```python
import tempfile
from pathlib import Path

import cv2
import einops
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tqdm
import yaml

from helpers.adam import Adam
from modules.siren_mlp import SirenMLP


def train(config_path: Path, use_last_checkpoint: bool):
    if config_path is None:
        config_path = Path("configs/image_fit_card_f.yaml")

    # HYPERPARAMETERS
    config = yaml.safe_load(config_path.read_text())
    refresh_rate = config["display"]["refresh_rate"]
    learning_patience = config["learning"]["learning_patience"]
    learning_rates = config["learning"]["learning_rates"]
    num_iters = config["learning"]["num_iters"]
    weight_decay = config["learning"]["weight_decay"]
    num_hidden_layers = config["siren"]["num_hidden_layers"]
    hidden_layer_width = config["siren"]["hidden_layer_width"]
    siren_resolution = config["siren"]["resolution"]
    image_path = config["data"]["image_path"]

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(0x43966E87BD57227011B5B03B58785EC1)
    tf.random.set_seed(0x43966E87BD57227011B5B03B58785EC1)

    siren = SirenMLP(
        2,
        3,
        num_hidden_layers=num_hidden_layers,
        hidden_layer_width=hidden_layer_width,
        hidden_activation=tf.math.sin,
        output_activation=tf.math.sin,
    )

    # Load the image
    input_image = cv2.imread(image_path)

    # Resize the image
    resized_img = cv2.resize(input_image, (siren_resolution, siren_resolution))

    # normalize the image
    input_img = resized_img / 255

    target = einops.rearrange(input_img, "h w c -> (h w) c")

    resolution = input_img.shape[0]

    # Generate a linear space from -1 to 1 with the same size as the resolution
    tmp = np.linspace(-1, 1, resolution)

    # Create a meshgrid for pixel coordinates
    x, y = np.meshgrid(tmp, tmp)

    # Reshape and concatenate x and y, and cast them to float32
    x_reshaped = x.reshape(-1, 1)
    y_reshaped = y.reshape(-1, 1)
```

```
    pixel_coordinates = tf.cast(tf.concat((x_reshaped, y_reshaped), 1), tf.float
32)

    used_patience = 0
    minimum_train_loss = np.inf
    minimum_loss_step_num = 0

    # Used For Plotting
    y_train_batch_loss = np.array([])
    x_train_loss_iterations = np.array([])

    learning_rate_change_steps = np.array([])

    # Index of the current learning rate, used to change the learning rate
    # when the training loss stops improving
    learning_rate_index = 0
    adam = Adam(
        learning_rates[learning_rate_index],
        weight_decay=weight_decay,
    )

    # find the temp_dir with the prefix if it exists
    # otherwise create a new one
    temp_dir = None
    for temp_dir in Path(tempfile.gettempdir()).iterdir():
        if temp_dir.is_dir() and temp_dir.name.startswith("siren_fit_image_"):
            break

    if not temp_dir.name.startswith("siren_fit_image_"):
        temp_dir = tempfile.mkdtemp(prefix="siren_fit_image_")

    checkpoint = tf.train.Checkpoint(siren)
    checkpoint_manager = tf.train.CheckpointManager(
        checkpoint,
        temp_dir,
        max_to_keep=1,
    )
    if use_last_checkpoint:
        print("\n\nRestoring from last checkpoint")
        checkpoint_manager.restore_or_initialize()

    overall_log = tqdm.tqdm(total=0, position=1, bar_format="{desc}")
    train_log = tqdm.tqdm(total=0, position=2, bar_format="{desc}")
    bar = tqdm.trange(num_iters, position=3)

    num_of_parameters = tf.math.add_n(
        [tf.math.reduce_prod(var.shape) for var in siren.trainable_variables]
    )
    print(f"\nNumber of Parameters => {num_of_parameters}")

    for i in bar:
        with tf.GradientTape() as tape:
            logits = siren(pixel_coordinates)

            current_train_loss = tf.reduce_mean((logits - target) ** 2)

        # Print initial train batch loss
        if i == 0:
            print("\n\n\n\n")
            print(f"Initial Training Loss => {current_train_loss:0.4f}")

        grads = tape.gradient(current_train_loss, siren.trainable_variables)

        adam.apply_gradients(zip(grads, siren.trainable_variables))

        current_train_loss = current_train_loss.numpy()
```

```
        if current_train_loss < minimum_train_loss:
            minimum_train_loss = current_train_loss
            minimum_loss_step_num = i
            checkpoint_manager.save()
        used_patience = i - minimum_loss_step_num

        y_train_batch_loss = np.append(y_train_batch_loss, current_train_loss)
        x_train_loss_iterations = np.append(x_train_loss_iterations, i)

        if i % refresh_rate == (refresh_rate - 1):
            learning_rates_left = len(learning_rates) - learning_rate_index
            patience_left = learning_patience - used_patience
