Visualize predictions

Try in a Colab Notebook here →

This covers how to track, visualize, and compare model predictions over the course of training, using PyTorch on MNIST data.

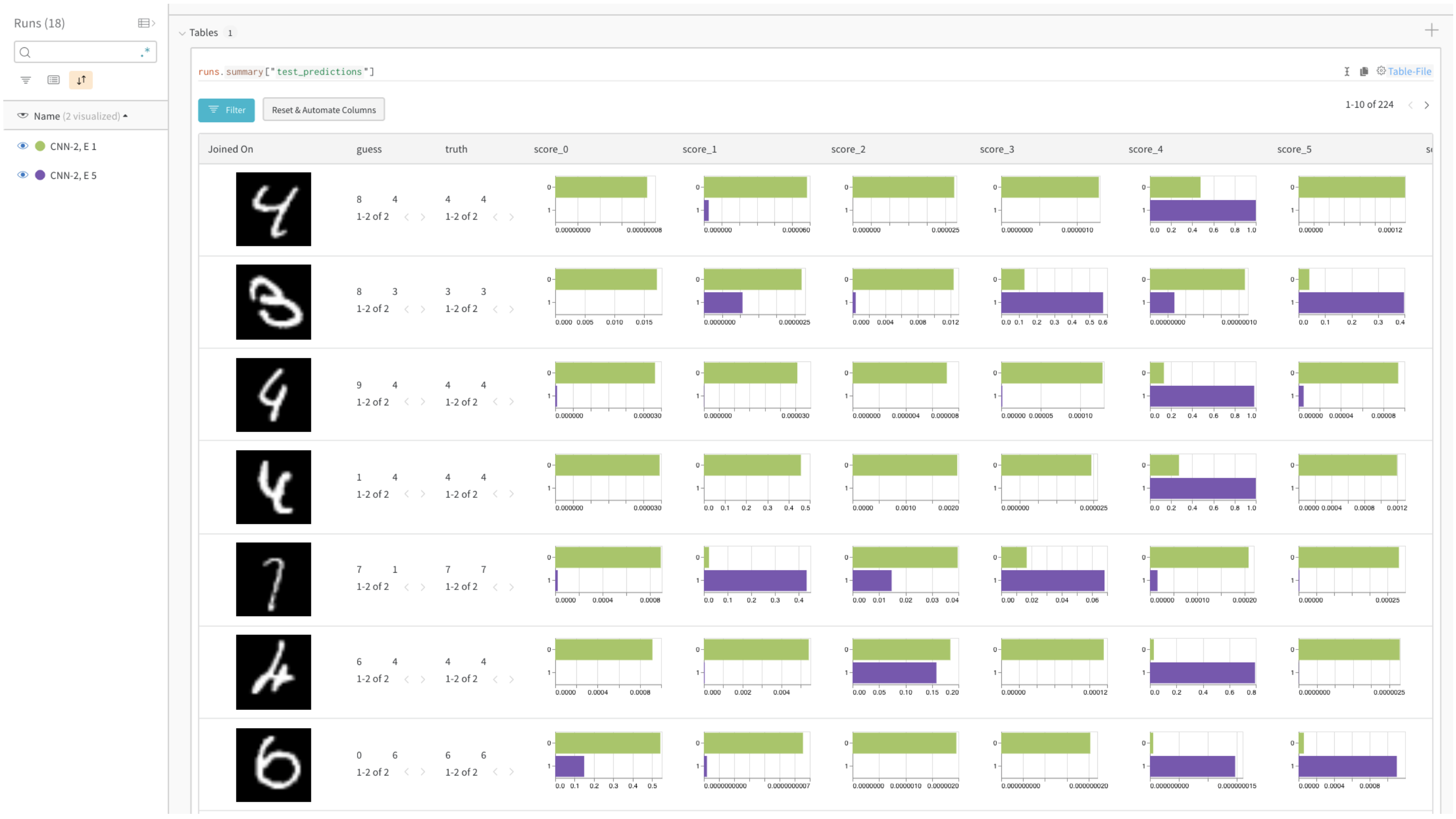
You will learn how to:

1. Log metrics, images, text, etc. to a `wandb.Table()` during model training or evaluation
2. View, sort, filter, group, join, interactively query, and explore these tables
3. Compare model predictions or results: dynamically across specific images, hyperparameters/model versions, or time steps.

