```
import tensorflow as tf


class Adam:
    def __init__(self, learning_rate=1e-3, beta_1=0.9, beta_2=0.999, ep=1e-7):
        # Initialize optimizer parameters and variable slots
        self.beta_1 = beta_1
        self.beta_2 = beta_2
        self.learning_rate = learning_rate
        self.ep = ep
        self.t = 1.0
        self.v_dvar, self.s_dvar = [], []
        self.built = False

    def apply_gradients(self, grads, vars):
        # Initialize variables on the first call
        if not self.built:
            for var in vars:
                v = tf.Variable(tf.zeros(shape=var.shape))
                s = tf.Variable(tf.zeros(shape=var.shape))
                self.v_dvar.append(v)
                self.s_dvar.append(s)
            self.built = True
        # Update the model variables given their gradients
        for i, (d_var, var) in enumerate(zip(grads, vars)):
            self.v_dvar[i].assign(
                self.beta_1 * self.v_dvar[i] + (1 - self.beta_1) * d_var
            )
            self.s_dvar[i].assign(
                self.beta_2 * self.s_dvar[i] + (1 - self.beta_2) * tf.square(d_v
ar)
            )
            v_dvar_bc = self.v_dvar[i] / (1 - (self.beta_1**self.t))
            s_dvar_bc = self.s_dvar[i] / (1 - (self.beta_2**self.t))
            var.assign_sub(
                self.learning_rate * (v_dvar_bc / (tf.sqrt(s_dvar_bc) + self.ep)
)
            )
        self.t += 1.0
        return
```

```
import tensorflow as tf


def he_init_uniform(shape):
     # Computes the He uniform initialization values for a weight matrix
    in_dim, out_dim = shape
    weight_vals = tf.random.uniform(
        shape=shape, minval=-tf.math.sqrt(1 / in_dim), maxval=tf.math.sqrt(1 / i
n_dim)
    )
    return weight_vals


def he_init(shape):
    # Computes the He normal initialization values for a weight matrix
    in_dim, out_dim = shape
    stddev = tf.sqrt(2.0 / tf.cast(in_dim, tf.float32))
    weight_vals = tf.random.normal(shape=shape, mean=0, stddev=stddev, seed=22)
    return weight_vals


class DenseLayer(tf.Module):
    def __init__(
        self,
        num_inputs,
        num_outputs,
        bias=True,
        activation=tf.identity,
        initializer=he_init_uniform,
    ):
        self.w = tf.Variable(
            initializer(shape=[num_inputs, num_outputs]),
            trainable=True,
            name="Linear/w",
        )
        self.activation = activation
        self.bias = bias

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

    def __call__(self, x):
        z = x @ self.w

        if self.bias:
            z += self.b

        return self.activation(z)
```

```python
import os

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from PIL import Image

from adam import Adam
from siren import Siren
from util import *

# Getting Test Card F
img = get_image("Testcard_F.jpg") / 255


# Training Image Fitting Model
siren_model = Siren(
    num_inputs=2, num_outputs=3, num_hidden_layers=5, hidden_layer_width=350
)

iterations = 1500
optimizer = Adam(1e-4)
model_output = siren_model.train_model(
    train_on=img,
    get_loss=get_loss_imagefit,
    iterations=iterations,
    shape=[273, 365, -1],
    optimizer=optimizer,
    frame_folder="image_fit",
)

img_tensor = tf.reshape(model_output, shape=[273, 365, -1])

plt.imshow(img_tensor)
plt.show()

tf_save_img(img_tensor, "artifacts/imagefit.jpg")


## upscaling image x2 (730x546) by interpolating points

x_vals_double, y_vals_double = tf.meshgrid(
    tf.linspace(-1, 1, 365 * 2), tf.linspace(-1, 1, 273 * 2)
)

cord_vals_double = tf.stack(
    [tf.reshape(y_vals_double, -1), tf.reshape(x_vals_double, -1)], axis=1
)

double_model_output = siren_model(tf.cast(cord_vals_double, dtype=tf.float32))
img_tensor_upscaled = tf.reshape(double_model_output, shape=[2 * 273, 2 * 365, -
1])

plt.imshow(img_tensor_upscaled)
plt.show()

tf_save_img(img_tensor_upscaled, "artifacts/imagefit_upscaled.jpg")
```

```
import os

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from PIL import Image

from adam import Adam
from siren import Siren
from util import *

# Getting Test Card F
img = get_image("Testcard_F.jpg") / 255
img = img.astype(np.float32)
# Preparing Poisson Model
sobel_siren = Siren(
    num_inputs=2, num_outputs=3, num_hidden_layers=5, hidden_layer_width=350
)

rainbow = get_image_resize("rainbow.jpg") / 255
rainbow = rainbow.astype(np.float32)

rainbow_sobel_stack = get_sobel(tf.convert_to_tensor(rainbow)[None, :])[-1]
img_sobel_stack = get_sobel(tf.convert_to_tensor(img)[None, :])[-1]


#to make rainbow more prominent scale gradients by 3 the gradients
combined_sobel = 3 * rainbow_sobel_stack + img_sobel_stack

iterations = 5000
optimizer = Adam(1e-4)

model_output = sobel_siren.train_model(
    train_on=combined_sobel,
    get_loss=get_loss_poisson,
    iterations=iterations,
    shape=[273, 365, -1],
    optimizer=optimizer,
    frame_folder="poisson",
)

img_tensor = tf.reshape(model_output, shape=[273, 365, -1])
plt.imshow(img_tensor)
plt.show()
tf_save_img(img_tensor, "artifacts/poisson_blend.jpg")
```

```
import tensorflow as tf


def siren_init_first(shape, omega_0):
    in_dim, out_dim = shape
    weight_vals = tf.random.uniform(shape=shape, minval=-1 / in_dim, maxval=1 /
out_dim)
    return weight_vals


def siren_init_layer(shape, omega_0):
    in_dim, out_dim = shape
    weight_vals = tf.random.uniform(
        shape=shape,
