Due Date : March 2nd, 24:00
Student : Ziqiang Wang

1. In figure 1.

```python
def cross_entropy_loss(logits: torch.Tensor, labels: torch.Tensor):
    """ Return the mean loss for this batch
    :param logits: [batch_size, num_class]
    :param labels: [batch_size]
    :return loss
    """

    log_probs = torch.nn.functional.log_softmax(logits, dim=1)
    log_probs_of_labels = log_probs[range(logits.shape[0]), labels]
    loss = -torch.mean(log_probs_of_labels)
    return loss
```

FIGURE 1 – Question1 : completed cross-entropy loss function in utils.py.

2. As shown in figue 2, ReLU activation function is the best choice.
   ReLU activation has the property of producing sparse activations, which can help with reducing overfitting and improving the generalization ability of the model. Also, ReLU is a non-saturating activation function, which allows the network to avoid the vanishing gradient problem that can occur with saturating activation functions like sigmoid and tanh.
3. As shown in figure 3, the learning rate of 0.001 has the best performance in terms of loss and accuracy of training and validation.
   The learning rate of 0.1 has the worst performance with the lowest accuracy on all sets. This is because a large learning rate causes the model to update the weights too aggressively, leading to unstable convergence and poor generalization.Besides, the learning rate of 0.00001 also has poor performance due to being too small, causing the model to take a long time to converge and potentially getting stuck in local minima.
   Therefore, a moderate learning rate is often a good starting point.
4. As shown in figure 4, the patch sizes of 2, 4, and 8 have similar performance in terms of loss and accuracy on valid dataset. The patch size of 16 gets the worst results. In MLPMixer, the patch embedding operation help to capture local information in images like convolution operation. However, the patch size of 16 is too large for the image size of 32, so it is hard to capture local information.

TABLE 1 – The effect on the number of model parameters and running time. Running time (inference) means the time for the model to feed forward 5000 valid images. Running time (training) represents the time to train the model on 45000 images for one epoch.

patch size	parameters	running time (inference)	running time (training)
2	2.38M	34.0s	1.5s
4	2.19M	10.3s	0.5s
8	2.18M	4.6s	0.3s
16	2.31M	2.9s	0.2s

As show in table 1, the model with larger patch size has faster running speed. For model parameters, the models with patch sizes of 4 and 8 have similar numbers of parameters and