Examples

Compare predicted scores for specific images

Live example: compare predictions after 1 vs 5 epochs of training →



```
runs.summary["test_predictions"]
```

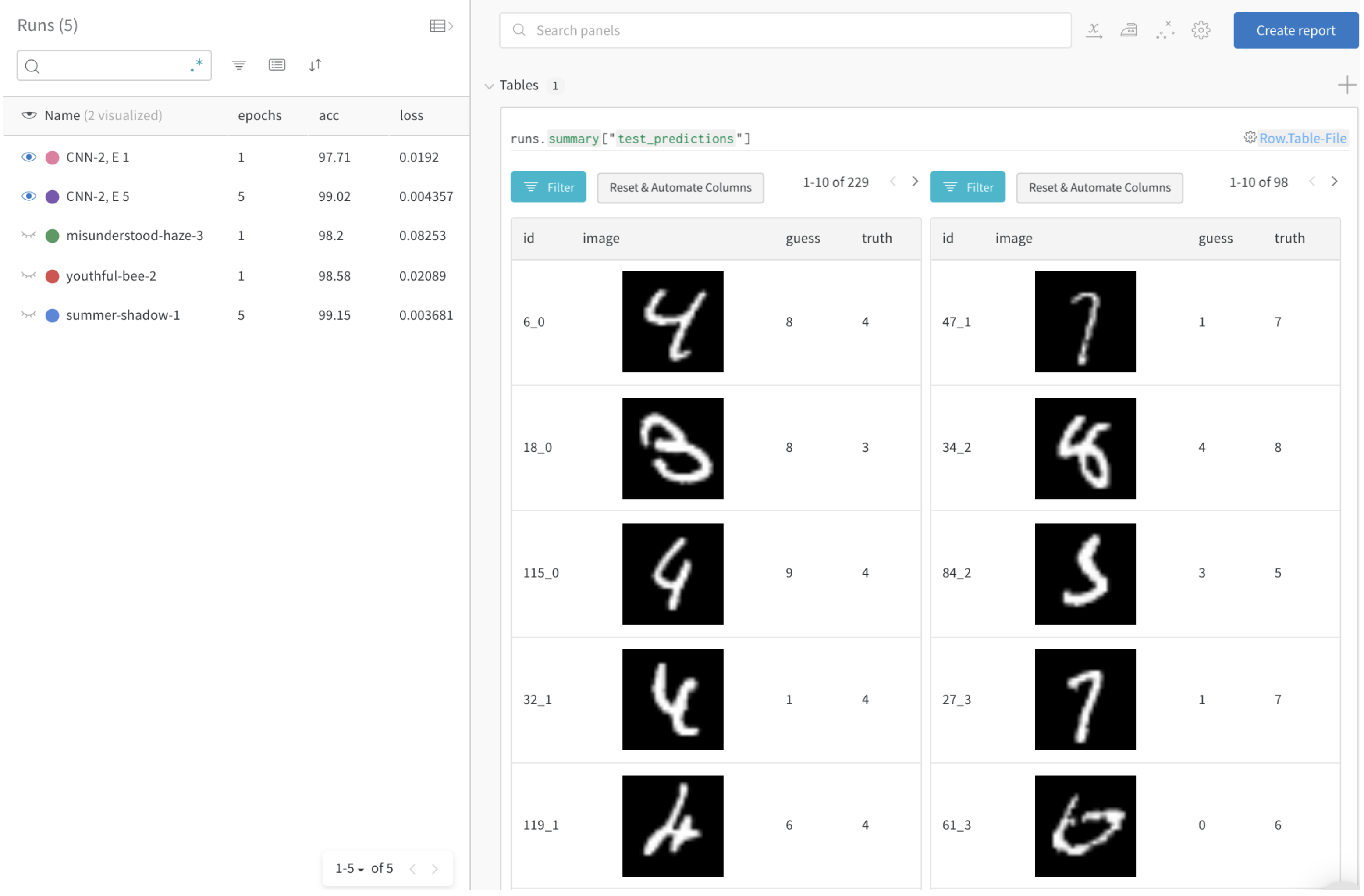
Joined On	guess	truth	score_0	score_1	score_2	score_3	score_4	score_5
	8 4 (1-2 of 2)	4 4 (1-2 of 2)						
	8 3 (1-2 of 2)	3 3 (1-2 of 2)						
	9 4 (1-2 of 2)	4 4 (1-2 of 2)						
	1 4 (1-2 of 2)	4 4 (1-2 of 2)						
	7 1 (1-2 of 2)	7 7 (1-2 of 2)						
	6 4 (1-2 of 2)	4 4 (1-2 of 2)						
	0 6 (1-2 of 2)	6 6 (1-2 of 2)						

