```
import tensorflow as tf


class Adam:
    def __init__(self,
                 learning_rate=1e-3,
                 beta_1=0.9,
                 beta_2=0.999,
                 epsilon=1e-7,
                 weight_decay=5e-3,):
        self.learning_rate = learning_rate
        self.beta_1 = beta_1
        self.beta_2 = beta_2
        self.epsilon = epsilon
        self.weight_decay = weight_decay

    def apply_gradients(
        self,
        grads_and_vars
    ):
        for grad, var in grads_and_vars:
            m = tf.Variable(tf.zeros(shape=var.shape))
            v = tf.Variable(tf.zeros(shape=var.shape))
            m.assign(self.beta_1 * m + (1 - self.beta_1) * tf.convert_to_tensor(
grad))
            v.assign(self.beta_2 * v + (1 - self.beta_2) * tf.convert_to_tensor(
grad) ** 2)
            m_hat = m / (1 - self.beta_1)
            v_hat = v / (1 - self.beta_2)
            var.assign(var - self.learning_rate * m_hat /
                       (tf.sqrt(v_hat) + self.epsilon))
            if (var.name.endswith('kernel') or var.name.endswith('w')):
                var.assign(var - self.weight_decay * var * self.learning_rate)
```

```python
import tensorflow as tf


class AugmentData:
    def __init__(self, augmentation_multiplier: int):
        """Initializes the AugmentData class

        Args:
            augmentation_multiplier (int): the percentage of the data to augment,
            must be between 0 and 1
        """
        self.augmentation_multiplier = augmentation_multiplier

    def __call__(
        self, labels: tf.Tensor, images: tf.Tensor
    ) -> tuple[tf.Tensor, tf.Tensor]:
        """Takes the data and augments it,
        by applying random zooms, rotations, brightness, contrast, hue, and
        saturation
        Args:
            images (tf.Tensor): a tensor of RGB images of shape
            [batch_size, height, width, 3]
            labels (tf.Tensor): a tensor of labels of shape [batch_size]
        Returns:
            tuple[tf.Tensor, tf.Tensor]: a tuple of the augmented images and labels,
            the images have shape [batch_size +
            augmentation_multiplier*batch_size*6, height, width, 3] and the labels have
            shape [batch_size + augmentation_multiplier*batch_size*6]
        """

        zoom_indices = tf.random.uniform(
            shape=[
                tf.cast(
                    tf.shape(images)[0].numpy() * self.augmentation_multiplier,
tf.int32
                )
            ],
            maxval=tf.shape(images)[0],
            dtype=tf.int32,
        )
        flipped_indices = tf.random.uniform(
            shape=[
                tf.cast(
                    tf.shape(images)[0].numpy() * self.augmentation_multiplier,
tf.int32
                )
            ],
            maxval=tf.shape(images)[0],
            dtype=tf.int32,
        )
        brightness_indices = tf.random.uniform(
            shape=[
                tf.cast(
                    tf.shape(images)[0].numpy() * self.augmentation_multiplier,
tf.int32
                )
            ],
            maxval=tf.shape(images)[0],
            dtype=tf.int32,
        )
        contrast_indices = tf.random.uniform(
            shape=[
                tf.cast(
                    tf.shape(images)[0].numpy() * self.augmentation_multiplier,
tf.int32
                )
```

```
            ],
            maxval=tf.shape(images)[0],
            dtype=tf.int32,
        )
        hue_indices = tf.random.uniform(
            shape=[
                tf.cast(
                    tf.shape(images)[0].numpy() * self.augmentation_multiplier,
tf.int32
                )
            ],
            maxval=tf.shape(images)[0],
            dtype=tf.int32,
        )
        saturation_indices = tf.random.uniform(
            shape=[
                tf.cast(
                    tf.shape(images)[0].numpy() * self.augmentation_multiplier,
tf.int32
                )
            ],
            maxval=tf.shape(images)[0],
            dtype=tf.int32,
        )

        # apply augmentations to the images
        zoomed_images = tf.gather(images, zoom_indices)
        zoomed_images = tf.image.random_crop(
            zoomed_images, [zoomed_images.shape[0], 24, 24, 3]
        )
        zoomed_images = tf.image.resize(zoomed_images, [32, 32])

        flipped_images = tf.gather(images, flipped_indices)
        flipped_images = tf.image.random_flip_left_right(flipped_images)

        brightness_images = tf.gather(images, brightness_indices)
        brightness_images = tf.image.random_brightness(brightness_images, 0.2)

        contrast_images = tf.gather(images, contrast_indices)
        contrast_images = tf.image.random_contrast(contrast_images, 0.2, 0.5)

        hue_images = tf.gather(images, hue_indices)
        hue_images = tf.image.random_hue(hue_images, 0.2)

        saturation_images = tf.gather(images, saturation_indices)
        saturation_images = tf.image.random_saturation(saturation_images, 0.2, 0
.5)

        # combine the augmented images with the original images
        output_images = tf.concat(
            [
                images,
                zoomed_images,
                flipped_images,
                brightness_images,
                contrast_images,
                hue_images,
                saturation_images,
            ],
            axis=0,
        )

        output_labels = tf.concat(
            [
                labels,
                tf.gather(labels, zoom_indices),
```

```
            tf.gather(labels, flipped_indices),
            tf.gather(labels, brightness_indices),
            tf.gather(labels, contrast_indices),
            tf.gather(labels, hue_indices),
            tf.gather(labels, saturation_indices),
        ],
        axis=0,
    )

    return output_labels, output_images
```

```
import tensorflow as tf


class Conv2D(tf.Module):
    def __init__(
        self,
        input_channels: int,
        output_channels: int,
        kernel_shape: tuple[int, int],
        stride: int = 1,
        bias_enabled: bool = False,
    ):
        """Initializes the Conv2D class

        Args:
            input_channels (int): How many channels the input has,
                e.g. for the first layer this is 3 for RGB images,
                1 for grayscale images
            output_channels (int): How many filters the convolution should have
            kernel_shape tuple[int, int]: Uses a filter of size
                kernel_height x kernel_width
            stride (int, optional): The stride of the convolution.
            bias (bool, optional): Whether or not to use a bias.
        """
        self.stride = stride
        self.bias_enabled = bias_enabled

        rng = tf.random.get_global_generator()

        # He initialization
        stddev = tf.sqrt(2 / (input_channels * kernel_shape[0] * kernel_shape[1]
))

        self.kernel = tf.Variable(
            rng.normal(
                shape=[
                    kernel_shape[0],
                    kernel_shape[1],
                    input_channels,
                    output_channels,
                ],
                stddev=stddev,
            ),
            trainable=True,
            name="Conv2D/kernel",
        )

        if self.bias_enabled:
            self.bias = tf.Variable(
                tf.constant(0.01, shape=[output_channels]),
                trainable=True,
                name="Conv2D/bias",
            )

    def __call__(self, x: tf.Tensor):
        """Applies the convolution to the input

        Args:
            input_tensor (tf.Tensor): The input to apply the convolution to.
            Shape should be [batch_size, height, width, input_channels]

        Returns:
            tf.Tensor: The result of the convolution with the bias added
                Shape should be [batch_size, height, width, output_channels]
        """
        result = tf.nn.conv2d(
```

```
            x, self.kernel, [1, self.stride, self.stride, 1], "SAME"
        )
        if self.bias_enabled:
            result = result + self.bias
        return result
```

```
import tensorflow as tf


class EmbedToVocabFile(tf.Module):
    def __init__(self, vocab_file, embedding_depth):
        rng = tf.random.get_global_generator()

        self.vocab_file = vocab_file
        vocab_initializer = tf.lookup.TextFileInitializer(
            vocab_file.name,
            key_dtype=tf.string,
            key_index=tf.lookup.TextFileIndex.WHOLE_LINE,
            value_dtype=tf.int64,
            value_index=tf.lookup.TextFileIndex.LINE_NUMBER,
        )

        self.vocab_table = tf.lookup.StaticHashTable(
            vocab_initializer, default_value=-1
        )

        self.embedding_depth = embedding_depth
        stddev = tf.cast(
            tf.sqrt(2 / (self.vocab_table.size() * embedding_depth)), tf.float32
        )
        self.embedding = tf.Variable(
            rng.normal(
                shape=[self.vocab_table.size(), embedding_depth],
                stddev=stddev,
            ),
            trainable=True,
            name="Embedder/embedding",
        )

    def tokens_to_ids(self, tokens):
        """
        Convert the tokens into integers.

        Args:
            tokens (tf.Tensor): The tokenized text. Shape: [batch_size, num_tokenized_words]

        Returns:
            tf.Tensor: The integer IDs of the tokens.
        """
        return tf.cast(self.vocab_table.lookup(tokens), tf.int32)

    def decode(self, logits):
        """
        Decode the logits into tokens.

        Args:
            logits (tf.Tensor): The logits. Shape: [batch_size, num_tokenized_words, vocab_size]

        Returns:
            tf.Tensor: The tokens corresponding to the logits. Shape: [batch_size, num_tokenized_words]
        """
        reverse_vocab = []
        with open(self.vocab_file.name, "r", encoding="utf-8") as f:
            for line in f:
                reverse_vocab.append(line.strip())

        # Get the most likely token ID for each embedding
        probabilities = tf.nn.softmax(logits, axis=-1)

        # Get the most likely token ID for each embedding
        token_ids = tf.argmax(probabilities, axis=-1, output_type=tf.int64)
```

```python
        # Convert the token IDs into tokens
        tokens = tf.gather(reverse_vocab, token_ids)

        return tokens

    def get_vocab_size(self):
        """
        Get the size of the vocabulary.

        Returns:
            int: The size of the vocabulary
        """
        return tf.cast(self.vocab_table.size(), tf.int32)

    def __call__(self, tokens):
        """
        Embed the tokenized text.

        Args:
            tokens (tf.Tensor): The tokenized text. Shape: [batch_size, num_tokenized_words]

        Returns:
            tf.Tensor: The embeddings of the tokens.
            Shape should be [batch_size, num_tokenized_words, embedding_depth]
        """
        token_ids = self.tokens_to_ids(tokens)
        embeddings = tf.nn.embedding_lookup(self.embedding, token_ids)
        return embeddings
```

```python
import tensorflow as tf


class Embedder(tf.Module):
    def __init__(self, embedding_buckets, embedding_depth):
        rng = tf.random.get_global_generator()

        self.embedding_buckets = embedding_buckets
        self.embedding_depth = embedding_depth
        stddev = tf.sqrt(2 / (embedding_buckets * embedding_depth))
        self.embedding = tf.Variable(
            rng.normal(
                shape=[
                    embedding_buckets,
                    embedding_depth,
                ],
                stddev=stddev,
            ),
            trainable=True,
            name="Embedder/embedding",
        )

    def __call__(self, tokens):
        """
        Embed the tokenized text.

        Args:
            tokens (tf.Tensor): The tokenized text. Shape: [batch_size, num_tokenized_words]

        Returns:
            tf.Tensor: The embeddings of the tokens.
            Shape should be [batch_size, num_tokenized_words, embedding_depth]
        """
        hashed_tokens = tf.strings.to_hash_bucket_fast(tokens, self.embedding_bu
ckets)

        embeddings = tf.nn.embedding_lookup(self.embedding, hashed_tokens)

        return embeddings
```

```
import tensorflow as tf


class GroupNorm(tf.Module):
    def __init__(
        self,
        num_groups: int,
        input_depth: int,
        input_channels: int = 2,
        epsilon: float = 1e-5,
    ):
        """Initializes the GroupNorm class

    Args:
      num_groups (int): the number of groups to split the channels into
      input_depth (int): the depth of the input tensor
      input_channels (int): the number of channels in the input tensor
      epsilon (float, optional): small value for numerical stability.
        Defaults to 1e-5.
    """
        self.num_groups = int(num_groups)
        self.epsilon = epsilon
        self.input_depth = input_depth
        self.input_channels = input_channels

        if input_channels == 1:
            self.gamma = tf.Variable(
                tf.ones(shape=[1, 1, input_depth]),
                trainable=True,
                name="GroupNorm/gamma",
            )

            self.beta = tf.Variable(
                tf.zeros(shape=[1, 1, input_depth]),
                trainable=True,
                name="GroupNorm/beta",
            )

        elif input_channels == 2:
            self.gamma = tf.Variable(
                tf.ones(shape=[1, 1, 1, input_depth]),
                trainable=True,
                name="GroupNorm/gamma",
            )

            self.beta = tf.Variable(
                tf.zeros(shape=[1, 1, 1, input_depth]),
                trainable=True,
                name="GroupNorm/beta",
            )

    def __call__(self, x: tf.Tensor) -> tf.Tensor:
        """Applies group normalization to the input tensor