        minval=-tf.math.sqrt(6 / in_dim) / omega_0,
        maxval=tf.math.sqrt(6 / in_dim) / omega_0,
    )
    return weight_vals



class SineLayer(tf.Module):
    def __init__(
        self, num_inputs, num_outputs, siren_initializer, bias=True, omega_0=30
    ):
        """Imeplementation of sine layer as denoted in the paper. To speed up training omega_0
    is used in the same fashion as mentioned in Appendix 1.5, leveraging omega_0 for all layers of the Siren
    network
    """

        self.omega_0 = omega_0

        self.w = tf.Variable(
            siren_initializer(shape=[num_inputs, num_outputs], omega_0=self.omeg
a_0),
            trainable=True,
            name="Linear/w",
        )

        self.bias = bias

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

    def __call__(self, x):
        z = x @ self.w

        if self.bias:
            z += self.b

        return tf.math.sin(self.omega_0 * z)
```

```
import tensorflow as tf

from dense import DenseLayer
# implementation of Siren network from https://arxiv.org/pdf/2006.09661.pdf
from sinelayer import *
from util import *


class Siren(tf.Module):
    def __init__(self, num_inputs, num_outputs, num_hidden_layers, hidden_layer_
width):
        # first layer has slightly different initialization scheme
        self.layers = [
            SineLayer(
                num_inputs, hidden_layer_width, siren_initializer=siren_init_fir
st
            )
        ]

        for i in range(num_hidden_layers):
            self.layers.append(
                SineLayer(
                    num_inputs=hidden_layer_width,
                    num_outputs=hidden_layer_width,
                    siren_initializer=siren_init_layer,
                )
            )


        """last layer uses a clipped relu as points can only go from [0, 1].
            Doing so helped improved training time"""

        clipped_relu = lambda x: tf.clip_by_value(tf.nn.relu(x), 0, 1)
        self.layers.append(
            DenseLayer(
                num_inputs=hidden_layer_width,
                num_outputs=num_outputs,
                activation=clipped_relu,
            )
        )

    def __call__(self, x):
        for layer in self.layers:
            x = layer(x)

        return x

    def train_model(
        self, train_on, get_loss, iterations, shape, optimizer=Adam, frame_folde
r=None
    ):

        """"arranging the points from -1 to 1 proved to be the most effective wa
y to train the network.
        this is similar to what the Siren paper does"""

        bar = trange(iterations)
        x_vals, y_vals = tf.meshgrid(
            tf.linspace(-1, 1, shape[1]), tf.linspace(-1, 1, shape[0])
        )

        cord_vals = tf.stack([tf.reshape(y_vals, -1), tf.reshape(x_vals, -1)], axis=1)
        train_on = tf.convert_to_tensor(train_on, dtype=tf.float32)
        train_on_true = tf.reshape(train_on, shape=[shape[0] * shape[1], -1])

        frames = []

        for i in bar:
            with tf.GradientTape() as tape:
```

```
                loss_output = get_loss(self, cord_vals, train_on_true, shape)
                grads = tape.gradient(loss_output[0], self.trainable_variables)
                optimizer.apply_gradients(grads=grads, vars=self.trainable_variables)

            if i % 10 == (10 - 1):
                bar.set_description(f"Step {i}; loss => {loss_output[0].numpy():0.4f}")
                bar.refresh()

                if frame_folder:
                    plt.imshow(tf.reshape(loss_output[1], shape=[273, 365, -1]))
                    plt.axis("off")
                    plt.savefig(f"{frame_folder}/frame_{i}.png", bbox_inches="tight")
                    plt.close()
                    frames.append(Image.open(f"{frame_folder}/frame_{i}.png"))

        if frame_folder:
            frames[0].save(
                f"{frame_folder}.gif",
                save_all=True,
                append_images=frames,
                duration=200,
                loop=0,
            )
        return loss_output[1]
```

```
import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from PIL import Image
from tqdm import trange

from adam import Adam


def get_image(image_path):
    img = np.asarray(Image.open(image_path))
    return img


def get_loss_imagefit(siren_model, cord_vals, image_true, shape):
    model_output = siren_model(tf.cast(cord_vals, dtype=tf.float32))
    return tf.reduce_mean((model_output - image_true) ** 2), model_output


def get_image_resize(image_path):
    img = Image.open(image_path)
    img = img.resize((365, 273), Image.Resampling.LANCZOS)
    return np.asarray(img)


def get_sobel(img):
    sobel_y = tf.image.sobel_edges(img)[0, :, :, :, 0]
    sobel_x = tf.image.sobel_edges(img)[0, :, :, :, 1]
    sobel_stack = tf.stack([sobel_x, sobel_y], axis=-1)
    return sobel_x, sobel_y, sobel_stack


def get_loss_poisson(siren_model, cord_vals, sobel_true, shape):
    model_output = siren_model(tf.cast(cord_vals, dtype=tf.float32))
    model_output_reshape = tf.reshape(model_output, shape=shape)[None, :]

    return (
        #assumning rgb (3 channel images)
        tf.reduce_mean((tf.reshape(get_sobel(model_output_reshape)[-1], shape=[s
hape[0] * shape[1], 6])- sobel_true)** 2),
        model_output,
    )


def tf_save_img(img_tensor, filename):
    img = img_tensor.numpy() * 255
    img = img.astype(np.uint8)
    img = Image.fromarray(img)
    img.save(filename)
```