1 Problem4

1.1 2

In this part we investigate the effect of the choice of non-linearity in MLP architecture while keeping the other hyperparameters the same as the default(batch-size= 128, optimizer ='adamw', epochs: 30, lr= 1e-3, momentum = 0.9, weight-decay= 5e-4). we use three different activation functions: Relu, Sigmoid and Tanh and keep track of the accuracy and loss of the architecture in each epoch of the three different activation functions.

In Figure1, 2,3, and 4 we provide four figures corresponding to training loss, validation loss, training accuracy, and validation accuracy, where the x-axis is the number of epochs and y-axis is either loss or accuracy. we further use the legend to denote the non-linearity being used.

Among the three activation functions, ReLU is the best and is generally the most widely used and recommended choice for image classification tasks such as CIFAR10. This is because ReLU is faster to compute than sigmoid and tanh, and has been shown to perform well in deep neural networks and also here in a simple MLP architecture.

For image classification tasks like CIFAR10, ReLU is a good starting point and often performs well. Tanh and sigmoid can also be used, but they may require longer training times and can be more prone to the vanishing gradient problem in deep networks. although, if we want to choose between Tanh and sigmoid, Tanh is generally preferred over sigmoid as it has a steeper gradient and can converge faster during training, even though the plot is saying sigmoid is performing much better than tanh.

1.2 3

We investigate the effect of learning rate with the Adam optimizer for Resnet18 architexture. we perform experiments with learning rates of 0.1, 0.01, 0.001, 0.0001, 0.00001 and in Figure 5, 6, 7, and 8 we provide four figures corresponding to training loss, validation loss, training accuracy, and validation accuracy, where the x-axis is the number of epochs and y-axis is either loss or accuracy. we further use the legend to denote the learning rate being used.

Based on our experiments, we find that a learning rate of 0.001 or 0.0001 works best for the ResNet18 model on the CIFAR10 dataset with the Adam optimizer. A learning rate of 0.1 or 0.01 may be too large and cause the loss to diverge, while a learning rate of 0.00001 may be too small and result in slow convergence.

In the plots, we can see that the model achieves the highest accuracy with a learning rate of 0.001, while larger or smaller learning rates lead to lower accuracy. This result is consistent with what we might expect based on our understanding of the impact of learning rate on optimization.

