MLP-Mixer report:

Yael Babichenko 036737062
Shahar Rozenberg 214544991

Introduction:

Objectives:

The paper introduces a method called ‘MLP-Mixer’, which is a neural network architecture for image classification tasks. Its significance is that it relies only on MLPs. The goal of this paper is to demonstrate that convolutional layers and even self-attention mechanisms, which are very dominant in the computer vision field (and incredibly dominant in the image classification task), are not strictly necessary for achieving competitive results in those tasks.

Methods:

Architecture

The MLP-Mixer processes input images by dividing them into non-overlapping patches, projecting them linearly, and representing them as a "patches × channels" table.

- It alternates between two types of MLP layers (MLP-blocks):
  - **Token-Mixing MLPs**: These mix spatial information by processing each feature independently across all patches.
  - **Channel-Mixing MLPs**: These mix feature information by processing each patch independently across all features.
- The architecture utilizes the MLP-blocks in a Mixer-block, which is being used in the MlpMixer itself.
- The model uses several improving components such as skip connections, layer normalization, and GELU activations.

Training

- Pre-training on large datasets such as ImageNet-21k and JFT-300M.
- Fine-tuning on smaller datasets for transfer learning (CIFAR-10 is between those).
- Key hyperparameters include data augmentation techniques like RandAugment and mixup, along with dropout and stochastic depth regularization.
- It also used weight decay, linear learning warmup and gradient clipping.

Implementation:

The Model:

1. **MLP-Block:**

   This component represents a single MLP block, consisting of:

   - **Fully Connected Layer**: Applies linear transformations to the input data.
   - **GELU Activation Function**: Non-linear function to enhance the model.
   - **Second Fully Connected Layer**: Further processes the output from GELU
   - **Dropout:** regularization to decrease overfitting

   The block ensures the output dimensions match the input dimensions. The hidden layer size is selected based on the article's Table 1 results for the S/32 configuration.

2. **Mixer-Block:**

   This component implements the core Mixer layer of the architecture, consisting of:

   - **Layer Normalization**: Stabilizes gradients by normalizing the input.
   - **Token-Mixing MLP**: Captures dependencies on tokens, using spatial mixing.
   - **Channel-Mixing MLP**: Focuses on relationships across feature channels.
   - **Skip Connections**: Adds the input back to the processed output, aiding in information preservation and gradient flow.

   This modular design effectively separates spatial and channel-wise processing, encapsulating the architecture's core principles.

3. **MlpMixer:**

   The full MLP-Mixer model comprises:

   - **Initial Convolutional Layer**: Splits the input data into patches and embeds them into a fixed-dimensional space (patches × channels).
   - **Mixer-Block Stack**: A loop iterates through a predefined number of Block layers also taken from table 1.
   - **Output Layer**: Following normalization and pooling, the final fully connected layer to the output classes.

Data Preparation:

We followed some of the data preparation steps outlined in the reference paper for the CIFAR-10 dataset (in the fine-tuning part):

- Resized the images to 256 pixels. ("resmall-crop")
- Applied a central crop to adjust the resolution to 224 × 224 pixels.
- Normalized the data for compatibility with the dataset's means and stds (taken from a viable source).

Baseline Results:

The paper's evaluation process:

In the paper they used various evaluation procedures:

Image size	Pre-Train Epochs	ImNet top-1	ReaL top-1	Avg. 5 top-1	Throughput (img/sec/core)	TPUv3 core-days

- The last 2 measures the cost of the model, but they used TPU to run it. In our case, we have a very-limited Google Colab GPU, so we didn't measure cost that way.
- ImNet and ReaL are specific tasks (datasets) that the model is fine-tuned to. The paper evaluates the performance using top-1 measurement (is the first prediction accurate?). We also do not consider those, as we only work with CIFAR-10.
- **Avg 5 top-1** is the calculated average top-1 measure for 5 different fine-tuned datasets (ImageNet, CIFAR-10, CIFAR-100, Pets, Flowers).

The evaluation in our capabilities:

Our main limitations are:

- We only use one dataset (CIFAR-10), and it is relatively small
- We cannot pre-train and then fine-tune.
- We have limited PU.

Therefore, we evaluate the strength of our model only using **top-1** over the CIFAR-10 test, using the parameters' structure of one of the small options (S or B) . We will compare it to the avg 5 top-1.

As for the cost, we considered the average time per epoch (which derives from iterations per second).

The result differences:

We are told that pre-training on ImageNet-21k and on JFT-300M and using L/16 and H/14 respectively, yields 93.91% and 95.71%, relatively.

Due to the bigger datasets, stronger computation units and longer time they had in the paper, here are the key differences between our and their results:

- We DIDN'T use most of the data augmentation methods, due to the fact that in order to check if they work, we only ran them on 5~ epochs (didn't want to lose the GPU). This resulted in mostly non-beneficial "improvements", so we did not consider most of those techniques.
- No pre-train to fine-tuning -> as we only trained over CIFAR-10
- Smaller structures (No L or H, only S or B) and only 20 epochs.
- Lack of time resulted in us choosing a patch size of 32 over 16. Although it is probably less good, its relatively low cost helped us to make this kind of decision.

Our best result was an accuracy of **69.24%**. This outcome is not competitive with the pre-trained over a lot of epochs with overfitting extraction tools, but it is not too bad either.

Discussion:

The biggest challenge we faced during implementation was the problem of overfitting. Throughout the project, we attempted to address this issue by adjusting various parameters, including: Patch size, Batch size, Learning rate, Image resolution, and in general we played with all the parameters and hyperparameters, as well as the data augmentation techniques.

Data Augmentation techniques:

- We left RandAugmented, stochastic depth, linear learning, weight decay and mixup because it took a lot of time and didn't really help.
    - Those are good for bigger datasets. In our case, it made our learning very weak.
- On the other hand, the dropout and setting the learning rate to 0.006 and even using gradient clipping in the train did help

Choosing the model structure:

- Checked on 5 epochs (due to limited time resources)
- Eventually we chose: **B/32**

```
S/32:
avg runtime per epoch - 02:30
Top-1 Test Accuracy: 63.16%
S/16
avg runtime per epoch - 05:15
Top-1 Test Accuracy: 64.47%
B/32
avg runtime per epoch - 09:30
Top-1 Test Accuracy: 65.27%
B/16
avg runtime per epoch - 02:40
Top-1 Test Accuracy: 64.50%
```

- It got us the best results AND we got a relatively low runtime.
- Our dataset is obviously small, but using B configurations improved the results without adding too much to the cost.
- As we understand that a patch size of 16 is more informative, getting kicked out of Colab's GPU helped us decide that a size of 32 is good enough for us :)

Appendix:

Comparing the results:

In the paper they use all of those techniques and have bigger PUs, and here are our results compared to them:

Model Name	Image Size	Pre-Trained Epochs	top-1
Our Model			
Mixer-B/32	224	20	**69.24**
Pre-Trained on ImageNet			
Mixer-B/16	224	300	88.33
Mixer L/16	224	300	87.25
Pre-Trained on ImageNet-21k			
Mixer-B/16	224	300	92.5
Mixer L/16	224	300	93.63
Pre-Trained on JFT-300M			
Mixer-S/32	224	5	87.13
Mixer-B/32	224	7	90.99
Mixer-S/16	224	5	89.5
Mixer-B/16	224	7	92.6
Mixer L/32	224	7	93.24
Mixer L/16	224	7	94.51