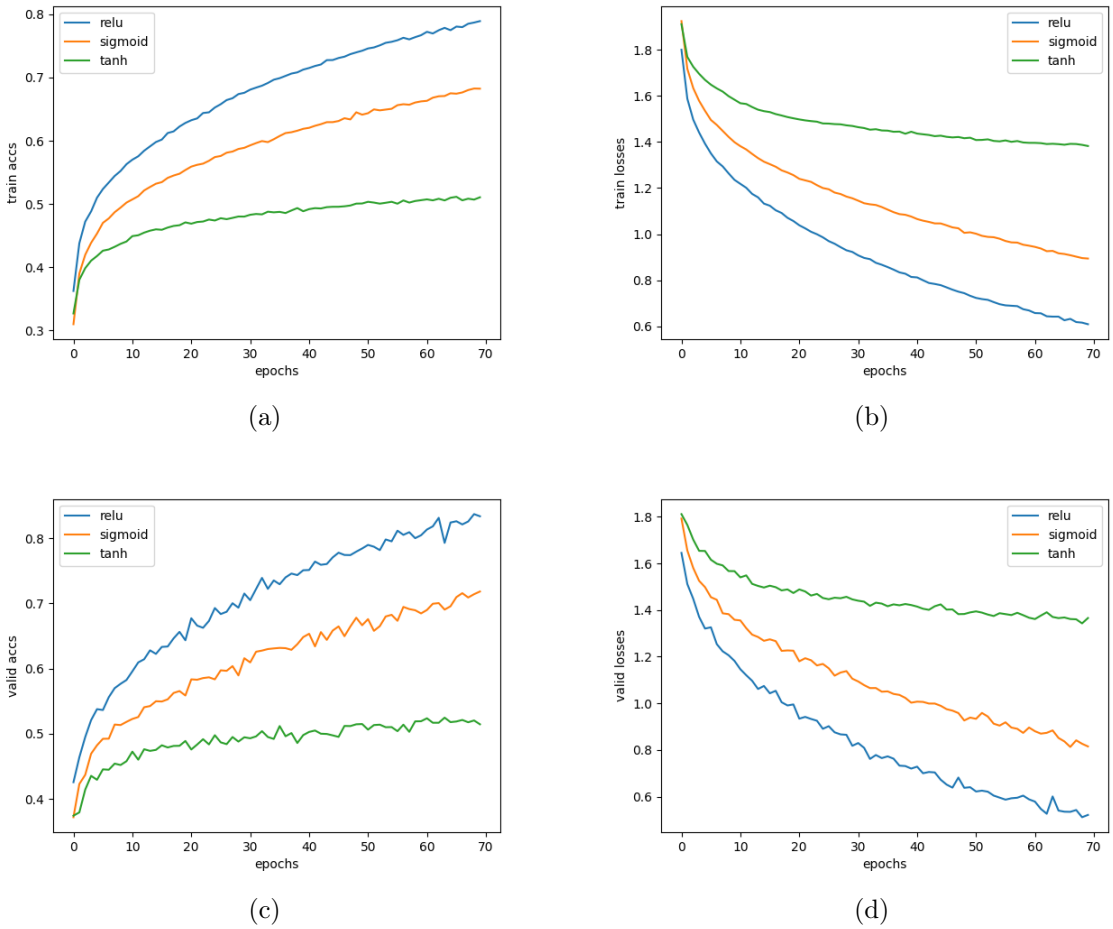


FIGURE 2 – Visualization of the effect of different non-linearity activation functions for the MLP.

less than those with patch sizes of 2 and 16. The models with patch sizes of 2 and 16 also have similar numbers of parameters. I guess that, given a fixed input size, the more close the patch size and grid size are, the less number of model parameters is.

5. Hyperparameters for my best ResNet18 model : batch size=128, learning rate=1e-3, optimizer="Adam", momentum=0.99, weight decay=5e-3, epoch=30.

   Results for my best model : test accuracy = 90.2 %. Loss and accuracy on training and valid sets are presented in figure 5.

   Visualization procedure :
   — Get the weight tensor of the first convolutional layer and standardize the weights.
   — Convert each kernel to a gray scale image by averaging the RGB channels (in channel).
   — Plot and save the images.
   The visualization result is shown in figure 6.

6. Hyperparameters for my best MLPMixer model : embed dim=512, num blocks=8, drop rate=0.4, batch size=128, learning rate=1e-3, optimizer="Adam", momentum=0.99, weight decay=5e-4, epoch=30.

   Results for my best model : test accuracy = 77.9 %. Loss and accuracy on training and valid sets are presented in figure 7.

   The visualization is shown in figure 8. It consists of 256 units that correspond to hidden

