Deep Learning Midterm: ConvMixer

Daniel Mezhiborsky and Theo Jaquenoud
Cooper Union - ECE-472 Deep Learning - Prof. Curro

Introduction

ConvMixer is a novel convolutional neural network architecture designed in part to replicate certain mechanisms of vision transformers (ViT). In particular, the authors speculated that the act of splitting images into patches contributed to the ViT's success, not just the "inherently-more-powerful" transformer architecture. In the paper "Patches Are All You Need" (a jab at the paper that introduced transformers, called "Attention is all you need"), the authors describe the ConvMixer architecture, which operates solely on equally-sized patches of an image, but uses only standard convolutional layers to achieve both the patching and mixing. They show that, for an equal number of parameters and similar dataset sizes, the ConvMixer outperforms ViTs as well as other common models like ResNet.

Our goal for this project is to implement the ConvMixer as it is presented in the paper, and replicate some of their results for a range of model widths and depths using the publicly available CIFAR-10 dataset. While the code for the implementation is available on github, we wanted to implement it in tensorflow, our library of choice for this course. In addition to implementing the architecture in tensorflow, we were interested in replicating their results, notably their stated test accuracy on CIFAR-10 for various variants of ConvMixer. To get as close as possible to their experimental setups, we also had to recreate their experimental setups. Most notably, we had to replicate their data augmentation methods and their learning rate scheduling.

Background

ViTs (Dosovitskiy et al., 2020) were designed to use the powerful transformer architecture (Vaswani et al., 2017), which proved successful in natural language processing (NLP) tasks, for computer vision (CV) tasks. In order to feed large images into transformers, ViTs first split images into patches, and then flatten these patches and learn lower dimensional embeddings, as well as positional embeddings (using self-attention). Those embeddings are then fed into a standard transformer. As the creators of ConvMixer

(under double-blind review) point out, this means ViTs introduce two new mechanisms to neural networks designed for image processing, patch-wise processing and transformers.

The ConvMixer architecture is motivated by the idea that the patch-wise processing of images is a strong contributor to the success of ViTs, and could lead to good models without being bogged down by the high number of parameters and training time of transformers. In particular, the authors point out the quadratic runtime of self-attention layers as a major shortcoming. Their proposed model works by first creating patches using a convolution with a kernel and stride who's sizes match the patch size. After an activation and batch normalization, these patch embeddings are passed into a series of ConvMixer layers. These layers consist of a depthwise convolution using a large kernel to mix spatial locations with a large receptive field, followed by a pointwise convolution to mix channel locations. Finally, global average pooling and a fully connected layer with softmax activation are used to output the classification. Their schematic for the architecture is shown in Figure 1.

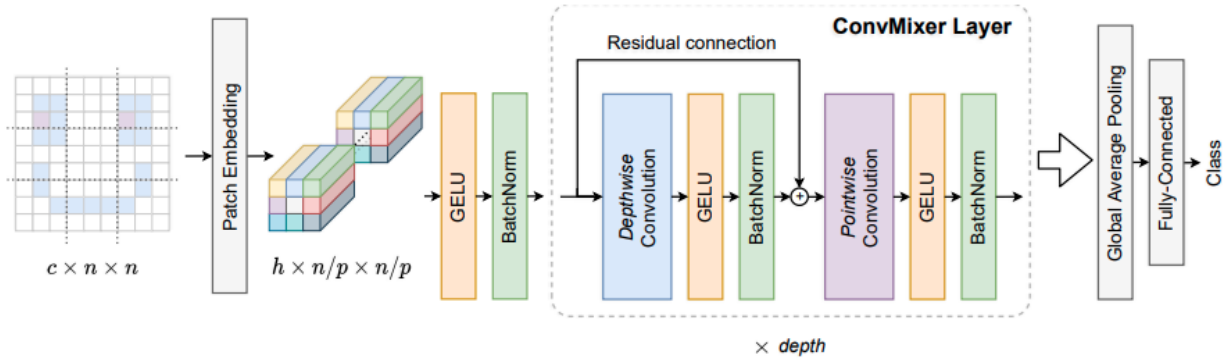

Figure 1: Diagram of ConvMixer architecture. Input image has c channels and n by n pixels. An h-dimensional embedding is obtained using a convolution with h output channels. The patch size p is obtained using a kernel and stride of the same size. Within the ConvMixer layer, the *Depthwise Convolution* is a grouped convolution with a "large" kernel (8x8 for CIFAR-10), while the *Pointwise Convolution* is a simple convolution with kernel size of 1.