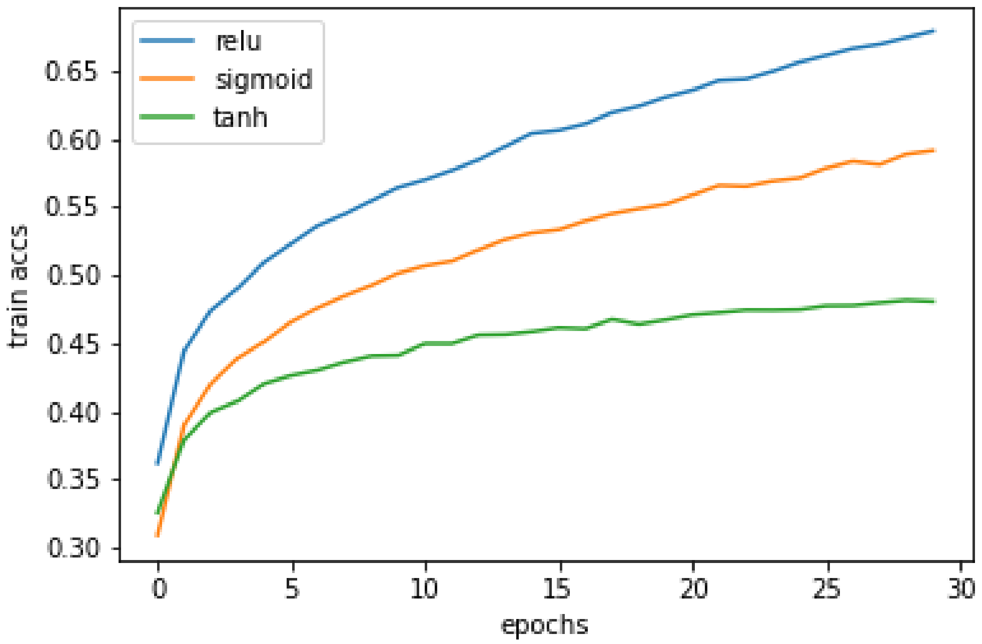

Figure 1: MLP training accuracy

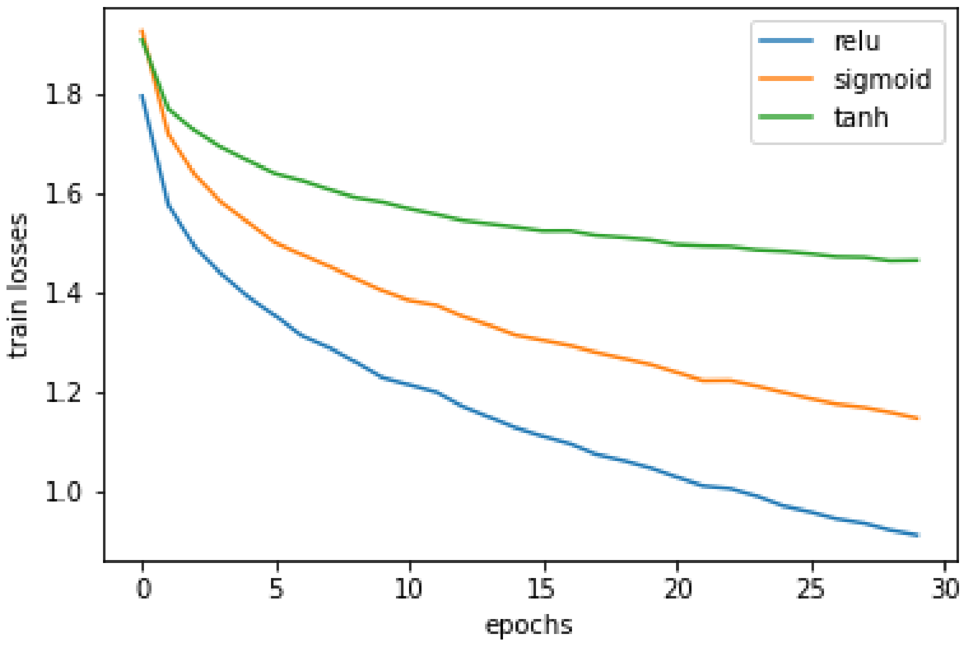

Figure 2: MLP training loss

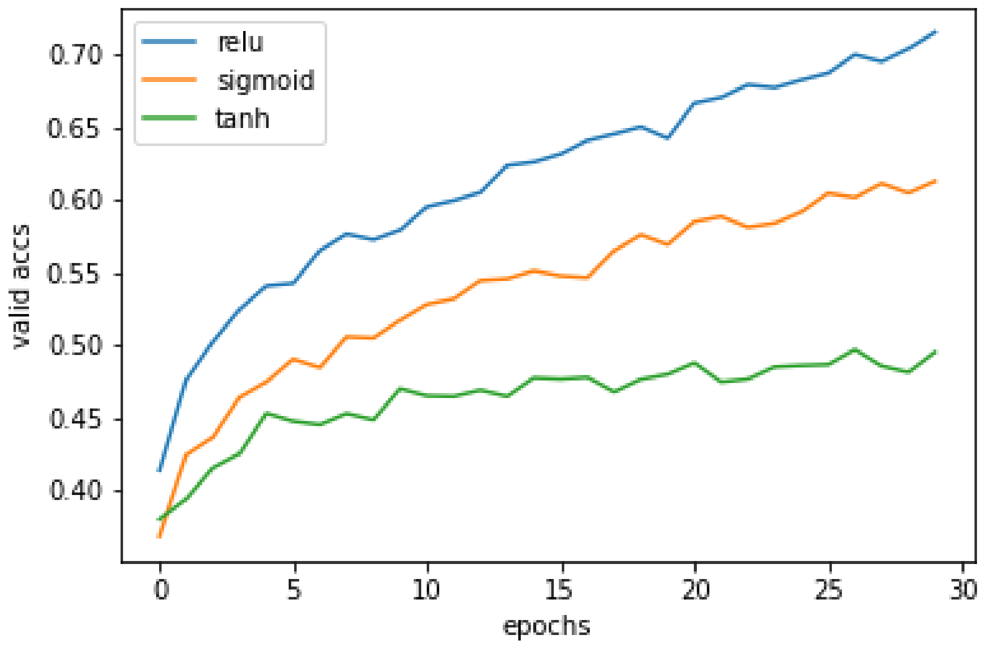