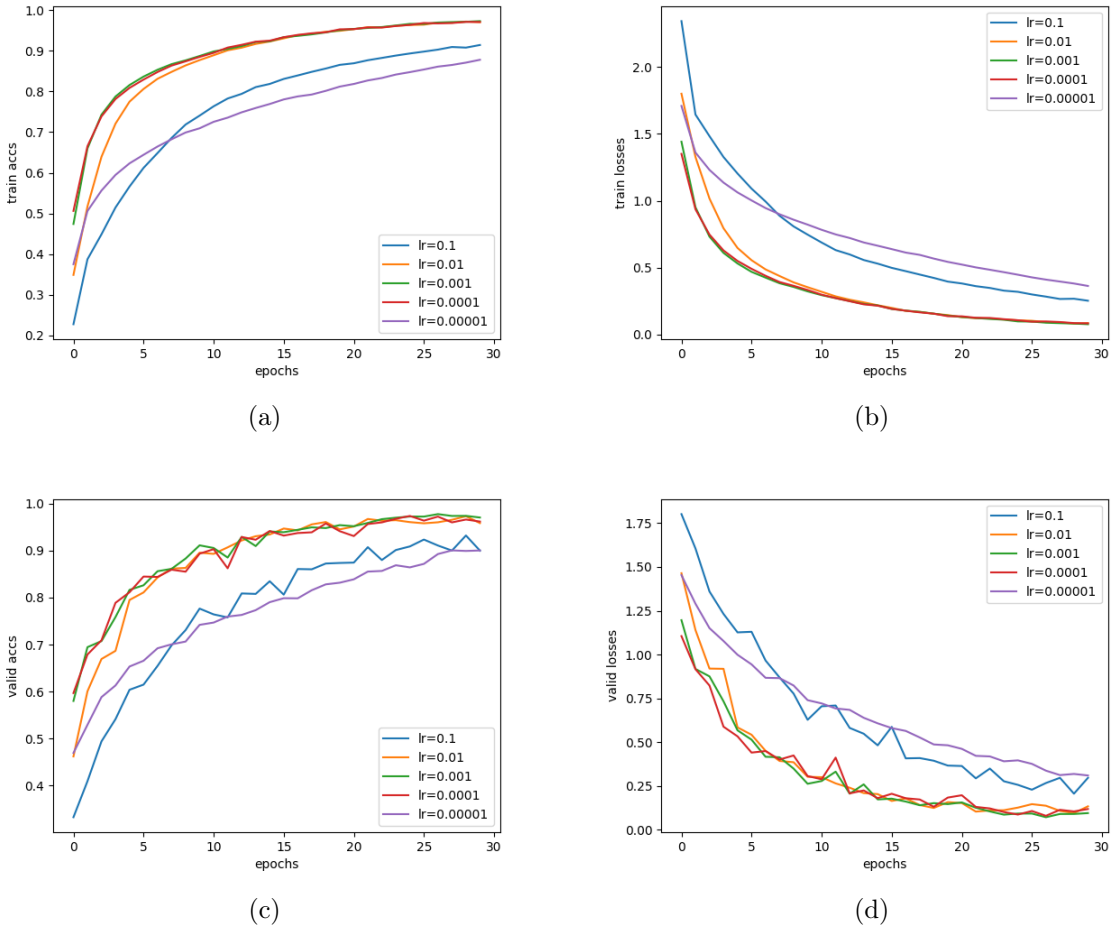


FIGURE 3 – Visualization of the effect of different non-linearity activation functions for the MLP.

size of 256 (embed dim=512). Each unit has 64 weights, corresponding to 8×8 patches respectively. Specifically, each weight in one units is multiplied with the information from a patch. Compared with the ResNet visualization in figure 6, the similarity is that the units from the two figures are both corresponding to channels. The difference is that, a single unit in the figure 6 weights a local window pixel by pixel, while one single unit in the figure 8 weights the whole image information patch by patch.

Therefore, the MLPMixer has ability to capture local information although it only uses MLPs which just capture global relationship of inputs. On the other hand, as mentioned in Problem 3, the patch embedding part has the same operation as convolution, thus the MLPMixer has more inductive bias than MLPs as well as local context representation. I think the two aspects both contribute to the success of MLPMixer, compared with MLPs.