    Args:
      x (tf.Tensor): the input tensor

    Returns:
      tf.Tensor: the normalized tensor
    """
        if self.input_channels == 1:
            (
                batch_size,
                context_length,
                model_dim,
            ) = x.shape
```

```
            x = tf.reshape(
                x,
                [
                    batch_size,
                    context_length,
                    self.num_groups,
                    model_dim // self.num_groups,
                ],
            )

            mean, variance = tf.nn.moments(x, axes=[1, 3], keepdims=True)
            x = (x - mean) / tf.sqrt(variance + self.epsilon)

            x = tf.reshape(x, [batch_size, context_length, model_dim])

        elif self.input_channels == 2:
            (
                batch_size,
                input_height,
                input_width,
                input_depth,
            ) = x.shape
            x = tf.reshape(
                x,
                [
                    batch_size,
                    input_height,
                    input_width,
                    self.num_groups,
                    input_depth // self.num_groups,
                ],
            )

            mean, variance = tf.nn.moments(x, axes=[1, 2, 4], keepdims=True)
            x = (x - mean) / tf.sqrt(variance + self.epsilon)

            x = tf.reshape(x, [batch_size, input_height, input_width, input_dept
h])

        return x * self.gamma + self.beta
```

```python
import idx2numpy
import tensorflow as tf


def load_idx_data(
    filename: str,
):
    """Uses idx2numpy to load an idx file into a tensor

    Args:
        filename (str): The path to the idx file

    Returns:
        tf.tensor: The tensor containing the data from the idx file
    """
    idx2numpy.convert_from_file(filename)
    numpy_data = idx2numpy.convert_from_file(filename)
    return tf.convert_to_tensor(numpy_data)[..., tf.newaxis]
```

```
import pickle
from pathlib import Path
import tensorflow as tf


def load_pickle_data(filename: Path, label_id: str = "labels", data_id: str = "data
"):
    """Load data from a pickle file. Convert into a tensorflow dataset.
    Return the labels and images as a tuple.

    Args:
        filename (Path): Path to the pickle file.

    Returns:
        tuple: Tuple of labels and images.
    """
    with open(filename, 'rb') as fo:
        data = pickle.load(fo, encoding='bytes')

    label_id = tf.cast(data[label_id.encode('utf-8')], tf.int32)
    data = tf.cast(data[data_id.encode('utf-8')], tf.float32)

    # data -- a 10000x3072 numpy array of uint8s. Each row of the array stores
    # a 32x32 colour image. The first 1024 entries contain the red channel
    # values, the next 1024 the green, and the final 1024 the blue. The image
    # is stored in row-major order, so that the first 32 entries of the array
    # are the red channel values of the first row of the image.

    data = tf.reshape(data, [-1, 3, 32, 32])
    # convert from (batch_size, depth, height, width) to
    # (batch_size, height, width, depth)
    data = tf.transpose(data, [0, 2, 3, 1])

    data = tf.cast(data, tf.float32)/255.0
    return label_id, data
```

```
import numpy as np
import tensorflow as tf


class PositionalEncoding(tf.Module):
    """PositionalEncoding layer.

  This is an implementation of positional encoding as described in the
  paper "Attention is all you Need" (Vaswani et al., 2017).

  This layer first calculates a positional encoding matrix, then adds it to
  the inputs.

  Args:
    max_position: Maximum position to encode.
    model_dim: Size of each attention head for value, query, and queue.

  Call arguments:
    inputs: Input `Tensor` of shape `(B, seq_len, model_dim)`.

  Returns:
    output: The result of the computation, of shape `(B, seq_len, model_dim)`,
  """

    # FIXME: this whole thing is a mess
    def __init__(self, max_position, model_dim):
        super(PositionalEncoding, self).__init__()
        self.positional_encoding = self._calculate_positional_encoding(
            max_position, model_dim
        )

    def _calculate_positional_encoding(self, max_position, model_dim):
        positions = np.arange(max_position)[:, np.newaxis]
        div_term = np.exp(-np.arange(0, model_dim, 2) * (np.log(10000.0) / model
_dim))
        positional_encoding = np.zeros((max_position, model_dim))
        positional_encoding[:, 0::2] = np.sin(positions * div_term)
        positional_encoding[:, 1::2] = np.cos(positions * div_term)
        return tf.convert_to_tensor(
            positional_encoding[np.newaxis, ...], dtype=tf.float32
        )

    def __call__(self, inputs):
        return inputs + self.positional_encoding[:, : inputs.shape[1], :]
```

```python
import einops
import tensorflow as tf


class Tokenizer(tf.Module):
    def __init__(self, num_word_to_tokenize, pre_batched=True):
        self.num_word_to_tokenize = num_word_to_tokenize
        self.pre_batched = pre_batched

    def __call__(self, text: tf.Tensor):
        """
        Tokenize the input text.

        Args:
            text (tf.Tensor): The text to tokenize. Shape: [batch_size, text_length]

        Returns:
            tokens (tf.Tensor): The tokenized text. Shape: [batch_size, num_word_to_tokenize]
        """
        tokens = tf.strings.split(text, sep=" ")
        if self.pre_batched:
            tokens = tokens[:, : self.num_word_to_tokenize]
            tokens = tokens.to_tensor(default_value=b"<PAD>")
            if tokens.shape[1] < self.num_word_to_tokenize:
                tokens = tf.pad(
                    tokens,
                    [
                        [0, 0],
                        [0, self.num_word_to_tokenize - tokens.shape[1]],
                    ],
                    constant_values=b"<PAD>",
                )
        else:
            # Pad the sequence with <PAD> tokens to make it a multiple of contex
t_length
            num_tokens = tokens.shape[0]
            remainder = num_tokens % (self.num_word_to_tokenize)

            if remainder != 0:
                tokens = tf.pad(
                    tokens,
                    [[0, (self.num_word_to_tokenize) - remainder]],
                    constant_values=b"<PAD>",
                )

            tokens = einops.rearrange(
                tokens,
                "(batch context_length) -> batch context_length",
                context_length=self.num_word_to_tokenize,
            )

        return tokens
```

```python
import tensorflow as tf


class BasisExpansion(tf.Module):
    def __init__(self, num_bases, num_inputs, num_outputs):
        rng = tf.random.get_global_generator()

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

        self.mu = tf.Variable(
            rng.normal(shape=[num_inputs, num_bases], stddev=stddev),
            trainable=True,
            name="BasisExpansion/mu",
        )

        self.sigma = tf.Variable(
            rng.normal(shape=[num_inputs, num_bases], stddev=stddev),
            trainable=True,
            name="BasisExpansion/sigma",
        )

    def __call__(self, x):
        z = tf.exp(-(x - self.mu)**2 / (self.sigma**2))

        return z
```

```
import tensorflow as tf

from modules.mlp import MLP
from modules.residual_block import ResidualBlock


class Classifier(tf.Module):
    def __init__(
        self,
        input_depth: int,
        layer_depths: list[int],
        layer_kernel_sizes: list[tuple[int, int]],
        num_classes: int,
        input_size: int,
        resblock_size: int = 2,
        pool_size: int = 2,
        dropout_prob: float = 0.5,
        group_norm_num_groups: int = 32,
        num_hidden_layers: int = 1,
        hidden_layer_width: int = 128,
    ):
        """Initializes the Classifier class

    Args:
        input_depth (int): number of input channels,
            e.g. this is 3 for RGB images, 1 for grayscale images
        layer_depths (list[int]): A list of how many filters each layer
            should have the length of this list determines how many layers
            the network has
        layer_kernel_sizes (list[tuple[int, int]]): A list of the kernel
            sizes for each layer, the length of this list should be the
            same as the length of layer_depths
        num_classes (int): How many classes the network should classify
            affects the output size of the call
        input_size (int): The size of the input image, the image should be
            square, e.g. 28 for MNIST
        num_hidden_layers (int): The number of hidden layers in the MLP
        hidden_layer_width (int): The width of the hidden layers in the MLP
        pool_every_n_layers (int, optional): Adds a max pooling layer
            every n layers. Defaults to 0. Aka, no
            pooling layers.
        pool_size (int, optional): The size of the kernel for the max
            pooling layer. Defaults to 2.
        dropout_prob (float, optional): The probability of dropping a node
        group_norm_num_groups (int, optional): The number of groups to
        split the channels into for group normalization. Defaults to 32.
    """
        self.layer_kernel_sizes = layer_kernel_sizes
        self.input_size = input_size
        self.pool_size = pool_size
        self.dropout_prob = dropout_prob

        self.flatten_size = layer_depths[-1]

        self.residual_blocks = []
        for layer_depth, layer_kernel_size, group_norm_num in zip(
            layer_depths, self.layer_kernel_sizes, group_norm_num_groups
        ):
            self.residual_blocks.append(
                ResidualBlock(
                    input_depth,
                    layer_depth,
                    layer_kernel_size,
                    group_norm_num,
                    resblock_size,
                )
```

```
            )
            input_depth = layer_depth

        self.fully_connected = MLP(
            self.flatten_size,
            num_classes,
            num_hidden_layers,
            hidden_layer_width,
            hidden_activation=tf.nn.relu,
            zero_init=True,
        )

    def __call__(self, x: tf.Tensor):
        """Applies the classifier to the input,

    Args:
        x (tf.Tensor): The Image to classify, should have shape
            [batch_size, input_size, input_size, input_depth]

    Returns:
        tf.Tensor: The logits of the classification, should have shape
            [batch_size, num_classes]
    """

        for residual_block in self.residual_blocks:
            x = residual_block(x)

        x = tf.nn.max_pool2d(x, self.pool_size, strides=2, padding="VALID")

        x = tf.nn.dropout(x, rate=self.dropout_prob)

        if self.flatten_size != (x.shape[3]):
            raise ValueError("Flatten size does not match output tensor shape")

        x = tf.nn.avg_pool2d(x, [x.shape[1], x.shape[2]], strides=1, padding="V
ALID")

        x = tf.reshape(x, [-1, self.flatten_size])

        x = self.fully_connected(x)

        return x
```

```python
import tensorflow as tf

from helpers.embedder import Embedder
from helpers.tokenizer import Tokenizer
from modules.mlp import MLP


class EmbedClassifier(tf.Module):
    def __init__(
        self,
        num_embedding,
        embedding_depth,
        num_word_to_tokenize,
        dropout_prob,
        num_hidden_layers,
        hidden_layer_width,
        num_classes,
    ):
        self.num_word_to_tokenize = num_word_to_tokenize
        self.embedding_depth = embedding_depth
        self.tokenizer = Tokenizer(num_word_to_tokenize)
        self.embedder = Embedder(num_embedding, embedding_depth)

        self.mlp = MLP(
            num_word_to_tokenize * embedding_depth,
            num_classes,
            num_hidden_layers,
            hidden_layer_width,
            tf.nn.relu,
            tf.nn.softmax,
            dropout_prob,
        )

    def __call__(self, text: tf.Tensor):
        """Applies the embedding and MLP to the text.

        Args:
            text (tf.Tensor): The text to tokenize.
            Shape should be [batch_size]

        Returns:
            tf.Tensor: The logits of the classes.
            Shape should be [batch_size, num_classes]
        """

        tokens = self.tokenizer(text)
        embeddings = self.embedder(tokens)

        embeddings = tf.reshape(
            embeddings, [-1, self.num_word_to_tokenize * self.embedding_depth]
        )

        return self.mlp(embeddings)
```

```
import tensorflow as tf


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

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

        self.bias = bias

        if zero_init:
            self.w = tf.Variable(
                tf.zeros(shape=[num_inputs, num_outputs]),
                trainable=True,
                name="Linear/w",
            )
        else:
            self.w = tf.Variable(
                rng.normal(shape=[num_inputs, num_outputs], stddev=stddev),
                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.bias:
            z += self.b

        return z
```

```
import tensorflow as tf

from modules.linear import Linear


class MLP(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.hidden_linear = (
            Linear(self.hidden_layer_width, self.hidden_layer_width)
            if self.num_hidden_layers > 0
            else None
        )
        self.first_linear = Linear(num_inputs, hidden_layer_width)
        self.final_linear = Linear(
            self.hidden_layer_width, self.num_outputs, zero_init=zero_init
        )
        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 _ in range(self.num_hidden_layers):
            x = self.hidden_activation(self.hidden_linear(x))

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

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