The histograms compare per-class scores between the two models. The top green bar in each histogram represents model "CNN-2, 1 epoch" (id 0), which only trained for 1 epoch. The bottom purple bar represents model "CNN-2, 5 epochs" (id 1), which trained for 5 epochs. The images are filtered to cases where the models disagree. For example, in the first row, the "4" gets high scores across all the possible digits after 1 epoch, but after 5 epochs it scores highest on the correct label and very low on the rest.

Focus on top errors over time

Live example →

See incorrect predictions (filter to rows where "guess" != "truth") on the full test data. Note that there are 229 wrong guesses after 1 training epoch, but only 98 after 5 epochs.



Runs (5)

Name (2 visualized)	epochs	acc	loss
CNN-2, E 1	1	97.71	0.0192
CNN-2, E 5	5	99.02	0.004357
misunderstood-haze-3	1	98.2	0.08253
youthful-bee-2	1	98.58	0.02089
summer-shadow-1	5	99.15	0.003681

1-5 of 5

Search panels

Create report

Tables 1

```
runs.summary["test_predictions"]
```

Row.Table-File

Filter | Reset & Automate Columns | 1-10 of 229

id	image	guess	truth
6_0		8	4
18_0		8	3
115_0		9	4
32_1		1	4
119_1		6	4

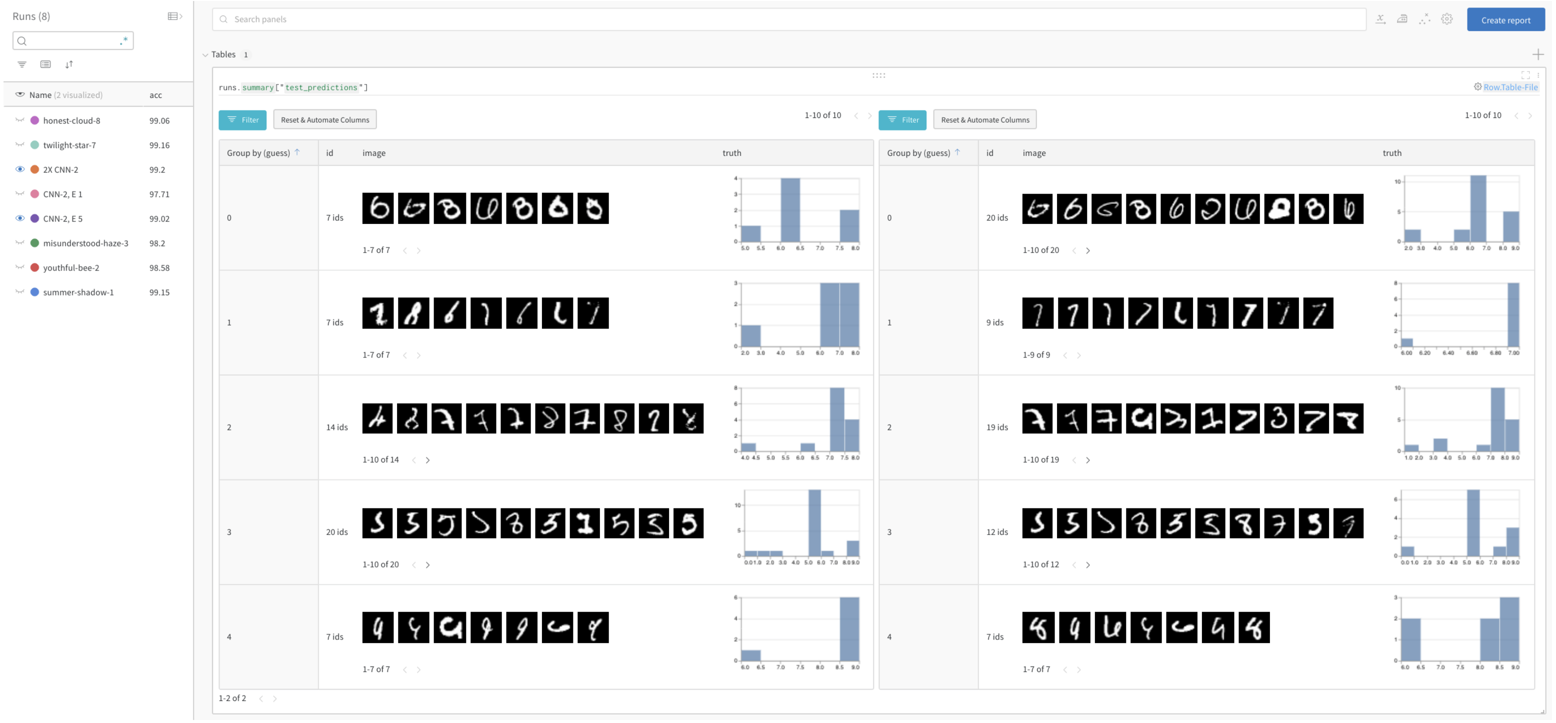
Filter | Reset & Automate Columns | 1-10 of 98