Implementation

Implementing the model itself in TensorFlow was not too challenging, since a goal of ConvMixer is to use standard convolutional layers. We did use TensorFlow Addons for the AdaptiveAveragePooling2D layer. However, we faced challenges replicating the training setup that the authors had. One major point was the learning rate scheduler. We originally used a slanted triangle scheduler because the authors mentioned using a

“simple triangle” schedule, but then we noticed that their GitHub had a different scheduler - OneCycleLR. We asked the authors on GitHub about this, and they replied with some info about the right scheduler to use.

We implemented the OneCycleLR scheduler in Tensorflow based on the authors’ comments and the ConvMixer repo. While this is not an exact replica - we couldn’t take advantage of some of the many features of the PyTorch/timm Scheduler class - it does work.

Results

Having implemented the basic architecture, we set about testing it in order to replicate their results for CIFAR-10. The body of the paper doesn’t describe their experimental set-up in any detail, but an appendix goes in more depth. As it turned out, we also had to reach out to the authors of the paper to get some additional details.

First Phase: Testing the architecture

The very first runs used the Adam optimizer (Kingma et al., 2015) with a fixed learning rate, and no data augmentation. While the upwards trend in accuracy suggested that our implementation was functioning as intended, we were achieving final test accuracies quite below their claimed results, at about 88.5% compared to their 91.26% for a model with width 128 and depth 4 (hereby referred to as 128/4).

Second Phase: Adding learning rate schedulers and data augmentation

The authors state that they used “a simple triangular learning rate schedule,” but do not specify further the parameters of this scheduler, notably the maximum learning rate, the number of steps or epochs before reaching that maximum, or the slope in either stage of the schedule. We reached out to the authors through their github for some clarification, but started experiments with other triangular learning rates as well as data augmentation while we awaited their response.

The scheduler we used was a slanted triangle with a maximum learning rate of 0.001 (Howard et al., 2018, implementation from linuxtut.com/en/bd6207e562c1b1270d33/); it is displayed below in Figure 2. For data augmentation, we used keras’ ImageDataGenerator which generates batches of image data with real time random data augmentation. We configured it with a zoom, width, and height shift range of 10%, rotation range of 10%, and horizontal flipping. We also added 20% dropout to the last layer and a l2 kernel regulzarizer with a lambda of 10^{-3}. We trained 6 of these models for every combination of widths from [128, 256] and depths from [4, 8, 12]. The resulting test

accuracies after 50 epochs are shown in Figure 3a and Figure 3b as a function of number of parameters and relative training time respectively. Relative training time was calculated per epoch on two different GPUs and proved to be very similar across both sets of hardware, with a standard deviation of <5%.