            overall_description = (
                f"Minimum Train Loss => {minimum_train_loss:0.4f}    "
                + f"Learning Rates Left => {learning_rates_left}    "
                + f"Patience Left => {patience_left}    "
            )
            overall_log.set_description_str(overall_description)
            overall_log.refresh()

            train_description = f"Train Batch Loss => {current_train_loss:0.4f}    "
            train_log.set_description_str(train_description)
            train_log.update(refresh_rate)

            bar_description = f"Step => {i}"
            bar.set_description(bar_description)
            bar.refresh()

            # if the training loss has not improved for learning_patience
            if (
                current_train_loss > minimum_train_loss
                and i - minimum_loss_step_num > learning_patience
            ):
                if learning_rate_index == (len(learning_rates) - 1):
                    break
                learning_rate_index += 1
                adam.learning_rate = learning_rates[learning_rate_index]
                learning_rate_change_steps = np.append(learning_rate_change_step
s, i)
                checkpoint_manager.restore_or_initialize()

    checkpoint_manager.restore_or_initialize()

    # delete the temporary directory
    tf.io.gfile.rmtree(temp_dir)

    checkpoint_manager = tf.train.CheckpointManager(
        checkpoint, "artifacts/siren_fit_image/model", max_to_keep=1
    )
    checkpoint_manager.save()

    fig = plt.figure(figsize=(10, 10))

    # Create a grid of 2 rows and 2 columns
    grid = plt.GridSpec(2, 2, hspace=0.2, wspace=0.2)

    # Use the grid to specify the location of each subplot
    main_ax = fig.add_subplot(grid[0, :])
    y_image_ax = fig.add_subplot(grid[1, 0])
    x_image_ax = fig.add_subplot(grid[1, 1])

    main_ax.semilogy(x_train_loss_iterations, y_train_batch_loss, label="Training
Loss")
    for learning_rate_change_step in learning_rate_change_steps:
        main_ax.axvline(
```

```
            x=learning_rate_change_step,
            color="black",
            linestyle="dashed",
            label="Learning Rate Change",
        )
    main_ax.axvline(
        x=minimum_loss_step_num, color="red", linestyle="dashed", label="Minimum Lo
ss"
    )
    main_ax.set_xlabel("Iterations")
    main_ax.set_ylabel("Loss")
    main_ax.legend()

    output_image = einops.rearrange(
        logits.numpy(), "(h w) c -> h w c", h=siren_resolution, w=siren_resolution
    )

    # Downscale the input image then rescale it back up to make it a fair compar
ison
    downscaled_input_image = cv2.resize(
        input_image, (siren_resolution, siren_resolution)
    )
    rescaled_input_image = cv2.resize(
        downscaled_input_image, (input_image.shape[1], input_image.shape[0])
    )

    y_image_ax.imshow(
        cv2.cvtColor(rescaled_input_image, cv2.COLOR_BGR2RGB), interpolation="no
ne"
    )
    y_image_ax.set_title("Ground Truth")
    y_image_ax.axis("off")

    # reshape output to input image shape
    output_image = cv2.resize(
        output_image, (input_image.shape[1], input_image.shape[0])
    )

    x_image_ax.imshow(
        cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB), interpolation="none"
    )
    x_image_ax.set_title("Prediction")
    x_image_ax.axis("off")

    fig.suptitle("Siren – Card F: Image Fitting")

    plt.show()

    print("\n\n\n\n")

    print(f"Stop Iteration => {i}")

    # if the file already exists add a number to the end of the file name
    # to avoid overwriting
    file_index = 0
    while Path(f"artifacts/siren_fit_image/siren_img_{file_index}.png").exists():
        file_index += 1
    fig.savefig(f"artifacts/siren_fit_image/siren_img_{file_index}.png")

    # Save the config file as a yaml under the same name as the image
    config_path = Path(f"artifacts/siren_fit_image/siren_img_{file_index}.yaml")
    config_path.write_text(yaml.dump(config))

    # save the model
    checkpoint_manager.save()
    config_path = Path(f"artifacts/siren_fit_image/model/model.yaml")
```

```
    config_path.write_text(yaml.dump(config))


def test(model_path: Path):
    if model_path is None:
        model_path = Path("artifacts/siren_fit_image/model")

    if not model_path.exists():
        print("Model does not exist, run the train script first")
        return

    config_path = Path("artifacts/siren_fit_image/model/model.yaml")

    # HYPERPARAMETERS
    config = yaml.safe_load(config_path.read_text())
    num_hidden_layers = config["siren"]["num_hidden_layers"]
    hidden_layer_width = config["siren"]["hidden_layer_width"]