id	image	guess	truth
47_1		1	7
34_2		4	8
84_2		3	5
27_3		1	7
61_3		0	6

Compare model performance and find patterns

See full detail in a live example ⟶

Filter out correct answers, then group by the guess to see examples of misclassified images and the underlying distribution of true labels—for two models side-by-side. A model variant with 2X the layer sizes and learning rate is on the left, and the baseline is on the right. Note that the baseline makes slightly more mistakes for each guessed class.



Runs (8)

Name (2 visualized)	acc
honest-cloud-8	99.06
twilight-star-7	99.16
2X CNN-2	99.2
CNN-2, E 1	97.71
CNN-2, E 5	99.02
misunderstood-haze-3	98.2
youthful-bee-2	98.58
summer-shadow-1	99.15

Search panels

Create report

Tables 1

```
runs.summary["test_predictions"]
```

Row.Table-File

Filter | Reset & Automate Columns | 1-10 of 10

Group by (guess)	id	image	truth
0	7 ids	1-7 of 7	
1	7 ids	1-7 of 7	
2	14 ids	1-10 of 14	
3	20 ids	1-10 of 20	
4	7 ids	1-7 of 7	

Filter | Reset & Automate Columns | 1-10 of 10

Group by (guess)	id	image	truth
0	20 ids	1-10 of 20	
1	9 ids	1-9 of 9	
2	19 ids	1-10 of 19	
3	12 ids	1-10 of 12	
4	7 ids	1-7 of 7	

1-2 of 2

Sign up or login

Sign up or login to W&B to see and interact with your experiments in the browser.

In this example we're using Google Colab as a convenient hosted environment, but you can run your own training scripts from anywhere and visualize metrics with W&B's experiment tracking tool.

```
!pip install wandb -qqq
```

log to your account

```

import wandb
wandb.login()

WANDB_PROJECT = "mnist-viz"
```

0. Setup

Install dependencies, download MNIST, and create train and test datasets using PyTorch.

```
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as T
import torch.nn.functional as F


device = "cuda:0" if torch.cuda.is_available() else "cpu"

# create train and test dataloaders
def get_dataloader(is_train, batch_size, slice=5):
    "Get a training dataloader"
    ds = torchvision.datasets.MNIST(root=".", train=is_train, transform=T.ToTensor(), download=True)
    loader = torch.utils.data.DataLoader(dataset=ds,
                                         batch_size=batch_size,
                                         shuffle=True if is_train else False,
                                         pin_memory=True, num_workers=2)
    return loader
```

