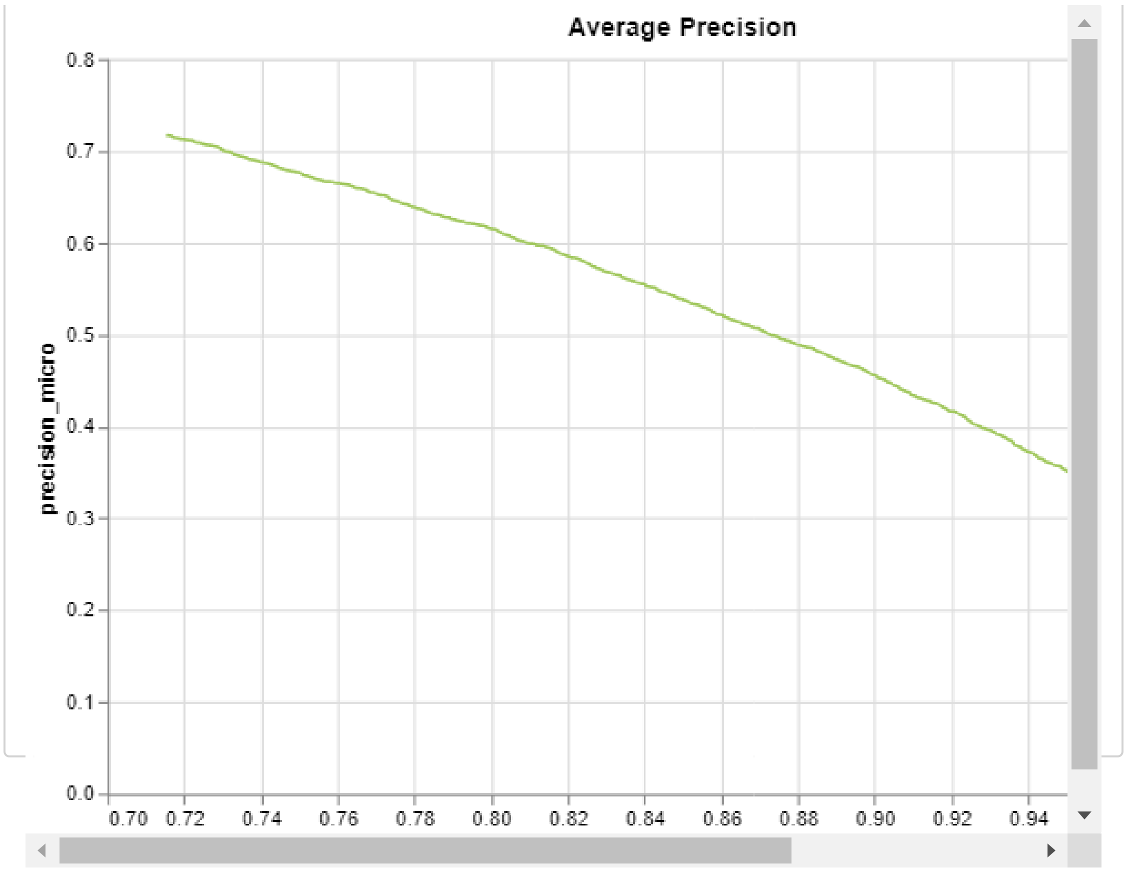
## 📉 Precision-Recall Curve

An image classification model needs to be tested thoroughly. The use of the precision-recall curve is standard practice.

Weights & Biases support custom vega plots using which one can plot literally anything that's supported by Vega. Let's look at the model's performance using the average precision-recall curve.

Check out this report to learn more about custom visualization support by Weights & Biases. Check out this report to learn how to log an average precision-recall curve.

Even though our test accuracy is ~70% there is a lot one can do to improve this classifier.



## Final Thoughts

I come from the TensorFlow/Keras ecosystem and find PyTorch a bit overwhelming even though it's an elegant framework. Just my personal experience though. While exploring PyTorch Lightning, I realized that almost all of the reasons that kept me away from PyTorch is taken care of. Here's a quick summary of my excitement:

- **Then**: Conventional PyTorch model definition used to be all over the place. With the model in some `model.py` script and the training loop in the `train.py` file. It was a lot of looking back and forth to understand the pipeline.
- **Now**: The `LightningModule` acts as a system where the model is defined along with the `training_step`, `validation_step`, etc. Now it's modular and shareable.
- **Then**: The best part about TensorFlow/Keras is the input data pipeline. Their dataset catalog is rich and growing. PyTorch's data pipeline used to be the biggest pain point. In normal PyTorch code,

the data download/cleaning/preparation is usually scattered across many files.

- **Now**: The `DataModule` organizes the data pipeline into one shareable and reusable class. It's simply a collection of a train_dataloader, val_dataloader(s), test_dataloader(s) along with the matching transforms and data processing/downloads steps required.
- **Then**: With Keras, one can call `model.fit` to train the model and `model.predict` to run inference on. `model.evaluate` offered a good old simple evaluation on the test data. This is not the case with PyTorch. One will usually find separate `train.py` and `test.py` files.
- **Now**: With the `LightningModule` in place, the `Trainer` automates everything. One needs to just call `trainer.fit` and `trainer.test` to train and evaluate the model.
- **Then**: TensorFlow loves TPU, PyTorch...well!
- **Now**: With PyTorch Lightning, it's so easy to train the same model with multiple GPUs and even on TPU. Wow!
- **Then**: I am a big fan of Callbacks and prefer writing custom callbacks. Something as trivial as Early Stopping used to be a point of discussion with conventional PyTorch.
- **Now**: With PyTorch Lightning using Early Stopping and Model Checkpointing is a piece of cake. I can even write custom callbacks.

I can probably keep going with my rant in the name of excitement. Here's a list of everything PyTorch Lightning has to offer.

## 🎨 Conclusion and Resources

I hope you find this report helpful. I will encourage to play with the code and train an image classifier with a dataset of your choice.

Here are some resources to learn more about PyTorch Lightning:

- From PyTorch to PyTorch Lightning — A gentle introduction by William Falcon who is one of the main creators of this library.
- Step-by-step walk-through - This is one of the official tutorials. Their documentation is really well written and I highly encourage it as a

good learning resource.
- Use Pytorch Lightning with Weights & Biases

Let me know your thoughts in the comments down below. 😄

Tags: Computer Vision, Classification, PyTorch Lightning, Experiment, Plots

Created with ❤️ on Weights & Biases.

https://wandb.ai/wandb/wandb-lightning/reports/Image-Classification-using-PyTorch-Lightning--VmlldzoyODk1NzY

;