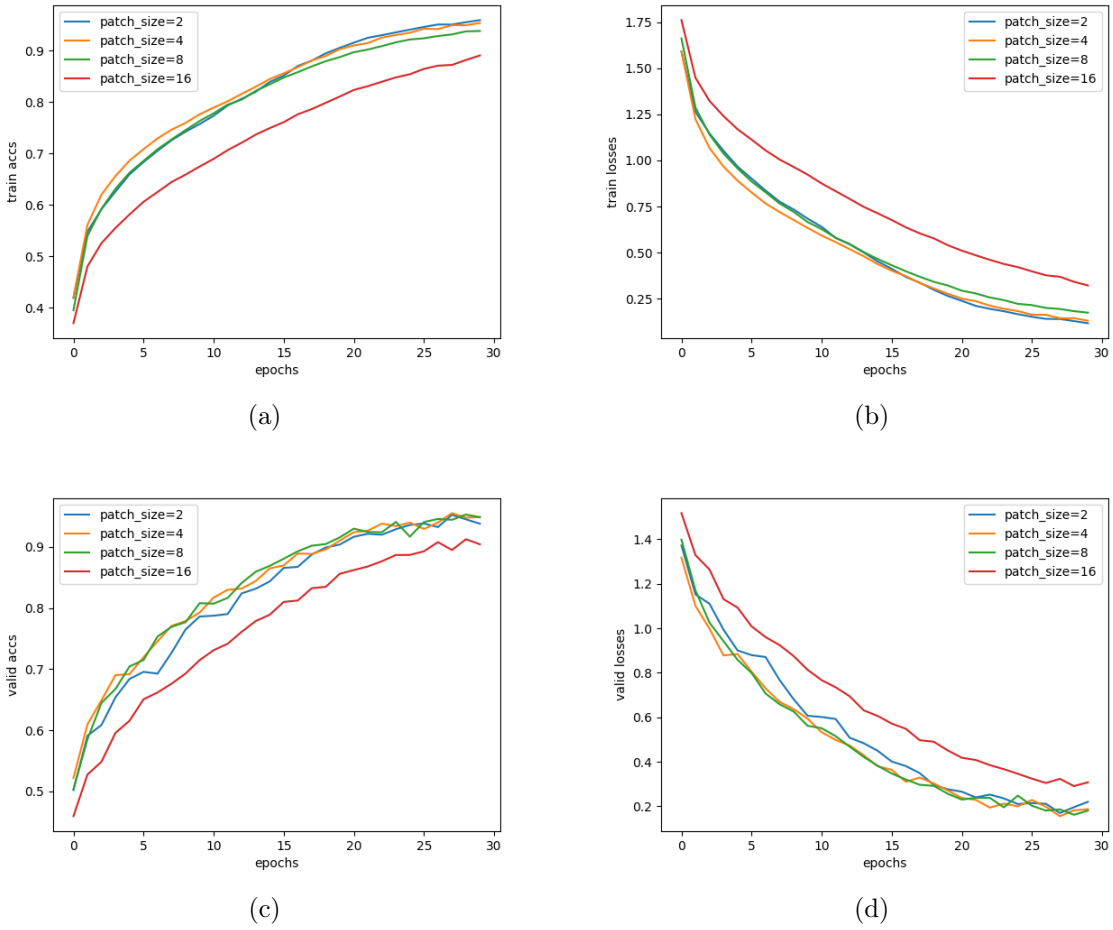


FIGURE 4 – Visualization of the effect of different patch sizes for the MLPMixer.

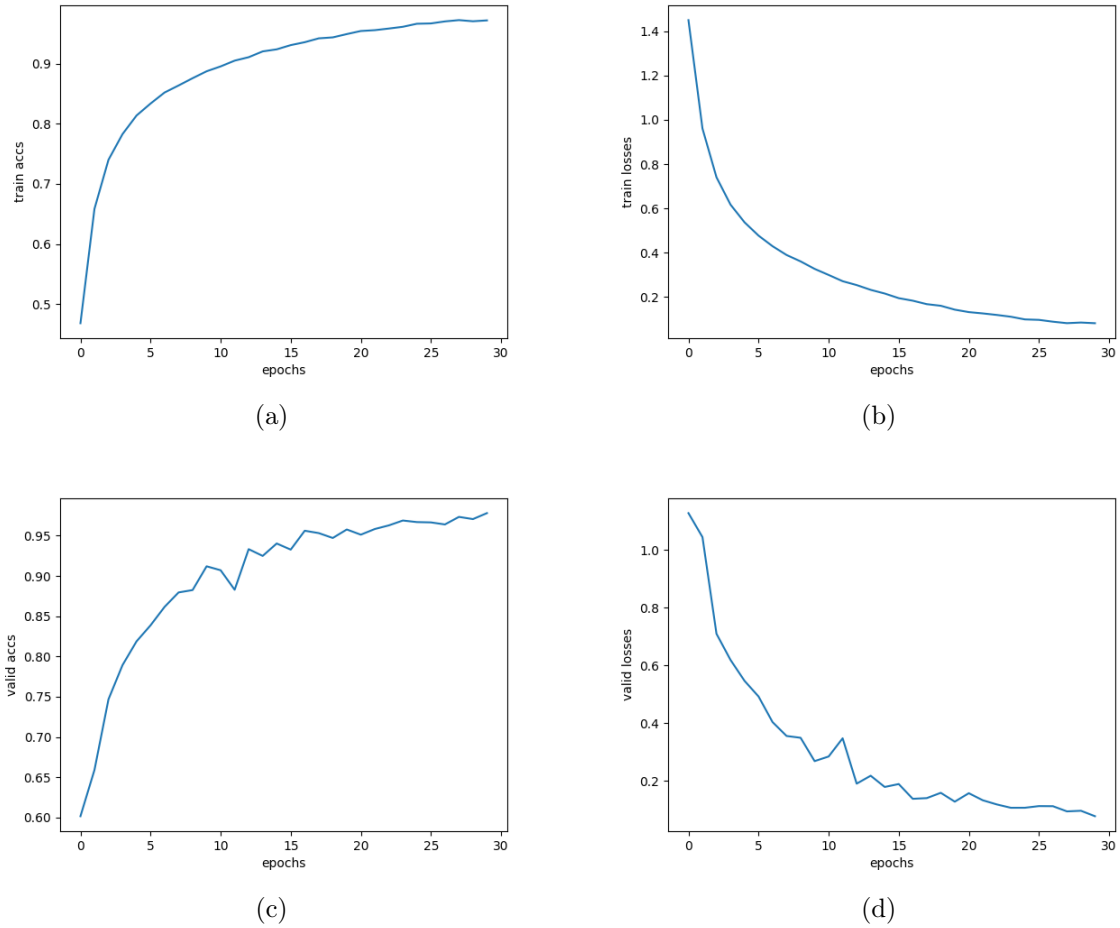


FIGURE 5 – Performance of my best MLPMixer model on training and validation sets.