1. Define the model and training schedule

- Set the number of epochs to run, where each epoch consists of a training step and a validation (test) step. Optionally configure the amount of data to log per test step. Here the number of batches and number of images per batch to visualize are set low to simplify the demo.
- Define a simple convolutional neural net (following pytorch-tutorial code).
- Load in train and test sets using PyTorch

```
# Number of epochs to run
# Each epoch includes a training step and a test step, so this sets
# the number of tables of test predictions to log
EPOCHS = 1

# Number of batches to log from the test data for each test step
# (default set low to simplify demo)
NUM_BATCHES_TO_LOG = 10 #79

# Number of images to log per test batch
# (default set low to simplify demo)
NUM_IMAGES_PER_BATCH = 32 #128

# training configuration and hyperparameters
NUM_CLASSES = 10
BATCH_SIZE = 32
LEARNING_RATE = 0.001
L1_SIZE = 32
L2_SIZE = 64
# changing this may require changing the shape of adjacent layers
CONV_KERNEL_SIZE = 5

# define a two-layer convolutional neural network
class ConvNet(nn.Module):
    def __init__(self, num_classes=10):
        super(ConvNet, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Conv2d(1, L1_SIZE, CONV_KERNEL_SIZE, stride=1, padding=2),
            nn.BatchNorm2d(L1_SIZE),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.layer2 = nn.Sequential(
            nn.Conv2d(L1_SIZE, L2_SIZE, CONV_KERNEL_SIZE, stride=1, padding=2),
            nn.BatchNorm2d(L2_SIZE),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2))
        self.fc = nn.Linear(7*7*L2_SIZE, NUM_CLASSES)
        self.softmax = nn.Softmax(NUM_CLASSES)

    def forward(self, x):
        # uncomment to see the shape of a given layer:
        #print("x: ", x.size())
        out = self.layer1(x)
        out = self.layer2(out)
        out = out.reshape(out.size(0), -1)
        out = self.fc(out)
        return out

train_loader = get_dataloader(is_train=True, batch_size=BATCH_SIZE)
test_loader = get_dataloader(is_train=False, batch_size=2*BATCH_SIZE)

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

2. Run training and log test predictions

For every epoch, run a training step and a test step. For each test step, create a wandb.Table() in which to store test predictions. These can be visualized, dynamically queried, and compared side by side in your browser.

```
# ✨ W&B: Initialize a new run to track this model's training
wandb.init(project="table-quickstart")

# ✨ W&B: Log hyperparameters using config
cfg = wandb.config
cfg.update({"epochs" : EPOCHS, "batch_size": BATCH_SIZE, "lr" : LEARNING_RATE,
            "l1_size" : L1_SIZE, "l2_size": L2_SIZE,
            "conv_kernel" : CONV_KERNEL_SIZE,
            "img_count" : min(10000, NUM_IMAGES_PER_BATCH*NUM_BATCHES_TO_LOG)})

# define model, loss, and optimizer
model = ConvNet(NUM_CLASSES).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=LEARNING_RATE)

# convenience funtion to log predictions for a batch of test images
def log_test_predictions(images, labels, outputs, predicted, test_table, log_counter):
  # obtain confidence scores for all classes
  scores = F.softmax(outputs.data, dim=1)
  log_scores = scores.cpu().numpy()
  log_images = images.cpu().numpy()
  log_labels = labels.cpu().numpy()
  log_preds = predicted.cpu().numpy()
  # adding ids based on the order of the images
  _id = 0
  for i, l, p, s in zip(log_images, log_labels, log_preds, log_scores):
    # add required info to data table:
    # id, image pixels, model's guess, true label, scores for all classes
    img_id = str(_id) + "_" + str(log_counter)
    test_table.add_data(img_id, wandb.Image(i), p, l, *s)
    _id += 1
    if _id == NUM_IMAGES_PER_BATCH:
      break

# train the model
total_step = len(train_loader)
for epoch in range(EPOCHS):
    # training step
    for i, (images, labels) in enumerate(train_loader):
        images = images.to(device)
        labels = labels.to(device)
        # forward pass
        outputs = model(images)
        loss = criterion(outputs, labels)
        # backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        # ✨ W&B: Log loss over training steps, visualized in the UI live
        wandb.log({"loss" : loss})
        if (i+1) % 100 == 0:
            print ('Epoch [{}/{}], Step [{}/{}], Loss: {:.4f}'
                .format(epoch+1, EPOCHS, i+1, total_step, loss.item()))