```
import einops
import tensorflow as tf

from modules.linear import Linear


class MultiHeadAttention(tf.Module):
    """MultiHeadAttention layer.

  This is an implementation of multi−headed attention as described in the
  paper "Attention is all you Need" (Vaswani et al., 2017).

  This layer first projects ‘query‘, ‘key‘ and ‘value‘. These are
  (effectively) a list of tensors of length ‘num_heads‘, where the
  corresponding shapes are ‘(batch_size, seq_len, model_dim)‘

  Then, the query and key tensors are dot−producted and scaled. These are
  softmaxed to obtain attention probabilities. The value tensors are then
  interpolated by these probabilities, then concatenated back to a single
  tensor.

  Finally, the result tensor with the last dimension as model_dim can take an
  linear projection and return.

  Args:
    num_heads: Number of attention heads.
    model_dim: Size of each attention head for value, query, and queue.
    dropout: Dropout probability.

  Call arguments:
    query: Query ‘Tensor‘ of shape ‘(B, seq_len, model_dim)‘.
    value: Value ‘Tensor‘ of shape ‘(B, seq_len, model_dim)‘.
    key: Optional key ‘Tensor‘ of shape ‘(B, seq_len, model_dim)‘. If not given, will use ‘value‘ for both ‘key‘ and ‘v
alue‘, which is the most common case.
    mask: Optional mask tensor of shape ‘(B, seq_len, seq_len)‘.

  Returns:
    output: The result of the computation, of shape ‘(B, seq_len, model_dim)‘,
  """

    def __init__(self, num_heads, model_dim, dropout_prob=0.1):
        self.num_heads = num_heads
        self.model_dim = model_dim
        self.dropout_prob = dropout_prob

        assert model_dim % num_heads == 0

        self.depth = model_dim // num_heads

        self.wq = Linear(model_dim, model_dim)
        self.wk = Linear(model_dim, model_dim)
        self.wv = Linear(model_dim, model_dim)
        self.wo = Linear(model_dim, model_dim)

    def _split_heads(self, inputs):
        """Split the last dimension into (num_heads, depth). Transpose and organize the result

    Args:
      inputs: input tensor of shape ‘(batch_size, seq_len, model_dim)‘
      batch_size: batch size

    Returns:
      A tensor with shape ‘(batch_size, num_heads, seq_len, depth)‘
    """
        output = einops.rearrange(
            inputs,
```

```
            "batch seq (heads depth) -> batch heads seq depth",
            heads=self.num_heads,
        )

        return output

    def _scaled_dot_product_attention(self, q, k, v, mask=None):
        """Scaled dot product attention

    Args:
      q: query tensor of shape `(batch_size, num_heads, seq_len, depth)`
      k: key tensor of shape `(batch_size, num_heads, seq_len, depth)`
      v: value tensor of shape `(batch_size, num_heads, seq_len, depth)`
      mask: optional mask tensor of shape `(batch_size, seq_len, seq_len)`

    Returns:
      output: output tensor of shape `(batch_size, num_heads, seq_len, depth)`
    """
        # matmul q and k while transposing k: (batch_size, num_heads, seq_len, s
eq_len)
        # Transpose the last two dimensions of k
        k_transposed = tf.transpose(k, [0, 1, 3, 2])
        matmul_qk = tf.einsum("bnqd,bndk->bnqk", q, k_transposed)

        scaled_attention_logits = matmul_qk / tf.math.sqrt(
            tf.cast(self.depth, tf.float32)
        )


        if mask is not None:
            # stack the mask so it can be applied to each head
            mask = tf.stack([mask for _ in range(self.num_heads)], axis=1)

            # we want -inf where mask is 1 because of the softmax
            scaled_attention_logits += mask * -1e9

        # Apply softmax to turn the attention scores into probabilities
        attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)

        output = tf.matmul(attention_weights, v)

        return output

    def __call__(self, query, value, key=None, mask=None):
        key = value if key is None else key

        query = self.wq(query)
        key = self.wk(key)
        value = self.wv(value)

        query = self._split_heads(query)
        key = self._split_heads(key)
        value = self._split_heads(value)

        scaled_attention = self._scaled_dot_product_attention(query, key, value,
 mask)

        scaled_attention = einops.rearrange(
            scaled_attention, "batch heads seq depth -> batch seq (heads depth)"
        )

        output = self.wo(scaled_attention)

        return output
```

```python
import tensorflow as tf

from helpers.conv_2d import Conv2D
from helpers.group_norm import GroupNorm


class ResidualBlock(tf.Module):
    def __init__(
        self,
        input_depth,
        output_depth,
        kernel_size,
        group_norm_num_groups,
        resblock_size=2,
    ):
        """Initializes the ResidualBlock class

        Args:
            input_depth (int): the number of input channels
            output_depth (int): the number of output channels
            kernel_size (list): the kernel size
            group_norm_num_groups (int): the number of groups to split the channels into
            resblock_size (int, optional): the number of residual blocks to stack
            inbetween skip connections. Defaults to 2.
        """
        self.resblock_size = resblock_size

        self.shortcut_conv = Conv2D(input_depth, output_depth, [1, 1])

        self.conv_layers = []
        for _ in range(self.resblock_size):
            self.conv_layers.append(
                Conv2D(input_depth, output_depth, kernel_size)
            )
            input_depth = output_depth

        self.group_norm = GroupNorm(group_norm_num_groups, output_depth)

    def __call__(self, x: tf.Tensor) -> tf.Tensor:
        shortcut = self.shortcut_conv(x)
        for conv_layer in self.conv_layers:
            x = conv_layer(x)
            x = self.group_norm(x)
            x = tf.nn.relu(x)
        return x + shortcut
```

```
import tempfile

import tensorflow as tf

from helpers.embed_to_vocab_file import EmbedToVocabFile
from helpers.positional_encoding import PositionalEncoding
from helpers.tokenizer import Tokenizer
from modules.linear import Linear
from modules.transformer_decoder_block import TransformerDecoderBlock


class TransformerDecoder(tf.Module):
    """Transformer Decoder.

    This is an implementation of a transformer decoder as described
    in the paper "Attention is all you Need" (Vaswani et al., 2017).

    Args:
        num_embedding: Number of embeddings to use.
        embedding_depth: Depth of each embedding.
        num_word_to_tokenize: Number of words to tokenize.
        num_heads: Number of attention heads.
        model_dim: Size of each attention head for value, query, and queue.
        ffn_dim: Size of the hidden layer in the feed-forward network.
        num_blocks: Number of transformer decoder blocks.
        input_text: Text to use for training. If None, training will not be possible.
        vocab_file: Path to the vocab file. If None, a temporary file will be created.
        dropout: Dropout probability.

    Call arguments:
        input: Input `Tensor` of shape `(B, seq_len)` during training and untokenized `(B, 1)` during inference.

    Returns:
        Output `Tensor` of shape `(B, seq_len, vocab_size)` during training and untokenized `(B, 1)` during inference.
    """

    def __init__(
        self,
        context_length,
        num_heads,
        model_dim,
        ffn_dim,
        num_blocks,
        input_text=None,
        vocab_file=None,
        dropout_prob=0.1,
    ):
        self.tokenizer = Tokenizer(context_length, False)

        self.input_text = input_text
        self.context_length = context_length
        if vocab_file is None:
            self.vocab_file = self._create_vocab_file(input_text)
        else:
            self.vocab_file = tf.io.gfile.GFile(vocab_file, "r")

        self.embedder = EmbedToVocabFile(self.vocab_file, model_dim)
        self.positional_encoding = PositionalEncoding(context_length, model_dim)

        self.layers = [
            TransformerDecoderBlock(num_heads, model_dim, ffn_dim, dropout_prob)
            for _ in range(num_blocks)
        ]

        self.vocab_size = self.embedder.get_vocab_size()
        self.linear = Linear(model_dim, self.vocab_size)
```

```
    def get_vocab_file(self):
        return self.vocab_file

    def get_tokens_and_targets(self):
        tokenized_text = self.tokenizer(self.input_text)

        tokenized_targets = tokenized_text[:, 1:]
        tokenized_text = tokenized_text[:, :-1]

        targets = self.embedder.tokens_to_ids(tokenized_targets)

        return tokenized_text, targets

    def decode(self, logits):
        return self.embedder.decode(logits)

    def predict(self, input_text):
        len_input = len(input_text.split())

        output_index = len_input
        output = ""
        for _ in range(self.context_length - len_input):
            tokenized_text = self.tokenizer(input_text)

            logits = self.__call__(tokenized_text, training=False)

            decoded_logits = self.decode(logits)

            next_word = decoded_logits[:, output_index - 1 : output_index]

            next_word_decoded = next_word[-1][0].numpy().decode("utf-8")

            output_index += 1

            if next_word_decoded == "<EOS>" or next_word_decoded == "<PAD>":
                break

            output += " " + next_word_decoded
            input_text += " " + next_word_decoded

        return output

    def _create_vocab_file(self, input_text):
        # Tokenize the contents of the file
        tokenized_text = self.tokenizer(input_text)

        # Flatten the tokenized_text tensor to 1D
        flattened_text = tf.reshape(tokenized_text, [-1])

        # Create a tensor of unique tokens
        unique_tokens, _ = tf.unique(flattened_text)

        # Write these unique tokens to a new vocab file
        vocab_file = tempfile.NamedTemporaryFile(delete=False)
        with open(vocab_file.name, "w", encoding="utf-8") as vocab_file:
            for token in unique_tokens.numpy():
                vocab_file.write(f"{token.decode('utf-8')}")
                if token != unique_tokens[-1]:
                    vocab_file.write("\n")

        return vocab_file

    def __call__(self, input_tokens, training=True):
        causal_mask = tf.linalg.band_part(
            tf.ones((input_tokens.shape[1], input_tokens.shape[1])), 0, -1
```

```
        )
        # make the main diagonal 0
        causal_mask = causal_mask - tf.eye(input_tokens.shape[1])

        # stack causal mask for each batch resulting in shape (B, seq_len, seq_l
en)
        causal_mask = tf.stack([causal_mask for _ in range(input_tokens.shape[0]
)])

        pad_mask_vector = tf.cast(tf.equal(input_tokens, b"<PAD>"), tf.float32)

        # stack seq_len copies of the pad mask vector for each batch resulting i
n shape (B, seq_len, seq_len)
        pad_mask1 = tf.stack(
            [pad_mask_vector for _ in range(input_tokens.shape[1])], axis=1
        )

        # transpose the pad mask vector to get shape (B, seq_len, seq_len)
        pad_mask2 = tf.transpose(pad_mask1, [0, 2, 1])

        # logical or of the two pad masks, switch 2's to 1's
        pad_mask = tf.cast(pad_mask1 + pad_mask2, tf.bool)
        pad_mask = tf.cast(pad_mask, tf.float32)
        pad_mask = pad_mask1

        mask = tf.cast(causal_mask + pad_mask, tf.bool)
        mask = tf.cast(mask, tf.float32)

        embeddings = self.embedder(input_tokens)

        for layer in self.layers:
            embeddings = layer(embeddings, mask, training)

        output = self.linear(embeddings)

        return output
```

```
import tensorflow as tf

from helpers.group_norm import GroupNorm
from modules.mlp import MLP
from modules.multi_head_attention import MultiHeadAttention


class TransformerDecoderBlock(tf.Module):
    """Transformer Decoder Block.

    This is an implementation of a single transformer decoder block as described
    in the paper "Attention is all you Need" (Vaswani et al., 2017).

    This layer first applies masked multi-headed attention to the inputs, then applies a feed-forward network to the result.

    Args:
        num_heads: Number of attention heads.
        model_dim: Size of each attention head for value, query, and queue.
        ffn_dim: Size of the hidden layer in the feed-forward network.
        dropout: Dropout probability.

    Call arguments:
        inputs: Input `Tensor` of shape `(B, seq_len, model_dim)`.
        mask: Optional mask tensor of shape `(B, seq_len, seq_len)`.