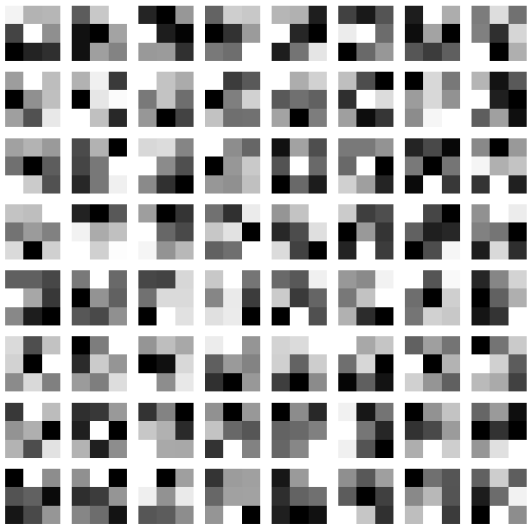
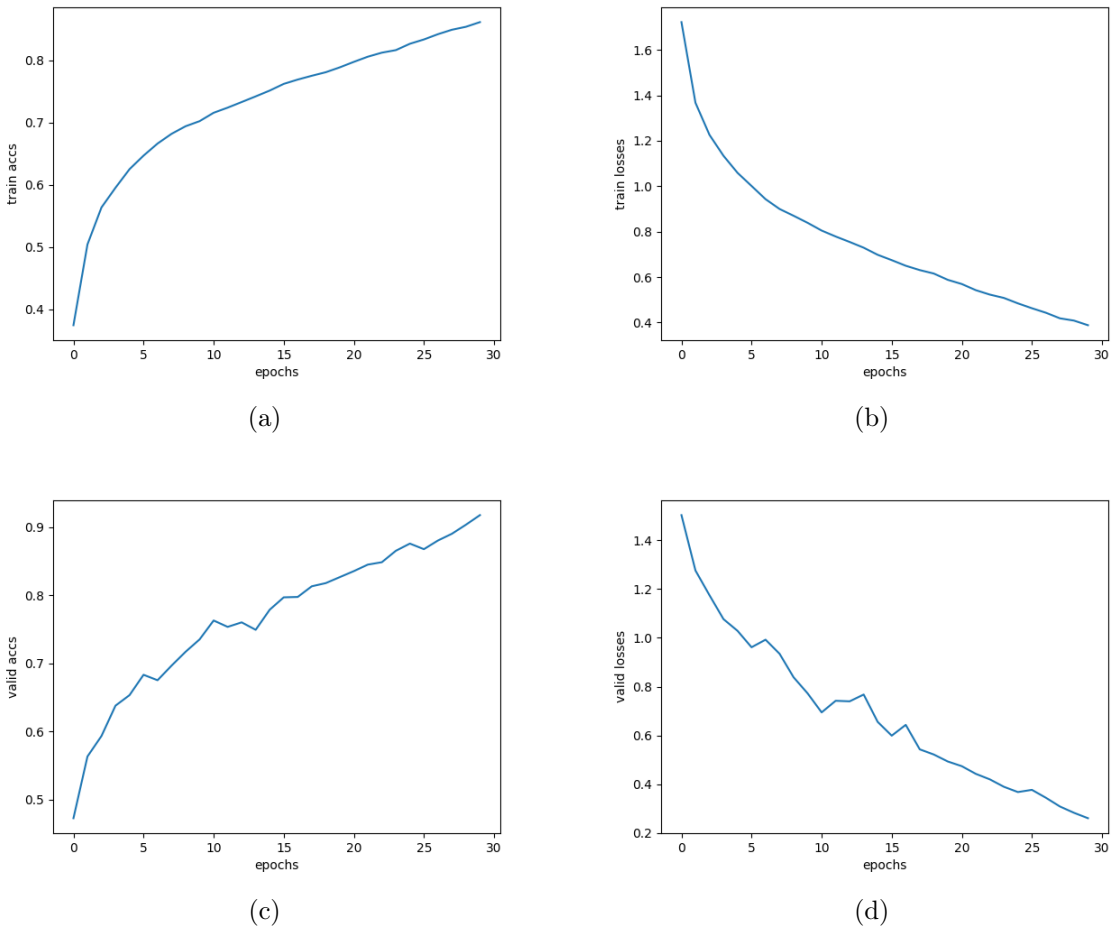


FIGURE 6 – Visualization of the kernels of the first layer in my trained ResNet18.