    # ✨ W&B: Create a Table to store predictions for each test step
    columns=["id", "image", "guess", "truth"]
    for digit in range(10):
      columns.append("score_" + str(digit))
    test_table = wandb.Table(columns=columns)

    # test the model
    model.eval()
    log_counter = 0
    with torch.no_grad():
        correct = 0
        total = 0
        for images, labels in test_loader:
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            if log_counter < NUM_BATCHES_TO_LOG:
              log_test_predictions(images, labels, outputs, predicted, test_table, log_counter)
              log_counter += 1
            total += labels.size(0)
            correct += (predicted == labels).sum().item()

        acc = 100 * correct / total
        # ✨ W&B: Log accuracy across training epochs, to visualize in the UI
        wandb.log({"epoch" : epoch, "acc" : acc})
        print('Test Accuracy of the model on the 10000 test images: {} %'.format(acc))

    # ✨ W&B: Log predictions table to wandb
    wandb.log({"test_predictions" : test_table})

# ✨ W&B: Mark the run as complete (useful for multi-cell notebook)
wandb.finish()
```

What's next?

The next tutorial, you will learn how to optimize hyperparameters using W&B Sweeps:

👉 Optimize Hyperparameters

Keras Tables

Try in a Colab Notebook here →

Use Weights & Biases for machine learning experiment tracking, dataset versioning, and project collaboration.

Experiments	Reports	Artifacts	Tables	Sweeps	Launch	Models
Experiment tracking	Collaborative dashboards	Dataset and model versionning	Interactive data visualization	Hyperparameter optimization	ML workflow automation	Model lifecycle management

This colab notebook introduces the `WandbEvalCallback` which is an abstract callback that be inherited to build useful callbacks for model prediction visualization and dataset visualization. Refer to the 💫 `WandbEvalCallback` section for more details.

🌴 Setup and Installation

First, let us install the latest version of Weights and Biases. We will then authenticate this colab instance to use W&B.

```
pip install -qq -U wandb
```

```
import os
import numpy as np
import tensorflow as tf
from tensorflow.keras import layers
from tensorflow.keras import models
import tensorflow_datasets as tfds

# Weights and Biases related imports
import wandb
from wandb.keras import WandbMetricsLogger
from wandb.keras import WandbModelCheckpoint
from wandb.keras import WandbEvalCallback
```

If this is your first time using W&B or you are not logged in, the link that appears after running `wandb.login()` will take you to sign-up/login page. Signing up for a free account is as easy as a few clicks.

```
wandb.login()
```

🌳 Hyperparameters

Use of proper config system is a recommended best practice for reproducible machine learning. We can track the hyperparameters for every experiment using W&B. In this colab we will be using simple Python `dict` as our config system.

```
configs = dict(
    num_classes=10,
    shuffle_buffer=1024,
    batch_size=64,
    image_size=28,
    image_channels=1,
    earlystopping_patience=3,
    learning_rate=1e-3,
    epochs=10,
)
```

🍁 Dataset

In this colab, we will be using CIFAR100 dataset from TensorFlow Dataset catalog. We aim to build a simple image classification pipeline using TensorFlow/Keras.

```
train_ds, valid_ds = tfds.load("fashion_mnist", split=["train", "test"])
```

```
AUTOTUNE = tf.data.AUTOTUNE
```

```
def parse_data(example):
    # Get image
    image = example["image"]
    # image = tf.image.convert_image_dtype(image, dtype=tf.float32)

    # Get label
    label = example["label"]
    label = tf.one_hot(label, depth=configs["num_classes"])

    return image, label


def get_dataloader(ds, configs, dataloader_type="train"):
    dataloader = ds.map(parse_data, num_parallel_calls=AUTOTUNE)

    if dataloader_type=="train":
        dataloader = dataloader.shuffle(configs["shuffle_buffer"])

    dataloader = (
        dataloader
        .batch(configs["batch_size"])
        .prefetch(AUTOTUNE)
    )

    return dataloader
```

```
trainloader = get_dataloader(train_ds, configs)
validloader = get_dataloader(valid_ds, configs, dataloader_type="valid")
```

🎄 Model