    siren_resolution = config["siren"]["resolution"]
    image_path = config["data"]["image_path"]

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(0x43966E87BD57227011B5B03B58785EC1)
    tf.random.set_seed(0x43966E87BD57227011B5B03B58785EC1)

    siren = SirenMLP(
        2,
        3,
        num_hidden_layers=num_hidden_layers,
        hidden_layer_width=hidden_layer_width,
        hidden_activation=tf.math.sin,
        output_activation=tf.math.sin,
    )

    checkpoint = tf.train.Checkpoint(siren)
    checkpoint.restore(tf.train.latest_checkpoint(model_path))

    # Generate an out of bounds pixel coordinate
    tmp = np.linspace(-1.5, 1.5, siren_resolution)

    # Create a meshgrid for pixel coordinates
    x, y = np.meshgrid(tmp, tmp)

    # Reshape and concatenate x and y, and cast them to float32
    x_reshaped = x.reshape(-1, 1)
    y_reshaped = y.reshape(-1, 1)
    pixel_coordinates = tf.cast(tf.concat((x_reshaped, y_reshaped), 1), tf.float
32)

    logits = siren(pixel_coordinates)

    fig = plt.figure(figsize=(10, 5))

    # Create a grid of 1 rows and 2 columns
    grid = plt.GridSpec(1, 2, hspace=0.2, wspace=0.2)

    # Use the grid to specify the location of each subplot
    y_image_ax = fig.add_subplot(grid[0, 0])
    x_image_ax = fig.add_subplot(grid[0, 1])

    output_image = einops.rearrange(
        logits.numpy(), "(h w) c -> h w c", h=siren_resolution, w=siren_resolution
    )

    # Load the image
    input_image = cv2.imread(image_path)
```

```
    # Downscale the input image then rescale it back up to make it a fair compar
ison
    downscaled_input_image = cv2.resize(
        input_image, (siren_resolution, siren_resolution)
    )
    rescaled_input_image = cv2.resize(
        downscaled_input_image, (input_image.shape[1], input_image.shape[0])
    )

    y_image_ax.imshow(
        cv2.cvtColor(rescaled_input_image, cv2.COLOR_BGR2RGB), interpolation="no
ne"
    )
    y_image_ax.set_title("Ground Truth")
    y_image_ax.axis("off")
    # if the file already exists add a number to the end of the file name

    # reshape output to input image shape
    output_image = cv2.resize(
        output_image, (input_image.shape[1], input_image.shape[0])
    )

    x_image_ax.imshow(
        cv2.cvtColor(output_image, cv2.COLOR_BGR2RGB), interpolation="none"
    )
    x_image_ax.set_title("Prediction")
    x_image_ax.axis("off")

    fig.suptitle("Siren – Card F: Out of Bounds Fitting")

    plt.show()

    fig.savefig(f"artifacts/siren_fit_image/siren_img_out_of_bounds.png")
```

```
import tensorflow as tf


class Linear(tf.Module):
    def __init__(
        self,
        num_inputs,
        num_outputs,
        bias=True,
        zero_init=False,
        siren_init=False,
        siren_first=False,
    ):
        rng = tf.random.get_global_generator()
        self.siren_first = siren_first

        stddev = tf.cast(tf.math.sqrt(2 / (num_inputs + num_outputs)), tf.float3
2)

        self.bias = bias

        w_initial_value = rng.normal(shape=[num_inputs, num_outputs], stddev=std
dev)
        if zero_init:
            w_initial_value = tf.zeros(shape=[num_inputs, num_outputs])
        elif siren_init:
            w_initial_value = rng.uniform(
                minval=-tf.math.sqrt(6 / num_inputs),
                maxval=tf.math.sqrt(6 / num_inputs),
                shape=[num_inputs, num_outputs],
            )
        elif siren_first:
            w_initial_value = rng.uniform(
                minval=-1,
                maxval=1,
                shape=[num_inputs, num_outputs],
            )

        self.w = tf.Variable(
            w_initial_value,
            trainable=True,
            name="Linear/w",
        )

        if self.bias:
            self.b = tf.Variable(
                tf.zeros(
                    shape=[1, num_outputs],
                ),
                trainable=True,
                name="Linear/b",
            )

    # create the logits by multiplying the inputs by the weights + the
    # optional bias
    def __call__(self, x):
        z = x @ self.w

        if self.siren_first:
            z *= 30

        if self.bias:
            z += self.b

        return z
```

```python
import tensorflow as tf

from modules.linear import Linear


class SirenMLP(tf.Module):
    def __init__(
        self,
        num_inputs,
        num_outputs,
        num_hidden_layers=0,
        hidden_layer_width=0,
        hidden_activation=tf.identity,
        output_activation=tf.identity,
        dropout_prob=0,
        zero_init=False,
    ):
        self.num_inputs = num_inputs
        self.num_outputs = num_outputs
        self.num_hidden_layers = num_hidden_layers
        self.hidden_layer_width = hidden_layer_width