(a) (b)

(c) (d)

FIGURE 7 – Performance of my best MLPMixer model on training and validation sets.

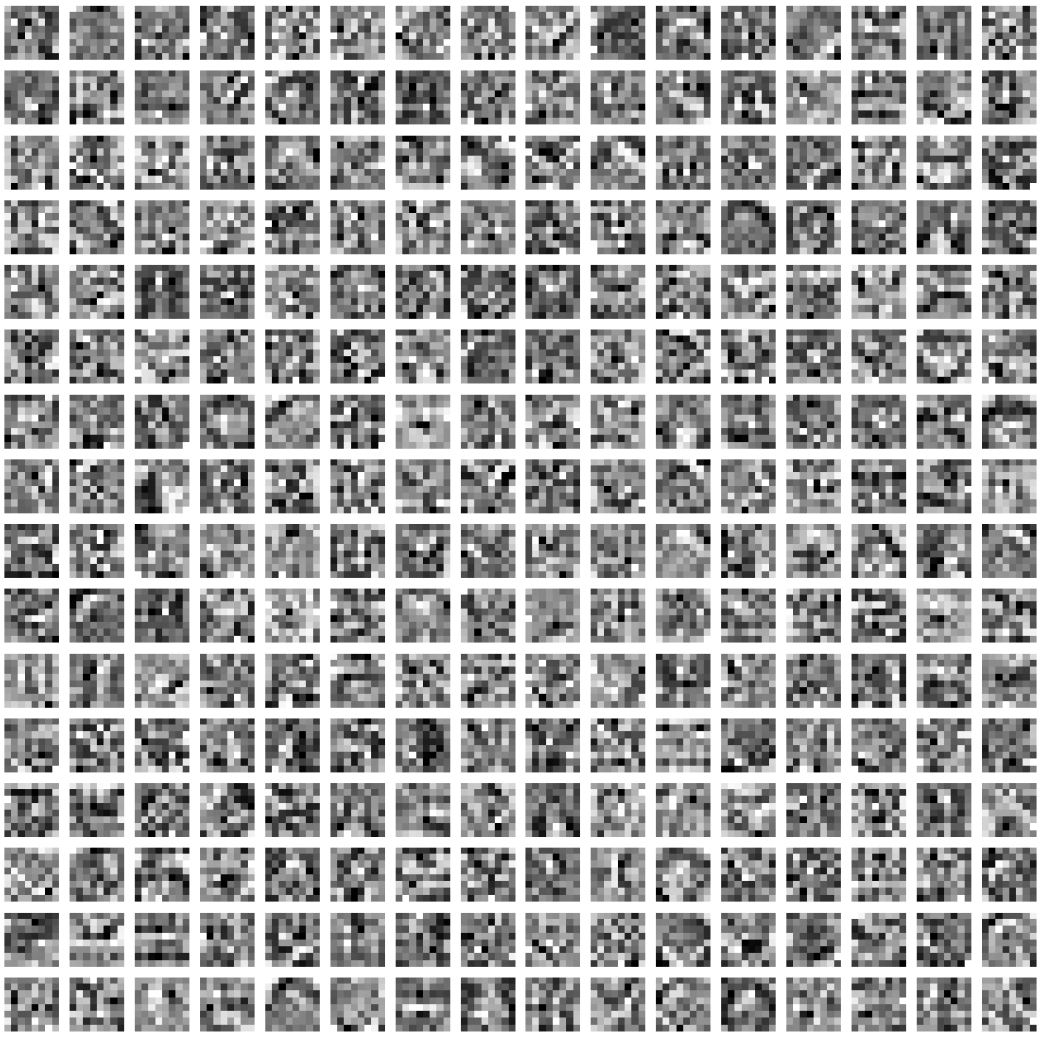


FIGURE 8 – Visualization of the first layer of token-mixing MLP in my trained MLPMixer.