```
def get_model(configs):
    backbone = tf.keras.applications.mobilenet_v2.MobileNetV2(
        weights="imagenet", include_top=False
    )
    backbone.trainable = False

    inputs = layers.Input(
        shape=(configs["image_size"], configs["image_size"], configs["image_channels"])
    )
    resize = layers.Resizing(32, 32)(inputs)
    neck = layers.Conv2D(3, (3, 3), padding="same")(resize)
    preprocess_input = tf.keras.applications.mobilenet.preprocess_input(neck)
    x = backbone(preprocess_input)
    x = layers.GlobalAveragePooling2D()(x)
    outputs = layers.Dense(configs["num_classes"], activation="softmax")(x)

    return models.Model(inputs=inputs, outputs=outputs)
```

```
tf.keras.backend.clear_session()
model = get_model(configs)
model.summary()
```

🌿 Compile Model

```
model.compile(
    optimizer="adam",
    loss="categorical_crossentropy",
    metrics=[
        "accuracy",
        tf.keras.metrics.TopKCategoricalAccuracy(k=5, name="top@5_accuracy"),
    ],
)
```

💫 `WandbEvalCallback`

The `WandbEvalCallback` is an abstract base class to build Keras callbacks for primarily model prediction visualization and secondarily dataset visualization.

This is a dataset and task agnostic abstract callback. To use this, inherit from this base callback class and implement the `add_ground_truth` and `add_model_prediction` methods.

The `WandbEvalCallback` is a utility class that provides helpful methods to:

- create data and prediction `wandb.Table` instances,
- log data and prediction Tables as `wandb.Artifact`,
- logs the data table `on_train_begin`,
- logs the prediction table `on_epoch_end`.

As an example, we have implemented `WandbClfEvalCallback` below for an image classification task. This example callback:

- logs the validation data (`data_table`) to W&B,
- performs inference and logs the prediction (`pred_table`) to W&B on every epoch end.

How the memory footprint is reduced?

We log the `data_table` to W&B when the `on_train_begin` method is ivoked. Once it's uploaded as a W&B Artifact, we get a reference to this table which can be accessed using `data_table_ref` class variable. The `data_table_ref` is a 2D list that can be indexed like `self.data_table_ref[idx][n]` where `idx` is the row number while `n` is the column number. Let's see the usage in the example below.

```
class WandbClfEvalCallback(WandbEvalCallback):
    def __init__(
        self, validloader, data_table_columns, pred_table_columns, num_samples=100
    ):
        super().__init__(data_table_columns, pred_table_columns)

        self.val_data = validloader.unbatch().take(num_samples)

    def add_ground_truth(self, logs=None):
        for idx, (image, label) in enumerate(self.val_data):
            self.data_table.add_data(idx, wandb.Image(image), np.argmax(label, axis=-1))

    def add_model_predictions(self, epoch, logs=None):
        # Get predictions
        preds = self._inference()
        table_idxs = self.data_table_ref.get_index()

        for idx in table_idxs:
            pred = preds[idx]
            self.pred_table.add_data(
                epoch,
                self.data_table_ref.data[idx][0],
                self.data_table_ref.data[idx][1],
                self.data_table_ref.data[idx][2],
                pred,
            )

    def _inference(self):
        preds = []
        for image, label in self.val_data:
            pred = self.model(tf.expand_dims(image, axis=0))
            argmax_pred = tf.argmax(pred, axis=-1).numpy()[0]
            preds.append(argmax_pred)

        return preds
```

🌻 Train

```
# Initialize a W&B run
run = wandb.init(project="intro-keras", config=configs)

# Train your model
model.fit(
    trainloader,
    epochs=configs["epochs"],
    validation_data=validloader,
    callbacks=[
        WandbMetricsLogger(log_freq=10),
        WandbClfEvalCallback(
            validloader,
            data_table_columns=["idx", "image", "ground_truth"],
            pred_table_columns=["epoch", "idx", "image", "ground_truth", "prediction"],
        ),  # Notice the use of WandbEvalCallback here
    ],
)

# Close the W&B run
run.finish()
```