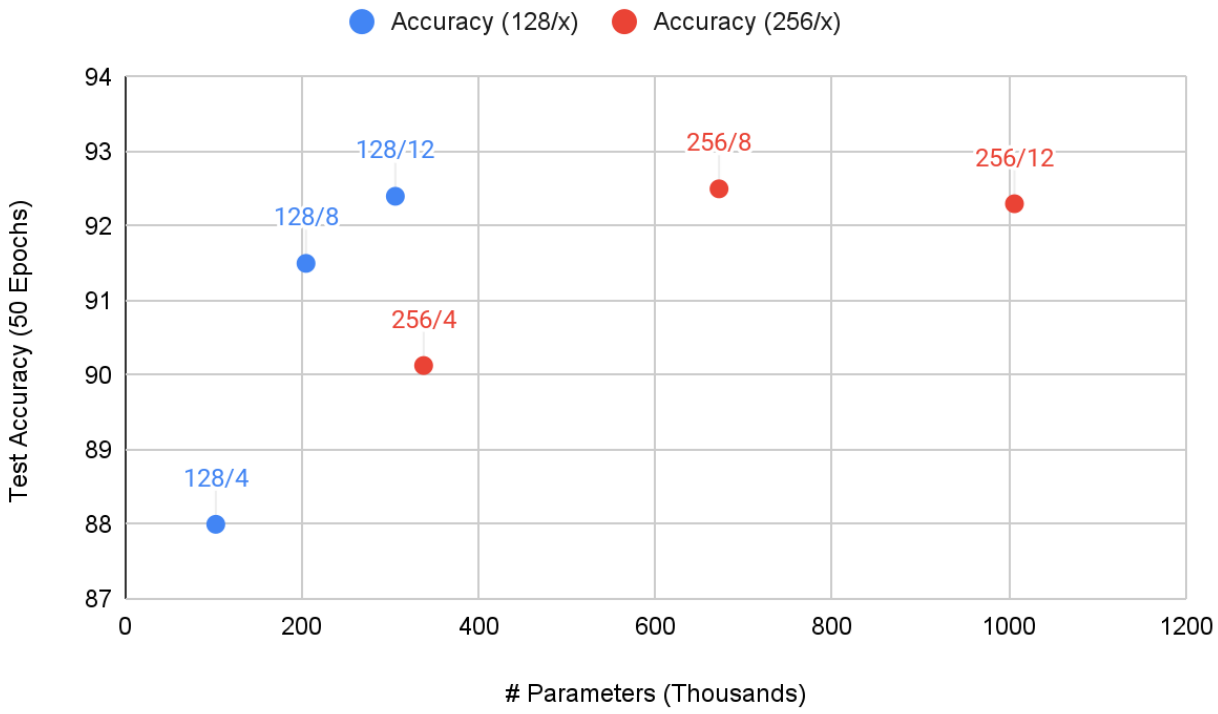

Figure 3a: Test accuracy as a function of number of parameters.