        self.hidden_activation = hidden_activation
        self.output_activation = output_activation

        self.first_linear = Linear(num_inputs, hidden_layer_width, siren_first=T
rue)
        self.hidden_linears = [
            Linear(hidden_layer_width, hidden_layer_width, siren_init=True)
            for _ in range(self.num_hidden_layers)
        ]
        self.final_linear = Linear(
            self.hidden_layer_width,
            self.num_outputs,
            zero_init=zero_init,
            siren_init=True,
        )

        self.dropout_prob = dropout_prob

    def __call__(self, x):
        """Applies the MLP to the input

    Args:
      x (tf.tensor): input tensor of shape [batch_size, num_inputs]

    Returns:
      tf.tensor: output tensor of shape [batch_size, num_outputs]
    """
        x = self.hidden_activation(self.first_linear(x))

        for hidden_linear in self.hidden_linears:
            x = self.hidden_activation(hidden_linear(x))

        if self.dropout_prob > 0:
            x = tf.nn.dropout(x, self.dropout_prob)

        return self.output_activation(self.final_linear(x))
```

```python
#!/usr/bin/env python3

import argparse
import importlib
from pathlib import Path

import argcomplete


def main():
    parser = argparse.ArgumentParser(description="Choose an example to train:")
    parser.add_argument("runner", type=Path,
                        help="Path to the runner file")
    parser.add_argument("--config", "-c", type=Path,
                        nargs='?', help="Path to the config file")
    parser.add_argument("--restore_from_checkpoint", "-r", action="store_true",
                        help="Whether or not to use the last checkpoint")

    argcomplete.autocomplete(parser)
    args = parser.parse_args()

    runner = importlib.import_module(f"runners.{args.runner.stem}")
    runner.train(args.config, args.restore_from_checkpoint)


if __name__ == "__main__":
    main()
```

```python
#!/usr/bin/env python3

import argparse
import importlib
from pathlib import Path

import argcomplete


def main():
    parser = argparse.ArgumentParser(description="Choose an example to train:")
    parser.add_argument("runner", type=Path,
                        help="Path to the runner file")
    parser.add_argument("--model", "-m", type=Path,
                        nargs='?', help="Path to the model directory")

    argcomplete.autocomplete(parser)
    args = parser.parse_args()

    runner = importlib.import_module(f"runners.{args.runner.stem}")
    runner.test(args.model)


if __name__ == "__main__":
    main()
```

```
- Setup Virtual Enviroment (Recommended).

- Install Requirements and Data with ``` ./setup ```
    - You may have to run ``` chmod +x ./setup ``` first.

- Train a model in the ``` ./runners ``` directory. e.g. ``` train.py ./runners/
classify_cifar10.py ```.
    - Change parameters in the ``` configs ``` directory (or create your own).

- Test the model. e.g. ``` test.py ./runners/classify_cifar10.py ```.

- View the results in ``` ./artifacts ```.

# Siren
Below is my result which was trained with ```./runners/siren_mlp.py```. The mode
l never actually stopped automatically and was instead stopped by my maximum ite
ration hyperparameter. It had a final loss of 0.0003 and would likely have conti
nued to decrease as shown by the red vertical line at the very end.

![Alt text](artifacts/siren_fit_image/siren_img_0.png)

### Creative endeavers
I tested out the siren's out of bounds abilities. This can be run by running ‘‘‘ test.py ./runners/image_fit_c
ard_f.py ‘‘‘ The result is shown below.

![Alt text](artifacts/siren_fit_image/siren_img_out_of_bounds_f.png)

I thought that looked pretty awful so I tried it again after training with this rainbow image to see if it would work better with something that has repeating patterns in one direction.

![Alt text](artifacts/siren_fit_image/siren_img_out_of_bounds_r.png)

That also failed miserably but at least the colors match the theme. This can be a very overkill solution to getting a color palette from an image. It is at least good that it correctly displayed the picture in the middle. Since none of those coordi
nates were actually in the training set, it means the model is at least good at interpolating between points.

I spent a really long time trying to fit an stl file (specifically the frog that was hidden around the school), but I kept runn
ing out of memory. When I did it with a very low resolution version of the file it worked but the predicted mesh looked like a gobbly gook mess of vertices. The paper did something very similar so i know it's possible, but I mus
t have been doing something wrong :(
```