    Returns:
        Output `Tensor` of shape `(B, seq_len, model_dim)`.
    """

    def __init__(self, num_heads, model_dim, ffn_dim, dropout_prob=0.1):
        self.dropout_prob = dropout_prob

        self.mha = MultiHeadAttention(num_heads, model_dim)
        self.ff = MLP(
            model_dim,
            model_dim,
            hidden_layer_width=ffn_dim,
            hidden_activation=tf.nn.relu,
        )
        # Split the model dimension into 5 groups for group normalization
        self.groupnorm1 = GroupNorm(model_dim // 4, model_dim, 1)
        self.groupnorm2 = GroupNorm(model_dim // 4, model_dim, 1)

    def __call__(self, inputs, mask=None, training=False):
        attn = self.mha(
            self.groupnorm1(inputs),
            self.groupnorm1(inputs),
            self.groupnorm1(inputs),
            mask,
        )
        if training:
            attn = tf.nn.dropout(attn, rate=self.dropout_prob)
        out1 = attn + inputs

        ffn_output = self.ff(self.groupnorm2(out1))
        if training:
            ffn_output = tf.nn.dropout(ffn_output, rate=self.dropout_prob)
        out2 = out1 + ffn_output

        return out2
```

```
from pathlib import Path

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
import tqdm
import yaml
from datasets import load_dataset

from helpers.adam import Adam
from modules.embed_classifier import EmbedClassifier


def train_batch_accuracy(classifier, train_text_batch, train_labels_batch):
    return tf.reduce_mean(
        tf.cast(
            tf.equal(
                classifier(train_text_batch).numpy().argmax(axis=1),
                train_labels_batch.numpy().reshape(-1),
            ),
            tf.float32,
        )
    )


def val_accuracy(classifier, val_text, val_labels):
    val_accuracy = 0
    val_text = tf.convert_to_tensor(val_text)
    for i in range(0, val_text.shape[0], val_text.shape[0] // 100):
        batch_indices = tf.range(i, i + val_text.shape[0] // 100)
        val_batch_text = tf.gather(val_text, batch_indices)
        val_batch_labels = tf.gather(val_labels, batch_indices)
        if i == 0:
            val_accuracy = tf.reduce_mean(
                tf.cast(
                    tf.equal(
                        classifier(val_batch_text).numpy().argmax(axis=1),
                        val_batch_labels.numpy().reshape(-1),
                    ),
                    tf.float32,
                )
            ) / (val_text.shape[0] // 100)
        else:
            val_accuracy += tf.reduce_mean(
                tf.cast(
                    tf.equal(
                        classifier(val_batch_text).numpy().argmax(axis=1),
                        val_batch_labels.numpy().reshape(-1),
                    ),
                    tf.float32,
                )
            ) / (val_text.shape[0] // 100)
    return val_accuracy.numpy()


def val_loss(
    classifier,
    val_text,
    val_labels,
    checkpoint_manager,
    minimum_val_loss,
    minimum_val_step,
    current_step,
):
    val_text = tf.convert_to_tensor(val_text)
    validation_loss = 0
```

```
    for i in range(0, val_text.shape[0], val_text.shape[0] // 100):
        batch_indices = tf.range(i, i + val_text.shape[0] // 100)
        val_batch_text = tf.gather(val_text, batch_indices)
        val_batch_labels = tf.gather(val_labels, batch_indices)
        validation_loss = tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(
                labels=tf.squeeze(val_batch_labels), logits=classifier(val_batch
_text)
            )
        )

        # average the validation loss over the batches
        if i == 0:
            validation_loss = validation_loss / (val_text.shape[0] // 100)
        else:
            validation_loss += validation_loss / (val_text.shape[0] // 100)

    if validation_loss < minimum_val_loss:
        minimum_val_loss = validation_loss
        minimum_val_step = current_step
        checkpoint_manager.save()

    return validation_loss, minimum_val_loss, minimum_val_step


def val_check(
    classifier,
    val_text,
    val_labels,
    checkpoint_manager,
    minimum_val_loss,
    minimum_val_step_num,
    current_step,
    val_check_rate,
    y_val_loss,
    y_val_accuracy,
    used_patience,
    current_val_loss,
    current_validation_accuracy,
    x_val_iterations,
):
    if current_step % val_check_rate == (val_check_rate - 1):
        (
            current_val_loss,
            minimum_val_loss,
            minimum_val_step_num,
        ) = val_loss(
            classifier,
            val_text,
            val_labels,
            checkpoint_manager,
            minimum_val_loss,
            minimum_val_step_num,
            current_step,
        )
        x_val_iterations = np.append(x_val_iterations, current_step)

        y_val_loss = np.append(y_val_loss, current_val_loss)

        current_validation_accuracy = val_accuracy(classifier, val_text, val_lab
els)

        y_val_accuracy = np.append(y_val_accuracy, current_validation_accuracy)
        used_patience = current_step - minimum_val_step_num
```

```
    return (
        used_patience,
        minimum_val_loss,
        minimum_val_step_num,
        current_validation_accuracy,
        current_val_loss,
        y_val_loss,
        y_val_accuracy,
        x_val_iterations,
    )


def test_accuracy(classifier, test_images, test_labels):
    test_accuracy = 0
    test_images = tf.convert_to_tensor(test_images)
    for i in range(0, test_images.shape[0], test_images.shape[0] // 100):
        batch_indices = tf.range(i, i + test_images.shape[0] // 100)
        test_batch_images = tf.gather(test_images, batch_indices)
        test_batch_labels = tf.gather(test_labels, batch_indices)
        if i == 0:
            test_accuracy = (
                tf.reduce_mean(
                    tf.cast(
                        tf.equal(
                            classifier(test_batch_images).numpy().argmax(axis=1)
,
                            test_batch_labels.numpy().reshape(-1),
                        ),
                        tf.float32,
                    )
                )
                / 100
            )
        else:
            test_accuracy += (
                tf.reduce_mean(
                    tf.cast(
                        tf.equal(
                            classifier(test_batch_images).numpy().argmax(axis=1)
,
                            test_batch_labels.numpy().reshape(-1),
                        ),
                        tf.float32,
                    )
                )
                / 100
            )
    return test_accuracy.numpy()


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

    config = yaml.safe_load(config_path.read_text())
    num_iters = config["learning"]["num_iters"]
    weight_decay = config["learning"]["weight_decay"]
    dropout_prob = config["learning"]["dropout_prob"]
    batch_size = config["learning"]["batch_size"]
    learning_patience = config["learning"]["learning_patience"]
    learning_rates = config["learning"]["learning_rates"]
    num_embeddings = config["learning"]["num_embeddings"]
    embedding_depth = config["learning"]["embedding_depth"]
    val_check_rate = config["learning"]["val_check_rate"]

    refresh_rate = config["display"]["refresh_rate"]
```

```
    num_hidden_layers = config["mlp"]["num_hidden_layers"]
    hidden_layer_width = config["mlp"]["hidden_layer_width"]

    num_word_to_tokenize = config["data"]["num_words_to_tokenize"]

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

    dataset = load_dataset("ag_news")
    train_and_val_labels = dataset["train"]["label"]
    train_and_val_text = dataset["train"]["text"]

    # use 10,000 training samples for validation
    train_labels = train_and_val_labels[:-10000]
    train_text = tf.convert_to_tensor(train_and_val_text[:-10000])
    val_labels = train_and_val_labels[-10000:]
    val_text = train_and_val_text[-10000:]

    minimum_val_step_num = 0
    current_val_loss = 0
    used_patience = 0
    current_val_loss = -1
    current_validation_accuracy = -1

    num_classes = 4
    embed_classifier = EmbedClassifier(
        num_embeddings,
        embedding_depth,
        num_word_to_tokenize,
        dropout_prob,
        num_hidden_layers,
        hidden_layer_width,
        num_classes,
    )

    # Used For Plotting
    y_train_batch_accuracy = np.array([])
    y_train_batch_loss = np.array([])
    y_val_accuracy = np.array([])
    y_val_loss = np.array([])
    x_train_loss_iterations = np.array([])
    x_train_accuracy_iterations = np.array([])
    x_val_iterations = np.array([])

    learning_rate_change_steps = np.array([])

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

    checkpoint = tf.train.Checkpoint(embed_classifier)
    checkpoint_manager = tf.train.CheckpointManager(
        checkpoint, "temp/checkpoints/classify_agnews", 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}")
```

```
    val_log = tqdm.tqdm(total=0, position=3, bar_format="{desc}")
    bar = tqdm.trange(num_iters, position=4)

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

    for i in bar:
        batch_indices = rng.uniform(
            shape=[batch_size], maxval=train_text.shape[0], dtype=tf.int32
        )
        with tf.GradientTape() as tape:
            train_text_batch = tf.gather(train_text, batch_indices)
            train_labels_batch = tf.gather(train_labels, batch_indices)

            current_train_batch_loss = tf.reduce_mean(
                tf.nn.sparse_softmax_cross_entropy_with_logits(
                    labels=tf.squeeze(train_labels_batch),
                    logits=embed_classifier(train_text_batch),
                )
            )

        # Print initial train batch loss
        if i == 0:
            print("\n\n\n\n")
            print(f"Initial Training Loss => {current_train_batch_loss:0.4f}")
            _, minimum_val_loss, minimum_val_step_num = val_loss(
                embed_classifier,
                val_text,
                val_labels,
                checkpoint_manager,
                np.inf,
                i,
                i,
            )
            current_validation_accuracy = val_accuracy(
                embed_classifier, val_text, val_labels
            )
            current_val_loss = minimum_val_loss

        grads = tape.gradient(
            current_train_batch_loss, embed_classifier.trainable_variables
        )

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

        (
            used_patience,
            minimum_val_loss,
            minimum_val_step_num,
            current_validation_accuracy,
            current_val_loss,
            y_val_loss,
            y_val_accuracy,
            x_val_iterations,
        ) = val_check(
            embed_classifier,
            val_text,
            val_labels,
            checkpoint_manager,
            minimum_val_loss,
            minimum_val_step_num,
            i,
            val_check_rate,
```

```
            y_val_loss,
            y_val_accuracy,
            used_patience,
            current_val_loss,
            current_validation_accuracy,
            x_val_iterations,
        )

        current_train_batch_loss = current_train_batch_loss.numpy()
        y_train_batch_loss = np.append(y_train_batch_loss, current_train_batch_l
oss)
        x_train_loss_iterations = np.append(x_train_loss_iterations, i)

        if i % refresh_rate == (refresh_rate - 1):
            current_batch_accuracy = train_batch_accuracy(
                embed_classifier, train_text_batch, train_labels_batch
            )
            x_train_accuracy_iterations = np.append(x_train_accuracy_iterations,
 i)
            y_train_batch_accuracy = np.append(
                y_train_batch_accuracy, current_batch_accuracy
            )

            learning_rates_left = len(learning_rates) - learning_rate_index
            patience_left = learning_patience - used_patience
            overall_description = (
                f"Minimum Val Loss => {minimum_val_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_batch_loss:0.4f}    "
                + f"Train Accuracy => {current_batch_accuracy:0.4f}    "
            )
            train_log.set_description_str(train_description)
            train_log.update(refresh_rate)

            val_description = (
                f"Val Loss => {current_val_loss:0.4f}    "
                + f"Val Accuracy => {current_validation_accuracy:0.4f}    "
            )
            val_log.set_description_str(val_description)
            val_log.update(refresh_rate)

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

            # if the validation loss has not improved for learning_patience
            if (
                current_val_loss > minimum_val_loss
                and i - minimum_val_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)
                minimum_val_step_num = i
                checkpoint_manager.restore_or_initialize()

    checkpoint_manager.restore_or_initialize()
```

```
    current_validation_accuracy = val_accuracy(embed_classifier, val_text, val_l
abels)

    fig, ax = plt.subplots(2, 1)

    ax[0].plot(
        x_train_accuracy_iterations, y_train_batch_accuracy, label="Train Accuracy"
    )
    ax[0].plot(x_val_iterations, y_val_accuracy, label="Val Accuracy")
    # plot vertical line on learning rate change
    for learning_rate_change_step in learning_rate_change_steps:
        ax[0].axvline(x=learning_rate_change_step, color="black", linestyle="dashe
d")
    ax[0].set_xlabel("Iterations")
    ax[0].set_ylabel("Accuracy")
    ax[0].legend()

    ax[1].semilogy(
        x_train_loss_iterations, y_train_batch_loss, label="Train Batch Loss"
    )
    ax[1].semilogy(x_val_iterations, y_val_loss, label="Val Loss")
    for learning_rate_change_step in learning_rate_change_steps:
        ax[1].axvline(x=learning_rate_change_step, color="black", linestyle="dashe
d")
    ax[1].set_xlabel("Iterations")
    ax[1].set_ylabel("Loss")
    ax[1].legend()

    print("\n\n\n\n")
    print(f"Final Training Loss => {current_train_batch_loss:0.4f}")
    print(f"Stop Iteration => {i}")

    fig.suptitle(
        "Classify AGNews: Final Val Accuracy = " + f"{current_validation_accuracy:0.4f}"
    )

    # 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/agnews/classify_agnews_img_{file_index}.png").exists():
        file_index += 1
    fig.savefig(f"artifacts/agnews/classify_agnews_img_{file_index}.png")

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

    # save the model
    checkpoint_manager.save()
    config_path = Path(f"artifacts/agnews/model.yaml")
    config_path.write_text(yaml.dump(config))


def test(checkpoint_path: Path):
    if checkpoint_path is None:
        checkpoint_path = Path("temp/checkpoints/classify_agnews")

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

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

    config = yaml.safe_load(config_path.read_text())
    dropout_prob = config["learning"]["dropout_prob"]
    num_embeddings = config["learning"]["num_embeddings"]
```