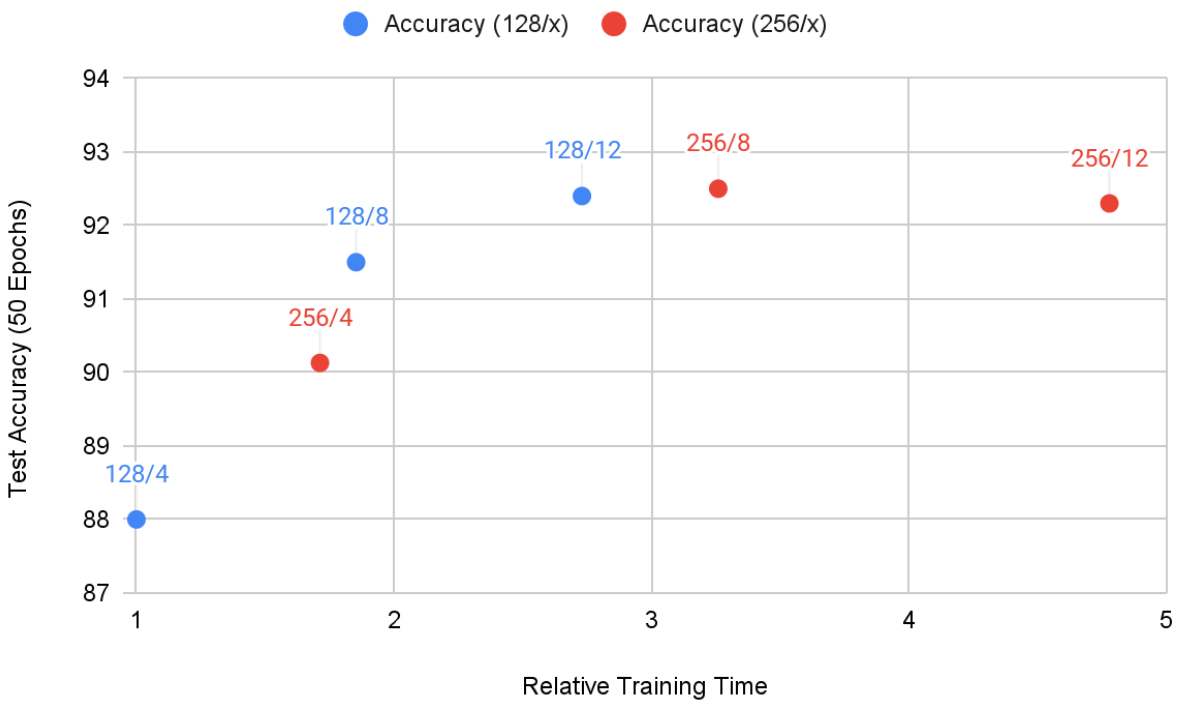

Figure 3b: Test accuracy as a function of relative training time (calculated by dividing the average training time per epoch per model on 2 different GPUs by that of 128/4).

Third Phase: Experimenting with a number of hyperparameters

Having trained models to within a couple percentage points of the accuracy stated by the paper, we noticed that the learning rate scheduler wasn't fully converging to an optimum. Furthermore, we noticed a slight tendency to overfit despite the regularization. In order to improve our models, we took a closer look at some of the details in their implementation, with a focus on learning rate scheduling, regularization, and data augmentation.

The first improvement we made to better mimic their experimental set up was to implement the same learning rate scheduler. We first tried a different one-cycle learning rate based on "Super-Convergence: Very Fast Training of Neural Networks Using Large Learning Rates" (Smith and Topin, 2018), but we still didn't know what parameters were used for ConvMixer. Once the authors replied to us on github forums with the learning rate scheduler they used, along with its parameters like the maximum learning rate, we abandoned the previous one-cycle learning rate, instead recreating their schedule in tensorflow and using it for our remaining experiments. A plot of the one-cycle learning

rate and an example of using it for training a 128/4 model are displayed in Figure 4a and 4b.

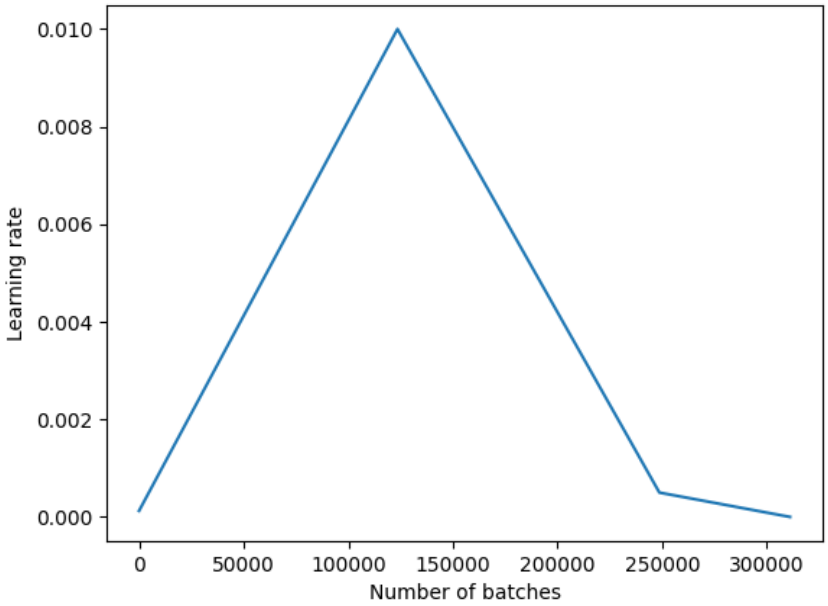

Figure 4a: Plot of the per-batch learning rate using the One-Cycle Learning Rate Scheduler.