Figure 3: MLP validation accuracy

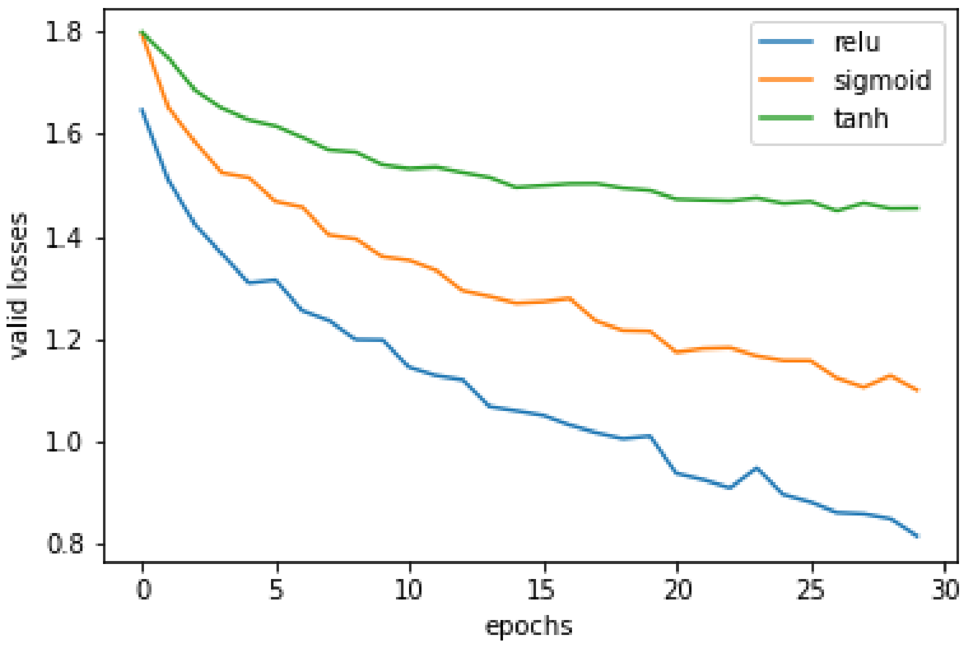

Figure 4: MLP validation loss

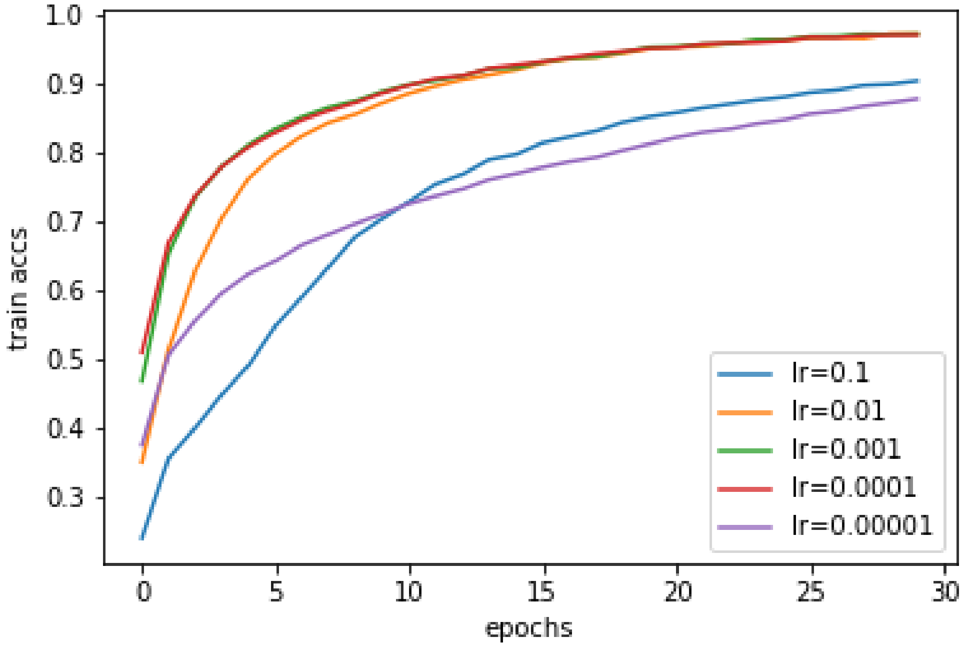