```
    embedding_depth = config["learning"]["embedding_depth"]

    num_hidden_layers = config["mlp"]["num_hidden_layers"]
    hidden_layer_width = config["mlp"]["hidden_layer_width"]

    num_word_to_tokenize = config["data"]["num_words_to_tokenize"]

    num_classes = 4
    embed_classifier = EmbedClassifier(
        num_embeddings,
        embedding_depth,
        num_word_to_tokenize,
        dropout_prob,
        num_hidden_layers,
        hidden_layer_width,
        num_classes,
    )

    checkpoint = tf.train.Checkpoint(embed_classifier)
    checkpoint.restore(tf.train.latest_checkpoint(checkpoint_path))

    dataset = load_dataset("ag_news")
    test_labels = dataset["test"]["label"]
    test_text = dataset["test"]["text"]

    test_text = tf.convert_to_tensor(test_text)
    test_labels = tf.convert_to_tensor(test_labels)

    test_accuracy_value = test_accuracy(embed_classifier, test_text, test_labels
)

    print(f"Test Accuracy => {test_accuracy_value:0.4f}")
```

```
from pathlib import Path

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

from helpers.adam import Adam
from helpers.augment_data import AugmentData
from helpers.load_pickle_data import load_pickle_data
from modules.classifier import Classifier

from sklearn.metrics import top_k_accuracy_score


def train_batch_accuracy(classifier, train_images_batch, train_labels_batch):
    return tf.reduce_mean(
        tf.cast(
            tf.equal(
                classifier(train_images_batch).numpy().argmax(axis=1),
                train_labels_batch.numpy().reshape(-1),
            ),
            tf.float32,
        )
    )


def val_accuracy(classifier, val_images, val_labels):
    val_accuracy = 0
    for i in range(0, val_images.shape[0], val_images.shape[0] // 100):
        batch_indices = tf.range(i, i + val_images.shape[0] // 100)
        val_batch_images = tf.gather(val_images, batch_indices)
        val_batch_labels = tf.gather(val_labels, batch_indices)
        if i == 0:
            val_accuracy = tf.reduce_mean(
                tf.cast(
                    tf.equal(
                        classifier(val_batch_images).numpy().argmax(axis=1),
                        val_batch_labels.numpy().reshape(-1),
                    ),
                    tf.float32,
                )
            ) / (val_images.shape[0] // 100)
        else:
            val_accuracy += (
                tf.reduce_mean(
                    tf.cast(
                        tf.equal(
                            classifier(val_batch_images).numpy().argmax(axis=1),
                            val_batch_labels.numpy().reshape(-1),
                        ),
                        tf.float32,
                    )
                )
                / (val_images.shape[0] // 100)
            )
    return val_accuracy.numpy()


def test_accuracy(classifier, test_images, test_labels):
    return tf.reduce_mean(
        tf.cast(
            tf.equal(
                classifier(test_images).numpy().argmax(axis=1),
                test_labels.numpy().reshape(-1),
```

```
            ),
            tf.float32,
        )
    ).numpy()


def top_5_test_accuracy(classifier, test_images, test_labels):
    return top_k_accuracy_score(
        test_labels.numpy().reshape(-1), classifier(test_images).numpy(), k=5
    )


def val_loss(
    classifier,
    val_images,
    val_labels,
    checkpoint_manager,
    minimum_val_loss,
    minimum_val_step,
    current_step,
):
    validation_loss = 0

    for i in range(0, val_images.shape[0], val_images.shape[0] // 100):
        batch_indices = tf.range(i, i + val_images.shape[0] // 100)
        val_batch_images = tf.gather(val_images, batch_indices)
        val_batch_labels = tf.gather(val_labels, batch_indices)
        validation_loss = tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(
                labels=tf.squeeze(val_batch_labels), logits=classifier(val_batch
_images)
            )
        )

        # average the validation loss over the batches
        if i == 0:
            validation_loss = validation_loss / (val_images.shape[0] // 100)
        else:
            validation_loss += validation_loss / (val_images.shape[0] // 100)

    if validation_loss < minimum_val_loss:
        minimum_val_loss = validation_loss
        minimum_val_step = current_step
        checkpoint_manager.save()

    return validation_loss, minimum_val_loss, minimum_val_step


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

    config = yaml.safe_load(config_path.read_text())
    resblock_size = config["cnn"]["resblock_size"]
    pool_size = config["cnn"]["pool_size"]
    augmentation_multiplier = config["cnn"]["augmentation_multiplier"]
    layers = config["cnn"]["layers"]
    num_iters = config["learning"]["num_iters"]
    weight_decay = config["learning"]["weight_decay"]
    dropout_prob = config["learning"]["dropout_prob"]
    batch_size = config["learning"]["batch_size"]
    learning_patience = config["learning"]["learning_patience"]
    learning_rates = config["learning"]["learning_rates"]
    refresh_rate = config["display"]["refresh_rate"]
    num_hidden_layers = config["mlp"]["num_hidden_layers"]
    hidden_layer_width = config["mlp"]["hidden_layer_width"]
```

```
    layer_depths = [layer["depth"] for layer in layers]
    kernel_sizes = [layer["kernel_size"] for layer in layers]
    group_norm_num_groups = [layer["group_norm_num_groups"] for layer in layers]

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

    train_and_val_labels, train_and_val_images = load_pickle_data(
        "data/cifar-10-batches-py/data_batch_1"
    )
    for i in range(2, 6):
        labels, images = load_pickle_data(f"data/cifar-10-batches-py/data_batch_{i}")
        train_and_val_labels = tf.concat([train_and_val_labels, labels], axis=0)
        train_and_val_images = tf.concat([train_and_val_images, images], axis=0)

    num_classes = 10
    train_labels = train_and_val_labels[:-10000]
    train_images = train_and_val_images[:-10000]
    val_labels = train_and_val_labels[-10000:]
    val_images = train_and_val_images[-10000:]

    test_labels, test_images = load_pickle_data("data/cifar-10-batches-py/test_batch")

    num_samples = train_images.shape[0]
    input_depth = train_images.shape[-1]
    classifier = Classifier(
        input_depth,
        layer_depths,
        kernel_sizes,
        num_classes,
        train_images.shape[1],
        resblock_size,
        pool_size,
        dropout_prob,
        group_norm_num_groups,
        num_hidden_layers,
        hidden_layer_width,
    )

    minimum_val_step_num = 0
    current_val_loss = 0

    # Used For Plotting
    y_train_batch_accuracy = np.array([])
    y_train_batch_loss = np.array([])
    y_val_accuracy = np.array([])
    y_val_loss = np.array([])
    x_loss_iterations = np.array([])
    x_accuracy_iterations = np.array([])

    learning_rate_change_steps = np.array([])

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

    checkpoint = tf.train.Checkpoint(classifier)
    checkpoint_manager = tf.train.CheckpointManager(
        checkpoint, "temp/checkpoints/classify_numbers", 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}")
    val_log = tqdm.tqdm(total=0, position=3, bar_format="{desc}")
    bar = tqdm.trange(num_iters, position=4)

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

    augment_data = AugmentData(augmentation_multiplier)
    for i in bar:
        batch_indices = rng.uniform(
            shape=[batch_size], maxval=num_samples, dtype=tf.int32
        )
        with tf.GradientTape() as tape:
            train_images_batch = tf.gather(train_images, batch_indices)
            train_labels_batch = tf.gather(train_labels, batch_indices)

            train_labels_batch, train_images_batch = augment_data(
                train_labels_batch, train_images_batch
            )
            current_train_batch_loss = tf.reduce_mean(
                tf.nn.sparse_softmax_cross_entropy_with_logits(
                    labels=tf.squeeze(train_labels_batch),
                    logits=classifier(train_images_batch),
                )
            )

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

        grads = tape.gradient(current_train_batch_loss, classifier.trainable_var
iables)

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

        if i % refresh_rate == (refresh_rate - 1):
            (
                current_val_loss,
                minimum_val_loss,
                minimum_val_step_num,
            ) = val_loss(
                classifier,
                val_images,
                val_labels,
                checkpoint_manager,
                minimum_val_loss,
                minimum_val_step_num,
                i,
            )

            y_val_loss = np.append(y_val_loss, current_val_loss)
            current_train_batch_loss = current_train_batch_loss.numpy()
            y_train_batch_loss = np.append(y_train_batch_loss, current_train_bat
ch_loss)
            x_loss_iterations = np.append(x_loss_iterations, i)

            current_batch_accuracy = train_batch_accuracy(
```

```
                classifier, train_images_batch, train_labels_batch
            )
            current_validation_accuracy = val_accuracy(
                classifier, val_images, val_labels
            )

            x_accuracy_iterations = np.append(x_accuracy_iterations, i)
            y_train_batch_accuracy = np.append(
                y_train_batch_accuracy, current_batch_accuracy
            )
            y_val_accuracy = np.append(y_val_accuracy, current_validation_accura
cy)

            learning_rates_left = len(learning_rates) - learning_rate_index
            used_patience = i - minimum_val_step_num
            patience_left = learning_patience - used_patience
            overall_description = (
                f"Minimum Val Loss => {minimum_val_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_batch_loss:0.4f}    "
                + f"Train Accuracy => {current_batch_accuracy:0.4f}    "
            )
            train_log.set_description_str(train_description)
            train_log.update(refresh_rate)

            val_description = (
                f"Val Loss => {current_val_loss:0.4f}    "
                + f"Val Accuracy => {current_validation_accuracy:0.4f}    "
            )
            val_log.set_description_str(val_description)
            val_log.update(refresh_rate)

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

            # if the validation loss has not improved for learning_patience
            if (
                current_val_loss > minimum_val_loss
                and i - minimum_val_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)
                minimum_val_step_num = i
                checkpoint_manager.restore_or_initialize()

    checkpoint_manager.restore_or_initialize()

    fig, ax = plt.subplots(2, 1)

    ax[0].plot(x_accuracy_iterations, y_train_batch_accuracy, label="Train Accuracy
")
    ax[0].plot(x_accuracy_iterations, y_val_accuracy, label="Val Accuracy")
    # plot vertical line on learning rate change
    for learning_rate_change_step in learning_rate_change_steps:
        ax[0].axvline(x=learning_rate_change_step, color="black", linestyle="dashe
```

```
d")
    ax[0].set_xlabel("Iterations")
    ax[0].set_ylabel("Accuracy")
    ax[0].legend()

    ax[1].semilogy(x_loss_iterations, y_train_batch_loss, label="Train Batch Loss")
    ax[1].semilogy(x_loss_iterations, y_val_loss, label="Val Loss")
    for learning_rate_change_step in learning_rate_change_steps:
        ax[1].axvline(x=learning_rate_change_step, color="black", linestyle="dashe
d")
    ax[1].set_xlabel("Iterations")
    ax[1].set_ylabel("Loss")
    ax[1].legend()

    print("\n\n\n\n")
    print(f"Final Training Loss => {current_train_batch_loss:0.4f}")
    print(f"Stop Iteration => {i}")

    final_test_accuracy = test_accuracy(classifier, test_images, test_labels)
    final_top_5_test_accuracy = top_5_test_accuracy(
        classifier, test_images, test_labels
    )
    print(f"Test Accuracy => {final_test_accuracy:0.4f}")
    print(f"Top 5 Test Accuracy => {final_top_5_test_accuracy:0.4f}")
    fig.suptitle(
        "Classify Cifar10: Test Accuracy = "
        + str(final_test_accuracy)
        + "\nTop 5 Test Accuracy = "
        + str(final_top_5_test_accuracy)
    )

    # 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/classify_cifar10_img_{file_index}.png").exists():
        file_index += 1
    fig.savefig(f"artifacts/classify_cifar10_img_{file_index}.png")

    # Save the config file as a yaml under the same name as the image
    config_path = Path(f"artifacts/classify_cifar10_img_{file_index}.yaml")
    config_path.write_text(yaml.dump(config))
```

```python
from pathlib import Path

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

from helpers.adam import Adam
from helpers.augment_data import AugmentData
from helpers.load_pickle_data import load_pickle_data
from modules.classifier import Classifier

from sklearn.metrics import top_k_accuracy_score


def train_batch_accuracy(classifier, train_images_batch, train_labels_batch):
    return tf.reduce_mean(
        tf.cast(
            tf.equal(
                classifier(train_images_batch).numpy().argmax(axis=1),
                train_labels_batch.numpy().reshape(-1),
            ),
            tf.float32,
        )
    )


def val_accuracy(classifier, val_images, val_labels):
    return tf.reduce_mean(
        tf.cast(
            tf.equal(
                classifier(val_images).numpy().argmax(axis=1),
                val_labels.numpy().reshape(-1),
            ),
            tf.float32,
        )
    )


def test_accuracy(classifier, test_images, test_labels):
    return tf.reduce_mean(
        tf.cast(
            tf.equal(
                classifier(test_images).numpy().argmax(axis=1),
                test_labels.numpy().reshape(-1),
            ),
            tf.float32,
        )
    ).numpy()


def top_5_test_accuracy(classifier, test_images, test_labels):
    return top_k_accuracy_score(
        test_labels.numpy().reshape(-1), classifier(test_images).numpy(), k=5
    )


def val_loss(
    classifier,
    val_images,
    val_labels,
    checkpoint_manager,
    minimum_val_loss,
    minimum_val_step,
    current_step,
```