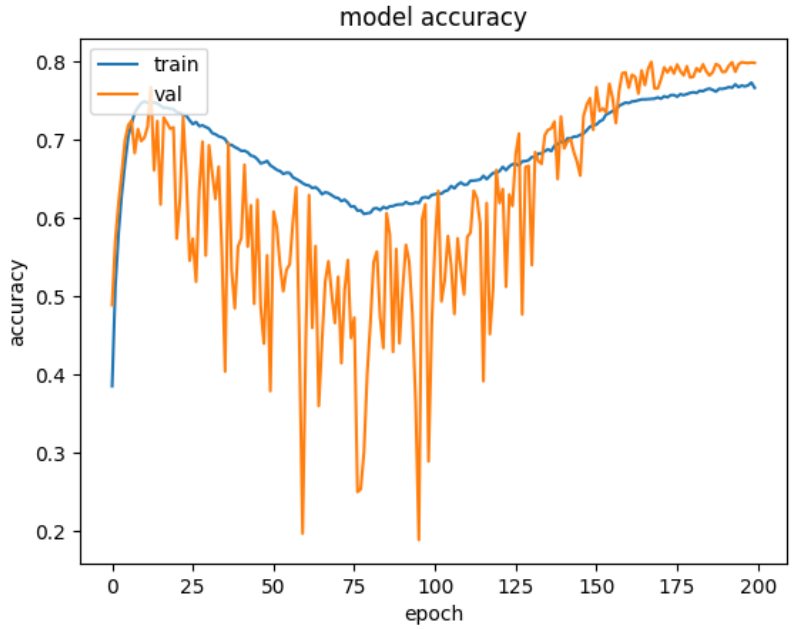

Figure 4b: Training and validation curves for 200 epochs of a one-cycle triangle learning rate without weight decay or norm clipping, and with a simple ImageDataGenerator for augmentation. The wild validation accuracy was an issue with OneCycleLR until we improved our data augmentation.

The second improvement was to mimic their regularization. In particular, they stated that they did not use kernel regularizers, or dropout, but instead used AdamW (Loshchilov and Hutter, 2019) (mostly with 0 weight decay) and gradient norm clipping as regularizers. They claimed that their data augmentation was significant enough that little regularization was needed.

The third and final improvements we made were with regards to data augmentation. First we just increased the intensity of the data augmentation, increasing zoom, width, and height shift range to 20%. We also looked into channel shift range which is supposed to apply a different random shift to each color channel in an image, but due to an implementation bug in tensorflow, it applies the same shift to all channels, which is a significantly less robust regulzarizer. We saw marginal improvements with this augmentation scheme, but it still lagged behind their reported results. Finally, we found an implementation of RandAug (Cubuk et al., 2019, implementation from imgaug.readthedocs.io) which matched the one described by the creators of ConvMixer. This method uses a set of 14 possible transformations and two hyperparameters: *n,* the number of transformations sampled uniformly from the set, and *m*, the "scale" of each transformation. We chose 3 and 7 respectively for these hyperparameters, as the values are not reported in "Patches Are All You Need"

Fourth Phase: Final Testing

Having achieved as close of an experimental setup as we could to the one described by the creators of ConvMixer, we set out one more time to match their results by training the 128/x models for 200 epochs. The results are presented below:

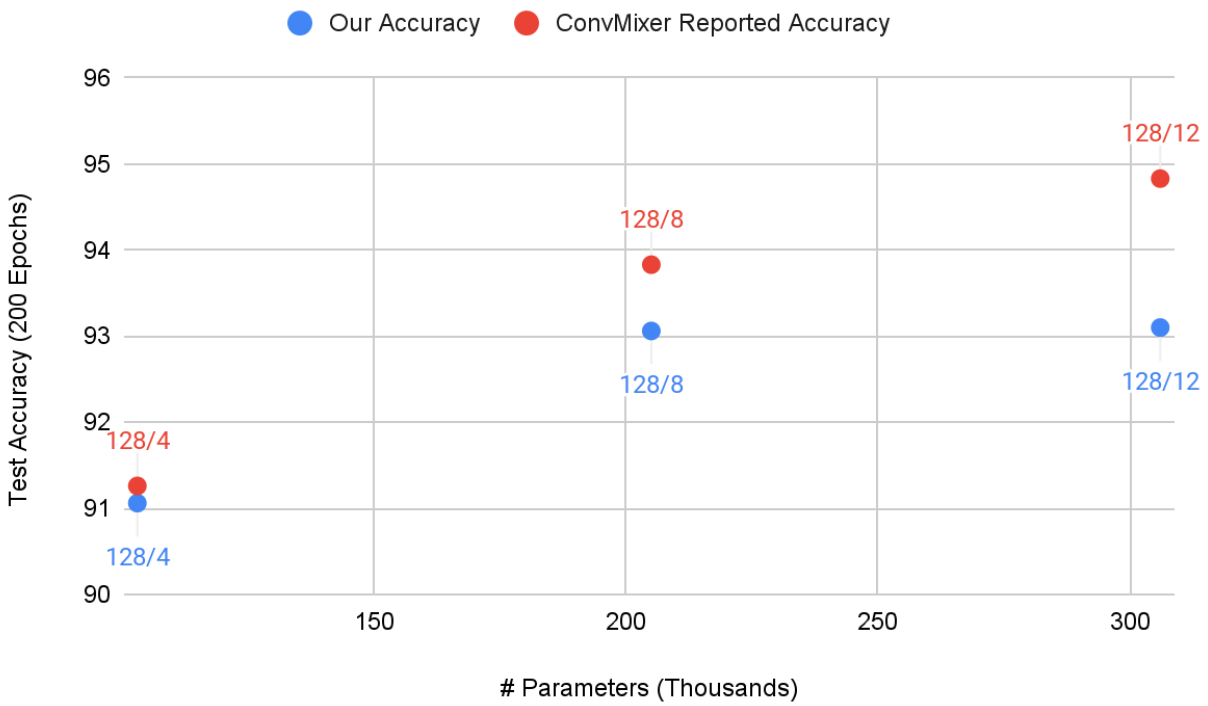

Figure 5: Test accuracy compared between our implementation and training compared to the stated accuracies in the ConvMixer paper. While we have narrowed the gap significantly compared to our first tests, we remain slightly behind. We suspect this discrepancy is due to the additional data augmentation used by the authors.

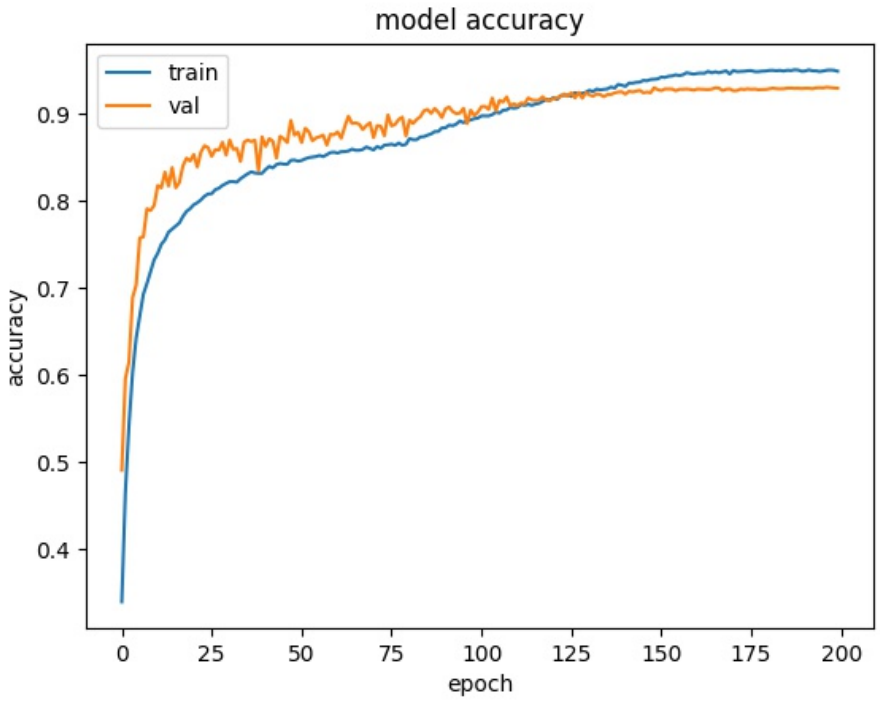

Figure 6: Model accuracies for our final 128/8 run. We suspect additional regularization is needed.

One notable observation while training our models was the different gaps between the training and validation accuracy. In our 128/4 model, the training accuracy was consistently lower than the validation accuracy, with the gap shrinking during the second phase of the learning rate cycle, ultimately ending at a 3% gap at the 200th epoch. This seems to suggest, as the authors stated, that strong data augmentation is a strong regularizer for ConvMixer. We also notice that our gap to the reported accuracies gets wider as the complexity of the model increases. If the discrepancy is indeed due to additional data augmentation (bigger transformation set for RandAug, mixup, random erasing, etc...), it is interesting to note that larger models require more regularization or data augmentation.

Conclusion and Future Work

Overall, we believe we were successful in reimplementing the ConvMixer model for tensorflow and keras. The number of trainable parameters being exactly the same between the paper's and our implementation, and us getting very similar training accuracies, are strong indicators that the implementations are equivalent. That being said, Pytorch and timm provide very robust frameworks for implementing various data

augmentation, not all of which we were able to replicate. We believe remaining discrepancies between models are mainly due to this fact.

Appendix A: Tensorflow Code

ConvMixer.py

```
import tensorflow as tf
import tensorflow_addons as tfa
from tensorflow import keras

class Residual(keras.layers.Layer):
  def __init__(self, fn):
    super().__init__()
    self.fn = fn

  def call(self, x):
    return self.fn(x) + x


def GELU():
  return keras.layers.Activation(tf.keras.activations.gelu)


def ConvMixer(dim, depth, kernel_size=9, patch_size=7, n_classes=10):
  return keras.Sequential([
    keras.layers.Conv2D(dim, kernel_size=(patch_size, patch_size), strides=(patch_size, patch_size),
                        input_shape=(32, 32, 3)),
    GELU(),
    keras.layers.BatchNormalization(),

    *[keras.Sequential([
      Residual(keras.Sequential([
        keras.layers.Conv2D(dim, kernel_size=(kernel_size, kernel_size), groups=dim, padding="same"),
        GELU(),
        keras.layers.BatchNormalization(),
      ])),
      keras.layers.Conv2D(dim, kernel_size=(1, 1)),
      GELU(),
      keras.layers.BatchNormalization()
    ]) for i in range(depth)],

    tfa.layers.AdaptiveAveragePooling2D((1, 1)),
    keras.layers.Flatten(),
    keras.layers.Activation(tf.keras.activations.linear),
    keras.layers.Dense(n_classes, activation='softmax')
```

```
])
```