```
):
    validation_loss = tf.reduce_mean(
        tf.nn.sparse_softmax_cross_entropy_with_logits(
            labels=tf.squeeze(val_labels), logits=classifier(val_images)
        )
    )

    if validation_loss < minimum_val_loss:
        minimum_val_loss = validation_loss
        minimum_val_step = current_step
        checkpoint_manager.save()

    return validation_loss, minimum_val_loss, minimum_val_step


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

    config = yaml.safe_load(config_path.read_text())
    resblock_size = config["cnn"]["resblock_size"]
    pool_size = config["cnn"]["pool_size"]
    augmentation_multiplier = config["cnn"]["augmentation_multiplier"]
    layers = config["cnn"]["layers"]
    num_iters = config["learning"]["num_iters"]
    weight_decay = config["learning"]["weight_decay"]
    dropout_prob = config["learning"]["dropout_prob"]
    batch_size = config["learning"]["batch_size"]
    learning_patience = config["learning"]["learning_patience"]
    learning_rates = config["learning"]["learning_rates"]
    refresh_rate = config["display"]["refresh_rate"]
    num_hidden_layers = config["mlp"]["num_hidden_layers"]
    hidden_layer_width = config["mlp"]["hidden_layer_width"]

    layer_depths = [layer["depth"] for layer in layers]
    kernel_sizes = [layer["kernel_size"] for layer in layers]
    group_norm_num_groups = [layer["group_norm_num_groups"] for layer in layers]

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

    train_and_val_labels, train_and_val_images = load_pickle_data(
        "data/cifar-100-python/train", "fine_labels"
    )

    num_classes = 100
    train_labels = train_and_val_labels[:-10000]
    train_images = train_and_val_images[:-10000]
    val_labels = train_and_val_labels[-10000:]
    val_images = train_and_val_images[-10000:]

    test_labels, test_images = load_pickle_data(
        "data/cifar-100-python/test", "fine_labels"
    )

    num_samples = train_images.shape[0]
    input_depth = train_images.shape[-1]
    classifier = Classifier(
        input_depth,
        layer_depths,
        kernel_sizes,
        num_classes,
        train_images.shape[1],
        resblock_size,
        pool_size,
        dropout_prob,
```

```
        group_norm_num_groups,
        num_hidden_layers,
        hidden_layer_width,
    )

    minimum_val_step_num = 0
    current_val_loss = 0

    # Used For Plotting
    y_train_batch_accuracy = np.array([])
    y_train_batch_loss = np.array([])
    y_val_accuracy = np.array([])
    y_val_loss = np.array([])
    x_loss_iterations = np.array([])
    x_accuracy_iterations = np.array([])

    learning_rate_change_steps = np.array([])

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

    checkpoint = tf.train.Checkpoint(classifier)
    checkpoint_manager = tf.train.CheckpointManager(
        checkpoint, "temp/checkpoints/classify_numbers", 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}")
    val_log = tqdm.tqdm(total=0, position=3, bar_format="{desc}")
    bar = tqdm.trange(num_iters, position=4)

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

    augment_data = AugmentData(augmentation_multiplier)
    for i in bar:
        batch_indices = rng.uniform(
            shape=[batch_size], maxval=num_samples, dtype=tf.int32
        )
        with tf.GradientTape() as tape:
            train_images_batch = tf.gather(train_images, batch_indices)
            train_labels_batch = tf.gather(train_labels, batch_indices)

            train_labels_batch, train_images_batch = augment_data(
                train_labels_batch, train_images_batch
            )
            current_train_batch_loss = tf.reduce_mean(
                tf.nn.sparse_softmax_cross_entropy_with_logits(
                    labels=tf.squeeze(train_labels_batch),
                    logits=classifier(train_images_batch),
                )
            )

        # Print initial train batch loss
        if i == 0:
```

```
            print("\n\n\n\n")
            print(f"Initial Training Loss => {current_train_batch_loss:0.4f}")
            minimum_val_loss = current_train_batch_loss

        grads = tape.gradient(current_train_batch_loss, classifier.trainable_var
iables)

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

        (
            current_val_loss,
            minimum_val_loss,
            minimum_val_step_num,
        ) = val_loss(
            classifier,
            val_images,
            val_labels,
            checkpoint_manager,
            minimum_val_loss,
            minimum_val_step_num,
            i,
        )

        y_val_loss = np.append(y_val_loss, current_val_loss)
        current_train_batch_loss = current_train_batch_loss.numpy()
        y_train_batch_loss = np.append(y_train_batch_loss, current_train_batch_l
oss)
        x_loss_iterations = np.append(x_loss_iterations, i)

        if i % refresh_rate == (refresh_rate - 1):
            current_batch_accuracy = train_batch_accuracy(
                classifier, train_images_batch, train_labels_batch
            )
            current_validation_accuracy = val_accuracy(
                classifier, val_images, val_labels
            )

            x_accuracy_iterations = np.append(x_accuracy_iterations, i)
            y_train_batch_accuracy = np.append(
                y_train_batch_accuracy, current_batch_accuracy
            )
            y_val_accuracy = np.append(y_val_accuracy, current_validation_accura
cy)

            learning_rates_left = len(learning_rates) - learning_rate_index
            used_patience = i - minimum_val_step_num
            patience_left = learning_patience - used_patience
            overall_description = (
                f"Minimum Val Loss => {minimum_val_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_batch_loss:0.4f}    "
                + f"Train Accuracy => {current_batch_accuracy:0.4f}    "
            )
            train_log.set_description_str(train_description)
            train_log.update(refresh_rate)

            val_description = (
                f"Val Loss => {current_val_loss:0.4f}    "
                + f"Val Accuracy => {current_validation_accuracy:0.4f}    "
            )
```

```
            val_log.set_description_str(val_description)
            val_log.update(refresh_rate)

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

            # if the validation loss has not improved for learning_patience
            if (
                current_val_loss > minimum_val_loss
                and i - minimum_val_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)
                minimum_val_step_num = i
                checkpoint_manager.restore_or_initialize()

    checkpoint_manager.restore_or_initialize()

    fig, ax = plt.subplots(2, 1)

    ax[0].plot(x_accuracy_iterations, y_train_batch_accuracy, label="Train Accuracy
")
    ax[0].plot(x_accuracy_iterations, y_val_accuracy, label="Val Accuracy")
    # plot vertical line on learning rate change
    for learning_rate_change_step in learning_rate_change_steps:
        ax[0].axvline(x=learning_rate_change_step, color="black", linestyle="dashe
d")
    ax[0].set_xlabel("Iterations")
    ax[0].set_ylabel("Accuracy")
    ax[0].legend()

    ax[1].semilogy(x_loss_iterations, y_train_batch_loss, label="Train Batch Loss")
    ax[1].semilogy(x_loss_iterations, y_val_loss, label="Val Loss")
    for learning_rate_change_step in learning_rate_change_steps:
        ax[1].axvline(x=learning_rate_change_step, color="black", linestyle="dashe
d")
    ax[1].set_xlabel("Iterations")
    ax[1].set_ylabel("Loss")
    ax[1].legend()

    print("\n\n\n\n")
    print(f"Final Training Loss => {current_train_batch_loss:0.4f}")
    print(f"Stop Iteration => {i}")

    final_test_accuracy = test_accuracy(classifier, test_images, test_labels)
    final_top_5_test_accuracy = top_5_test_accuracy(
        classifier, test_images, test_labels
    )
    print(f"Test Accuracy => {final_test_accuracy:0.4f}")
    print(f"Top 5 Test Accuracy => {final_top_5_test_accuracy:0.4f}")
    fig.suptitle(
        "Classify Cifar100: Test Accuracy = "
        + str(final_test_accuracy)
        + "\nTop 5 Test Accuracy = "
        + str(final_top_5_test_accuracy)
    )

    # 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/classify_cifar100_img_{file_index}.png").exists():
```

```
        file_index += 1
    fig.savefig(f"artifacts/classify_cifar100_img_{file_index}.png")

    # Save the config file as a yaml under the same name as the image
    config_path = Path(f"artifacts/classify_cifar100_img_{file_index}.yaml")
    config_path.write_text(yaml.dump(config))
```

```
import tempfile
from pathlib import Path

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

from helpers.adam import Adam
from modules.transformer_decoder import TransformerDecoder


def train_batch_accuracy(logits, labels):
    return tf.reduce_mean(
        tf.cast(
            tf.equal(
                logits.numpy().argmax(axis=2).reshape(-1),
                labels.numpy().reshape(-1),
            ),
            tf.float32,
        )
    )


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

    # HYPERPARAMETERS
    config = yaml.safe_load(config_path.read_text())
    refresh_rate = config["display"]["refresh_rate"]
    batch_size = config["learning"]["batch_size"]
    learning_patience = config["learning"]["learning_patience"]
    learning_rates = config["learning"]["learning_rates"]
    num_iters = config["learning"]["num_iters"]
    weight_decay = config["learning"]["weight_decay"]
    context_length = config["data"]["context_length"]
    num_heads = config["transformer"]["num_heads"]
    model_dim = config["transformer"]["model_dim"]
    ffn_dim = config["transformer"]["ffn_dim"]
    num_blocks = config["transformer"]["num_blocks"]

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

    with open("data/who_bites_who.txt", "r", encoding="utf-8") as file:
        input_text = file.read()

    transformer_decoder = TransformerDecoder(
        context_length,
        num_heads,
        model_dim,
        ffn_dim,
        num_blocks,
        input_text,
    )

    text, targets = transformer_decoder.get_tokens_and_targets()

    used_patience = 0
    minimum_train_loss = np.inf
    minimum_loss_step_num = 0

    # Used For Plotting
```

```
    y_train_accuracy = np.array([])
    y_train_batch_loss = np.array([])
    x_train_loss_iterations = np.array([])
    x_train_accuracy_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 "who_bites_who_" 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("who_bites_who_"):
            break

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

    checkpoint = tf.train.Checkpoint(transformer_decoder)
    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 transformer_decoder.trainable_variables
        ]
    )
    print(f"\nNumber of Parameters => {num_of_parameters}")

    for i in bar:
        batch_indices = rng.uniform(
            shape=[batch_size], maxval=text.shape[0], dtype=tf.int32
        )

        with tf.GradientTape() as tape:
            input_tokens_batch = tf.gather(text, batch_indices)
            targets_batch = tf.gather(targets, batch_indices)

            labels = targets_batch
            logits = transformer_decoder(input_tokens_batch)
            current_train_loss = tf.reduce_mean(
                tf.nn.sparse_softmax_cross_entropy_with_logits(
                    labels=labels,
                    logits=logits,
                )
            )

        # 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, transformer_decoder.trainable_variables
        )

        adam.apply_gradients(zip(grads, transformer_decoder.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):
            current_batch_accuracy = train_batch_accuracy(logits, labels)
            x_train_accuracy_iterations = np.append(x_train_accuracy_iterations,
 i)
            y_train_accuracy = np.append(y_train_accuracy, current_batch_accurac
y)

            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}    "
                + f"Train Accuracy => {current_batch_accuracy: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()

    # get the vocab file from
```

```
    vocab_file = transformer_decoder.get_vocab_file()
    with open(vocab_file.name, "r", encoding="utf−8") as file:
        vocab_file_contents = file.read()

    # save a copy to the artifacts directory
    vocab_file_copy = Path("artifacts/who_bites_who/model/vocab.txt")
    with open(vocab_file_copy, "w", encoding="utf−8") as file:
        file.write(vocab_file_contents)

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

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

    batch_indices = rng.uniform(
        shape=[batch_size], maxval=text.shape[0], dtype=tf.int32
    )
    input_tokens_batch = tf.gather(text, batch_indices)
    targets_batch = tf.gather(targets, batch_indices)

    fig, ax = plt.subplots(2, 1)
    plt.subplots_adjust(hspace=0.5)

    ax[0].semilogy(x_train_loss_iterations, y_train_batch_loss, label="Training Los
s")
    for learning_rate_change_step in learning_rate_change_steps:
        ax[0].axvline(
            x=learning_rate_change_step,
            color="black",
            linestyle="dashed",
            label="Learning Rate Change",
        )
    ax[0].axvline(
        x=minimum_loss_step_num, color="red", linestyle="dashed", label="Minimum Lo
ss"
    )
    ax[0].set_xlabel("Iterations")
    ax[0].set_ylabel("Loss")

    ax[1].plot(x_train_accuracy_iterations, y_train_accuracy, label="Training Accura
cy")
    for learning_rate_change_step in learning_rate_change_steps:
        ax[1].axvline(
            x=learning_rate_change_step,
            color="black",
            linestyle="dashed",
            label="Learning Rate Change",
        )
    ax[1].axvline(
        x=minimum_loss_step_num, color="red", linestyle="dashed", label="Minimum Lo
ss"
    )
    ax[1].set_xlabel("Iterations")
    ax[1].set_ylabel("Accuracy")
    ax[1].legend(loc="lower left")

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

    labels = targets_batch
    logits = transformer_decoder(input_tokens_batch)
    final_train_accuracy = train_batch_accuracy(logits, labels)

    print(f"Final Training Accuracy => {final_train_accuracy:0.4f}")
```

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

    fig.suptitle(
        "Predict Who Bites Who: Final Train Accuracy = "
        + f"{final_train_accuracy:0.4f}"
    )

    # 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/who_bites_who/predict_who_bites_who_img_{file_index}.png"
    ).exists():
        file_index += 1
    fig.savefig(f"artifacts/who_bites_who/predict_who_bites_who_img_{file_index}.png")

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

    # save the model
    checkpoint_manager.save()
    config_path = Path(f"artifacts/who_bites_who/model/model.yaml")
    config_path.write_text(yaml.dump(config))


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

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

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

    config = yaml.safe_load(config_path.read_text())
    context_length = config["data"]["context_length"]
    num_heads = config["transformer"]["num_heads"]
    model_dim = config["transformer"]["model_dim"]
    ffn_dim = config["transformer"]["ffn_dim"]
    num_blocks = config["transformer"]["num_blocks"]

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

    vocab_file = Path("artifacts/who_bites_who/model/vocab.txt")

    transformer_decoder = TransformerDecoder(
        context_length,
        num_heads,
        model_dim,
        ffn_dim,
        num_blocks,
        vocab_file=vocab_file,
    )

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

    print("\n\n\n\nEnter 'exit' to exit")
    while 1:
        input_text = input("\n\nEnter a sentence: ")
```

```
        if input_text == "exit":
            break

        tokenized_text = transformer_decoder.predict(input_text)
        print(f"Bite Bot: " + tokenized_text)
```

```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()
```

```
import pytest


@pytest.mark.parametrize("augmentation_probability", [0.0, 0.5, 1.0])
def test_dimenionality(augmentation_probability):
    import tensorflow as tf

    from helpers.augment_data import AugmentData
    from helpers.load_pickle_data import load_pickle_data

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    augment_data = AugmentData(augmentation_probability)

    labels, images = load_pickle_data("data/cifar-10-batches-py/data_batch_1")

    augmented_labels, augmented_images = augment_data(labels, images)

    # check the image dimensions
    assert (
        tf.shape(augmented_images)[0].numpy()
        == tf.shape(images)[0].numpy()
        + augmentation_probability * tf.shape(images)[0].numpy() * 6
    )

    # check the label dimensions
    assert (
        tf.shape(augmented_labels)[0].numpy()
        == tf.shape(labels)[0].numpy()
        + augmentation_probability * tf.shape(labels)[0].numpy() * 6
    )


if __name__ == "__main__":
    pytest.main([__file__])
```

```
import pytest


@pytest.mark.parametrize("kernel_size", [[2, 2], [4, 4], [8, 8]])
@pytest.mark.parametrize("input_channels", [1, 3, 16])
@pytest.mark.parametrize("output_channels", [1, 3, 16])
def test_dimensionality(kernel_size, input_channels, output_channels):
    import tensorflow as tf
    from helpers.conv_2d import Conv2D

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    conv2d = Conv2D(input_channels, output_channels, kernel_size)

    a = rng.normal(shape=[1, 28, 28, input_channels])
    z = conv2d(a)

    tf.assert_equal(tf.shape(z)[-1], output_channels)


if __name__ == "__main__":
    pytest.main([__file__])
```

```python
import pytest


def test_dimensionality():
    import tensorflow as tf
    from helpers.group_norm import GroupNorm

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    group_norm = GroupNorm(4, 16)

    input = rng.normal(shape=[1, 32, 32, 16])
    output = group_norm(input)

    tf.assert_equal(tf.shape(output), tf.shape(input))


def test_mean_and_variance():
    import tensorflow as tf
    from helpers.group_norm import GroupNorm

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    group_norm = GroupNorm(4, 16)

    input = rng.normal(shape=[1, 32, 32, 16])
    output = group_norm(input)

    mean, variance = tf.nn.moments(output, axes=[1, 2, 3])

    assert mean[0].numpy() == pytest.approx(0.0, abs=1e-3)
    assert variance[0].numpy() == pytest.approx(1.0, abs=1e-3)


if __name__ == "__main__":
    pytest.main([__file__])
```

```python
import pytest


def test_dimensionality():
    from helpers.load_idx_data import load_idx_data

    train_images = load_idx_data("data/train-images-idx3-ubyte")
    train_labels = load_idx_data("data/train-labels-idx1-ubyte")
    test_labels = load_idx_data("data/t10k-labels-idx1-ubyte")
    test_images = load_idx_data("data/t10k-images-idx3-ubyte")

    assert train_images.shape == (60000, 28, 28, 1)
    assert train_labels.shape == (60000, 1)
    assert test_images.shape == (10000, 28, 28, 1)
    assert test_labels.shape == (10000, 1)


if __name__ == "__main__":
    pytest.main([__file__])
```

```
import pytest


def test_dimensionality():
    import tensorflow as tf

    from helpers.load_pickle_data import load_pickle_data

    train_and_val_labels, train_and_val_images = load_pickle_data(
        "data/cifar-10-batches-py/data_batch_1"
    )

    assert tf.shape(train_and_val_labels)[0] == 10000
    assert train_and_val_images.shape == tf.TensorShape([10000, 32, 32, 3])

    train_and_val_labels, train_and_val_images = load_pickle_data(
        "data/cifar-100-python/train", "fine_labels"
    )

    assert tf.shape(train_and_val_labels)[0] == [50000]
    assert train_and_val_images.shape == tf.TensorShape([50000, 32, 32, 3])


def test_labels():
    import tensorflow as tf

    from helpers.load_pickle_data import load_pickle_data

    train_and_val_labels, _ = load_pickle_data(
        "data/cifar-10-batches-py/data_batch_1"
    )

    assert tf.reduce_min(train_and_val_labels) == 0
    assert tf.reduce_max(train_and_val_labels) == 9


if __name__ == "__main__":
    pytest.main([__file__])
```

```
import pytest


@pytest.mark.parametrize(
    "sentence, expected_tokens",
    [
        (
            "Woah oh wee woo Wah wi wah woo",
            [[b"Woah", b"oh", b"wee", b"woo", b"Wah", b"wi", b"wah", b"woo"]],
        ),
        (
            "AHHHHHHH AHHHHHH AHH AHHHHHHH AHHHHHHHHHHHHHH AHH AHAHHH AHH
H AHHH AHH",
            [
                [
                    b"AHHHHHHH",
                    b"AHHHHHH",
                    b"AHH",
                    b"AHHHHHHH",
                    b"AHHHHHHHHHHHHHH",
                    b"AHH",
                    b"AHAHHH",
                    b"AHHH",
                ],
                [
                    b"AHHH",
                    b"AHH",
                    b"<PAD>",
                    b"<PAD>",
                    b"<PAD>",
                    b"<PAD>",
                    b"<PAD>",
                    b"<PAD>",
                ],
            ],
        ),
    ],
)
def test_tokenizer(sentence, expected_tokens):
    import tensorflow as tf

    from helpers.tokenizer import Tokenizer

    tokenizer = Tokenizer(8, False)

    tokens = tokenizer(sentence)

    assert tf.reduce_all(tf.equal(tokens, tf.convert_to_tensor(expected_tokens))
)


if __name__ == "__main__":
    pytest.main([__file__])
```

```
import pytest


def test_non_additivity():
    import tensorflow as tf

    from modules.basis_expansion import BasisExpansion

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    num_bases = 10

    basisExpansion = BasisExpansion(num_bases, num_inputs, num_outputs)

    a = rng.normal(shape=[1, num_inputs])
    b = rng.normal(shape=[1, num_inputs])

    case1 = basisExpansion(a + b)
    case2 = basisExpansion(a) + basisExpansion(b)

    tol = 2.22e-15 + 2.22e-15*tf.abs(case2)

    tf.debugging.Assert(
        tf.reduce_any(
            tf.greater(
                tf.abs(
                    case1 - case2
                    ),
                tol
                )
            ),
        [case1, case2],
        summarize=2
    )


def test_homogeneity():
    import tensorflow as tf

    from modules.basis_expansion import BasisExpansion

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    num_bases = 10
    num_test_cases = 100

    basisExpansion = BasisExpansion(num_bases, num_inputs, num_outputs)

    a = rng.normal(shape=[1, 1, num_inputs])
    b = rng.normal(shape=[num_test_cases, 1, 1])

    case1 = basisExpansion(a * b)
    case2 = basisExpansion(a) * b

    tol = 2.22e-15 + 2.22e-15*tf.abs(case2)

    tf.debugging.Assert(
        tf.reduce_any(
            tf.greater(
                tf.abs(
```

```
                    case1 - case2
                    ),
                tol
                )
            ),
        [case1, case2],
        summarize=2
    )


if __name__ == "__main__":
    pytest.main([__file__])
```

```python
import pytest


@pytest.mark.parametrize("layer_depths", [[10, 20], [20, 40]])
@pytest.mark.parametrize("kernel_sizes", [[[2, 2], [2, 2]], [[3, 3], [3, 3]]])
@pytest.mark.parametrize("num_classes", [10, 100])
@pytest.mark.parametrize("resblock_size", [1, 2, 3])
@pytest.mark.parametrize("group_norm_num_groups", [[2, 2], [2, 5]])
def test_dimensionality(
    layer_depths,
    kernel_sizes,
    num_classes,
    resblock_size,
    group_norm_num_groups,
):
    import tensorflow as tf

    from modules.classifier import Classifier

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    input_depth = 3
    input_size = 32
    dropout_prob = 0.5
    pool_size = 2

    num_hidden_layers = 1
    hidden_layer_width = 10

    classifier = Classifier(
        input_depth,
        layer_depths,
        kernel_sizes,
        num_classes,
        input_size,
        resblock_size,
        pool_size,
        dropout_prob,
        group_norm_num_groups,
        num_hidden_layers,
        hidden_layer_width,
    )

    input_data = rng.normal(shape=[1, input_size, input_size, input_depth])
    classified_data = classifier(input_data)

    tf.assert_equal(tf.shape(classified_data)[-1], num_classes)


if __name__ == "__main__":
    pytest.main([__file__])
```

```python
import pytest
import tensorflow as tf


@pytest.mark.parametrize(
    "text",
    [
        tf.constant(["professor curro likes", "reading about", "apples"]),
        tf.constant(["apples are a good fruit"]),
    ],
)
@pytest.mark.parametrize("num_embedding", [100, 200, 300])
@pytest.mark.parametrize("embedding_depth", [50, 100, 150])
@pytest.mark.parametrize("num_word_to_tokenize", [10, 20, 30])
@pytest.mark.parametrize("num_classes", [10, 20, 30])
def test_EmbedClassifier_call(
    text, num_embedding, embedding_depth, num_word_to_tokenize, num_classes
):
    from modules.embed_classifier import EmbedClassifier

    num_classes = 10
    embed_classifier = EmbedClassifier(
        num_embedding,
        embedding_depth,
        num_word_to_tokenize,
        dropout_prob=0.5,
        num_hidden_layers=3,
        hidden_layer_width=30,
        num_classes=num_classes,
    )

    embeddings = embed_classifier(text)

    assert embeddings.shape[0] == len(text)
    assert embeddings.shape[1] == num_classes


if __name__ == "__main__":
    pytest.main([__file__])
```

```python
import pytest


def test_additivity():
    import tensorflow as tf

    from modules.linear import Linear

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1

    linear = Linear(num_inputs, num_outputs)

    a = rng.normal(shape=[1, num_inputs])
    b = rng.normal(shape=[1, num_inputs])

    tf.debugging.assert_near(linear(a + b), linear(a) + linear(b), summarize=2)


def test_homogeneity():
    import tensorflow as tf

    from modules.linear import Linear

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    num_test_cases = 100

    linear = Linear(num_inputs, num_outputs)

    a = rng.normal(shape=[1, num_inputs])
    b = rng.normal(shape=[num_test_cases, 1])

    tf.debugging.assert_near(linear(a * b), linear(a) * b, summarize=2)


@pytest.mark.parametrize("num_outputs", [1, 16, 128])
def test_dimensionality(num_outputs):
    import tensorflow as tf

    from modules.linear import Linear

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10

    linear = Linear(num_inputs, num_outputs)

    a = rng.normal(shape=[1, num_inputs])
    z = linear(a)

    tf.assert_equal(tf.shape(z)[-1], num_outputs)


@pytest.mark.parametrize("bias", [True, False])
def test_trainable(bias):
    import tensorflow as tf

    from modules.linear import Linear
```

```
    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1

    linear = Linear(num_inputs, num_outputs, bias=bias)

    a = rng.normal(shape=[1, num_inputs])

    with tf.GradientTape() as tape:
        z = linear(a)
        loss = tf.math.reduce_mean(z**2)

    grads = tape.gradient(loss, linear.trainable_variables)