run_convmixer.py

```
#!/usr/bin/python3

import scipy  # needed for data aug
import numpy as np
import imgaug
import tensorflow as tf
import tensorflow_addons as tfa
from tensorflow import keras

from load_cifar_10 import load_cifar_10_data

from ConvMixer import ConvMixer
from OneCycleLR import OneCycleLRScheduler

from imgaug import augmenters as iaa

AUTO = tf.data.AUTOTUNE
rand_aug = iaa.RandAugment(n=3, m=7)

CIFAR_10_DIR = 'cifar-10-python/cifar-10-batches-py'
BATCH_SIZE = 32
NUM_EPOCHS = 200

MAX_LR = 0.01

INITIAL_EPOCH = 0
LOAD_SAVED_MODEL = False
SAVED_MODEL_PATH = 'aug_onecyclecmx_0.01_nodropout_noreg_200/cifar_10_model_128x12_epoch_125-0.92/'

SAVE_PERIOD = 5

EXPERIMENT_NAME = f'aug_onecyclecmx_lr-{MAX_LR}_bs-{BATCH_SIZE}' \
                  f'_nodropout_noreg_{NUM_EPOCHS}_saved-{LOAD_SAVED_MODEL}'

def augment(images):
  # Input to `augment()` is a TensorFlow tensor which
```

```
    # is not supported by `imgaug`. This is why we first
    # convert it to its `numpy` variant.
    images = tf.cast(images, tf.uint8)
    return rand_aug(images=images.numpy())


def main():
    sched = lambda t, lr_max: np.interp()

    images, train_filenames, labels, t_images, test_filenames, t_labels, label_names = \
        load_cifar_10_data(CIFAR_10_DIR)

    images_conv = (
        tf.data.Dataset.from_tensor_slices((images, labels))
            .shuffle(BATCH_SIZE * 100)
            .batch(BATCH_SIZE)
            .map(lambda x, y: (tf.py_function(augment, [x], [tf.float32])[0], y), num_parallel_calls=AUTO)
            .prefetch(AUTO)
    )

    t_images_conv = (
        tf.data.Dataset.from_tensor_slices((t_images, t_labels))
            .batch(BATCH_SIZE)
            .prefetch(AUTO)
    )

    if LOAD_SAVED_MODEL:
        model = keras.models.load_model(SAVED_MODEL_PATH)
    else:
        model = ConvMixer(128, 12, kernel_size=8, patch_size=1, n_classes=10)
        model.compile(optimizer=tfa.optimizers.AdamW(weight_decay=0),
                      loss="sparse_categorical_crossentropy", metrics=['accuracy'])
        model.build()
        model.summary()

    filename_prefix = "./{}/cifar_10_model_{}x{}".format(EXPERIMENT_NAME, 128, 12)
    filename = filename_prefix + "_epoch_{epoch:02d}-{val_accuracy:.2f}"
    checkpt = tf.keras.callbacks.ModelCheckpoint(filename, period=SAVE_PERIOD)

    lrsched = OneCycleLRScheduler(NUM_EPOCHS, MAX_LR, images.shape[0]/BATCH_SIZE)

    history = model.fit(
```

```
                images_conv,
                initial_epoch=INITIAL_EPOCH,
                epochs=NUM_EPOCHS,
                batch_size=BATCH_SIZE,
                validation_data=t_images_conv,
                callbacks=[checkpt, lrsched]
    )

    np.save(f'./{EXPERIMENT_NAME}/history.npy', history.history)

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

OneCycleLR.py

```
import numpy as np
import tensorflow as tf
from tensorflow import keras

class OneCycleLRScheduler(keras.callbacks.Callback):
    def __init__(self, epoch_count, lr_max, batches_per_epoch):
        super().__init__()
        self.epoch_count = epoch_count
        self.epoch = 1
        self.lr_max = lr_max
        self.batches_per_epoch = batches_per_epoch

    def on_batch_begin(self, batch: int, logs=None):
        self.batch = batch
        self.t = self.epoch + (self.batch + 1) / self.batches_per_epoch
        sched = np.interp([self.t], [0, self.epoch_count * 2 // 5, self.epoch_count * 4 // 5, self.epoch_count],
                          [0, self.lr_max, self.lr_max / 20.0, 0])[0]
        tf.keras.backend.set_value(self.model.optimizer.lr, sched)

    def on_epoch_begin(self, epoch: int, logs=None):
        epoch = epoch + 1  # tensorflow is off-by-one :P
        self.epoch = epoch
        print(f"lr at epoch {epoch}: {tf.keras.backend.get_value(self.model.optimizer.lr)}")
```