    for grad, var in zip(grads, linear.trainable_variables):
        tf.debugging.check_numerics(grad, message=f"{var.name}: ")
        tf.debugging.assert_greater(tf.math.abs(grad), 0.0)

    assert len(grads) == len(linear.trainable_variables)

    if bias:
        assert len(grads) == 2
    else:
        assert len(grads) == 1


@pytest.mark.parametrize(
    "a_shape, b_shape",
    [
        ([1000, 1000], [100, 100]),
        ([1000, 100], [100, 100]),
        ([100, 1000], [100, 100])
    ],
)
def test_init_properties(a_shape, b_shape):
    import tensorflow as tf

    from modules.linear import Linear

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs_a, num_outputs_a = a_shape
    num_inputs_b, num_outputs_b = b_shape

    linear_a = Linear(num_inputs_a, num_outputs_a, bias=False)
    linear_b = Linear(num_inputs_b, num_outputs_b, bias=False)

    std_a = tf.math.reduce_std(linear_a.w)
    std_b = tf.math.reduce_std(linear_b.w)

    tf.debugging.assert_less(std_a, std_b)


def test_bias():
    import tensorflow as tf

    from modules.linear import Linear

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    linear_with_bias = Linear(1, 1, bias=True)
```

```
    assert hasattr(linear_with_bias, "b")

    linear_with_bias = Linear(1, 1, bias=False)
    assert not hasattr(linear_with_bias, "b")


if __name__ == "__main__":
    pytest.main([__file__])
```

```
import pytest


def test_non_additivity():
    import tensorflow as tf

    from modules.mlp import MLP

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    num_hidden_layers = 10
    hidden_layer_width = 10
    relu = tf.nn.relu
    sigmoid = tf.nn.sigmoid

    mlp = MLP(
        num_inputs, num_outputs, num_hidden_layers, hidden_layer_width, relu, si
gmoid
    )

    a = rng.normal(shape=[1, num_inputs])
    b = rng.normal(shape=[1, num_inputs])

    case1 = mlp(a + b)
    case2 = mlp(a) + mlp(b)

    tol = 2.22e-15 + 2.22e-15 * tf.abs(case2)

    tf.debugging.Assert(
        tf.reduce_any(tf.greater(tf.abs(case1 - case2), tol)),
        [case1, case2],
        summarize=2,
    )


def test_non_homogeneity():
    import tensorflow as tf

    from modules.mlp import MLP

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    num_test_cases = 100
    num_hidden_layers = 10
    hidden_layer_width = 10
    relu = tf.nn.relu
    sigmoid = tf.nn.sigmoid

    mlp = MLP(
        num_inputs, num_outputs, num_hidden_layers, hidden_layer_width, relu, si
gmoid
    )

    a = rng.normal(shape=[1, 1, num_inputs])
    b = rng.normal(shape=[num_test_cases, 1, 1])

    case1 = mlp(a * b)
    case2 = mlp(a) * b

    tol = 2.22e-15 + 2.22e-15 * tf.abs(case2)
```

```python
    tf.debugging.Assert(
        tf.reduce_any(tf.greater(tf.abs(case1 - case2), tol)),
        [case1, case2],
        summarize=2,
    )


def test_additivity():
    import tensorflow as tf

    from modules.mlp import MLP

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    num_hidden_layers = 10
    hidden_layer_width = 10

    mlp = MLP(num_inputs, num_outputs, num_hidden_layers, hidden_layer_width)

    a = rng.normal(shape=[1, num_inputs])
    b = rng.normal(shape=[1, num_inputs])

    tf.debugging.assert_near(mlp(a + b), mlp(a) + mlp(b), summarize=2)


def test_homogeneity():
    import tensorflow as tf

    from modules.mlp import MLP

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    num_test_cases = 100
    num_hidden_layers = 10
    hidden_layer_width = 10

    mlp = MLP(num_inputs, num_outputs, num_hidden_layers, hidden_layer_width)

    a = rng.normal(shape=[1, num_inputs])
    b = rng.normal(shape=[num_test_cases, 1])

    tf.debugging.assert_near(mlp(a * b), mlp(a) * b, summarize=2)


def test_dimensionality():
    import tensorflow as tf

    from modules.mlp import MLP

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    num_hidden_layers = 10
    hidden_layer_width = 10

    mlp = MLP(num_inputs, num_outputs, num_hidden_layers, hidden_layer_width)
```

```
    a = rng.normal(shape=[1, num_inputs])
    z = mlp(a)

    tf.assert_equal(tf.shape(z)[-1], num_outputs)


def test_trainable():
    import tensorflow as tf

    from modules.mlp import MLP

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    num_hidden_layers = 10
    hidden_layer_width = 10

    mlp = MLP(num_inputs, num_outputs, num_hidden_layers, hidden_layer_width)

    a = rng.normal(shape=[1, num_inputs])

    with tf.GradientTape() as tape:
        z = mlp(a)
        loss = tf.reduce_mean(z**2)

    grads = tape.gradient(loss, mlp.trainable_variables)

    for grad, var in zip(grads, mlp.trainable_variables):
        tf.debugging.check_numerics(grad, message=f"{var.name}: ")
        tf.debugging.assert_greater(tf.math.abs(grad), 0.0)

    assert len(grads) == len(mlp.trainable_variables)


def test_no_hidden_layers():
    import tensorflow as tf

    from modules.mlp import MLP

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    num_inputs = 10
    num_outputs = 1
    relu = tf.nn.relu
    sigmoid = tf.nn.sigmoid

    mlp = MLP(
        num_inputs, num_outputs, hidden_activation=relu, output_activation=sigmo
id
    )

    a = rng.normal(shape=[1, num_inputs])
    b = rng.normal(shape=[1, num_inputs])

    case1 = mlp(a + b)
    case2 = mlp(a) + mlp(b)

    tol = 2.22e-15 + 2.22e-15 * tf.abs(case2)

    tf.debugging.Assert(
        tf.reduce_any(tf.greater(tf.abs(case1 - case2), tol)),
        [case1, case2],
        summarize=2,
```

```
    )


if __name__ == "__main__":
    pytest.main([__file__])
```

```
import pytest


def test_causal_mask():
    import tensorflow as tf

    from modules.multi_head_attention import MultiHeadAttention

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

    context_length = 5
    num_heads = 1
    model_dim = 3
    batch_size = 1

    queries = rng.normal(shape=[batch_size, context_length, model_dim])
    keys = rng.normal(shape=[batch_size, context_length, model_dim])
    values = rng.normal(shape=[batch_size, context_length, model_dim])

    mask = tf.linalg.band_part(tf.ones((context_length, context_length)), 0, -1)
    # make the main diagonal 0
    mask = mask - tf.eye(context_length)

    # stack causal mask for each batch resulting in shape (B, seq_len, seq_len)
    mask = tf.stack([mask for _ in range(batch_size)])

    mha = MultiHeadAttention(num_heads, model_dim)

    with tf.GradientTape() as tape:
        tape.watch([queries, keys, values])
        output = mha(queries, keys, values, mask)

    dy_dx = tape.gradient(output, [queries, keys, values])

    # ensure that the derivative is zero for future tokens with respect to previ
ous tokens
    assert tf.reduce_all(dy_dx[0][:, 0, 1:] == 0)

    # ensure that the derivative is not zero for future tokens with respect to p
revious tokens
    assert tf.reduce_all([dy_dx[0][:, i] != 0 for i in range(1, context_length)]
)


@pytest.mark.parametrize(
    "batch_size, context_length, num_heads, model_dim",
    [
        (1, 5, 1, 3),
        (5, 10, 3, 9),
        (10, 20, 9, 27),
    ],
)
def test_dimensionality(batch_size, context_length, num_heads, model_dim):
    import tensorflow as tf

    from modules.multi_head_attention import MultiHeadAttention

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

    queries = rng.normal(shape=[batch_size, context_length, model_dim])
    keys = rng.normal(shape=[batch_size, context_length, model_dim])
    values = rng.normal(shape=[batch_size, context_length, model_dim])
```

```
    mask = tf.linalg.band_part(tf.ones((context_length, context_length)), 0, -1)
    # make the main diagonal 0
    mask = mask - tf.eye(context_length)

    # stack causal mask for each batch resulting in shape (B, seq_len, seq_len)
    mask = tf.stack([mask for _ in range(batch_size)])

    mha = MultiHeadAttention(num_heads, model_dim)

    output = mha(queries, keys, values, mask)

    assert output.shape == (batch_size, context_length, model_dim)


if __name__ == "__main__":
    pytest.main([__file__])
```

```python
import pytest


@pytest.mark.parametrize("input_depth", [3, 6])
@pytest.mark.parametrize("output_depth", [3, 6])
@pytest.mark.parametrize("kernel_size", [[2, 2], [4, 4], [8, 8]])
@pytest.mark.parametrize("group_norm_num_groups", [1, 3])
@pytest.mark.parametrize("resblock_size", [1, 2, 3])
def test_dimensionality(
    input_depth, output_depth, kernel_size, group_norm_num_groups, resblock_size
):
    import tensorflow as tf

    from modules.residual_block import ResidualBlock

    rng = tf.random.get_global_generator()
    rng.reset_from_seed(2384230948)

    residual_block = ResidualBlock(
        input_depth, output_depth, kernel_size, group_norm_num_groups, resblock_
size
    )

    input_tensor = rng.normal(shape=[1, 32, 32, input_depth])
    output_tensor = residual_block(input_tensor)

    tf.assert_equal(tf.shape(output_tensor)[-1], output_depth)
    tf.assert_equal(tf.shape(output_tensor)[0:3], tf.shape(input_tensor)[0:3])


if __name__ == "__main__":
    pytest.main([__file__])
```

```python
import pytest


@pytest.mark.parametrize(
    "batch_size, seq_len, model_dim, num_heads",
    [
        (1, 6, 12, 3),
        (5, 12, 36, 9),
        (10, 27, 108, 27),
    ],
)
def test_dimensionality(batch_size, seq_len, model_dim, num_heads):
    import tensorflow as tf

    from modules.transformer_decoder_block import TransformerDecoderBlock

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

    ffn_dim = 2048

    decoder_block = TransformerDecoderBlock(num_heads, model_dim, ffn_dim)

    input_embeddings = tf.Variable(rng.normal(shape=[batch_size, seq_len, model_
dim]))
    embeddings = decoder_block(input_embeddings)

    assert embeddings.shape == (batch_size, seq_len, model_dim)


if __name__ == "__main__":
    pytest.main([__file__])
```

```
import pytest


def test_autoregressive():
    import tensorflow as tf

    from helpers.adam import Adam
    from modules.transformer_decoder import TransformerDecoder

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

    batch_size = 100
    context_length = 6
    num_heads = 8
    model_dim = 512
    ffn_dim = 2048
    num_blocks = 6

    input_text = "<SOS> Florida man bites dog. <EOS> Dog bites professor curro. <EOS>"

    transformer_decoder = TransformerDecoder(
        context_length,
        num_heads,
        model_dim,
        ffn_dim,
        num_blocks,
        input_text,
    )
    learning_rate = 0.0001
    adam = Adam(learning_rate)

    text, targets = transformer_decoder.get_tokens_and_targets()

    for _ in range(20):
        batch_indices = rng.uniform(
            shape=[batch_size], maxval=text.shape[0], dtype=tf.int32
        )

        with tf.GradientTape() as tape:
            input_tokens_batch = tf.gather(text, batch_indices)
            targets_batch = tf.gather(targets, batch_indices)

            labels = targets_batch
            logits = transformer_decoder(input_tokens_batch)
            current_train_loss = tf.reduce_mean(
                tf.nn.sparse_softmax_cross_entropy_with_logits(
                    labels=labels,
                    logits=logits,
                )
            )

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

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

    accuracy = tf.reduce_mean(
        tf.cast(
            tf.equal(
                logits.numpy().argmax(axis=2).reshape(-1),
                labels.numpy().reshape(-1),
            ),
```

```
            tf.float32,
        )
    )

    assert accuracy == 1.0
    assert transformer_decoder.predict("<SOS> Florida") == " man bites dog."
    assert transformer_decoder.predict("Dog") == " bites professor curro."


def test_exploding_gradients():
    import tensorflow as tf

    from helpers.adam import Adam
    from modules.transformer_decoder import TransformerDecoder

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

    context_length = 6
    num_heads = 8
    model_dim = 512
    ffn_dim = 2048
    num_blocks = 6

    input_text = "<SOS> Florida man bites dog. <EOS> Dog bites professor curro. <EOS>"

    transformer_decoder = TransformerDecoder(
        context_length,
        num_heads,
        model_dim,
        ffn_dim,
        num_blocks,
        input_text,
    )
    learning_rate = 0.0001
    adam = Adam(learning_rate)

    text, targets = transformer_decoder.get_tokens_and_targets()

    with tf.GradientTape() as tape:
        labels = targets
        logits = transformer_decoder(text)
        loss = tf.reduce_mean(
            tf.nn.sparse_softmax_cross_entropy_with_logits(
                labels=labels,
                logits=logits,
            )
        )

    grads = tape.gradient(loss, transformer_decoder.trainable_variables)

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

    for grad, var in zip(grads, transformer_decoder.trainable_variables):
        tf.debugging.check_numerics(grad, message=f"{var.name}: ")

    assert len(grads) == len(transformer_decoder.trainable_variables)


if __name__ == "__main__":
    pytest.main([__file__])
```

```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()
```