Figure 5: Resnet18 training accuracy

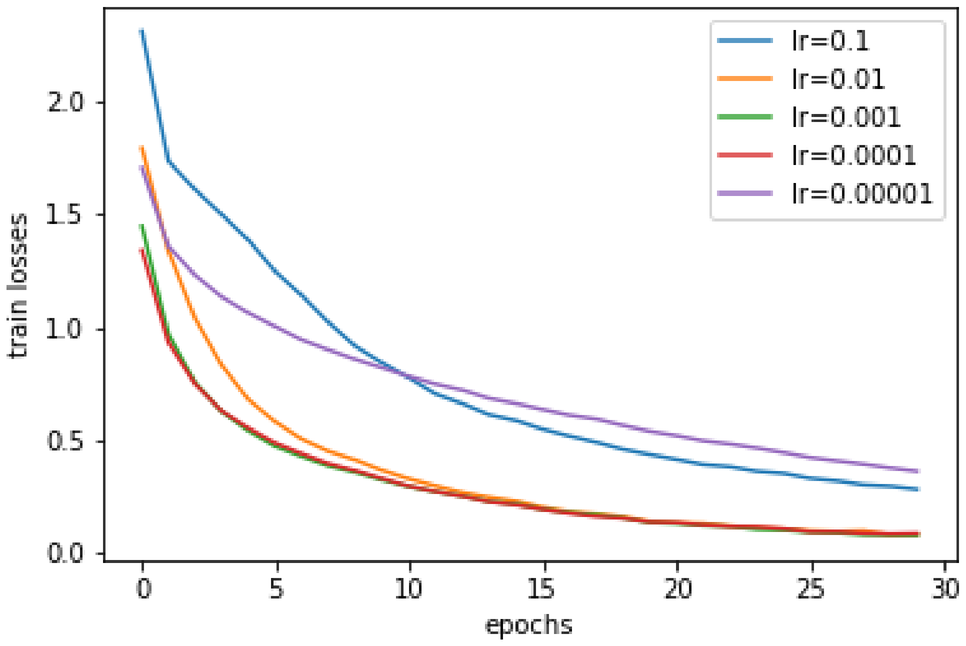

Figure 6: Resnet18 training loss

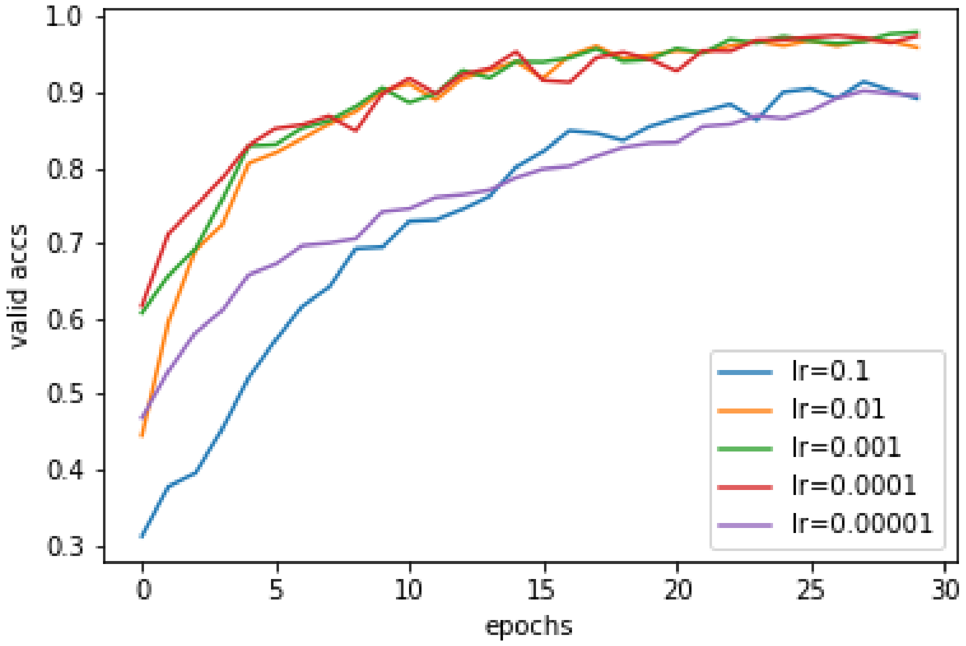

Figure 7: Resnet18 validation accuracy

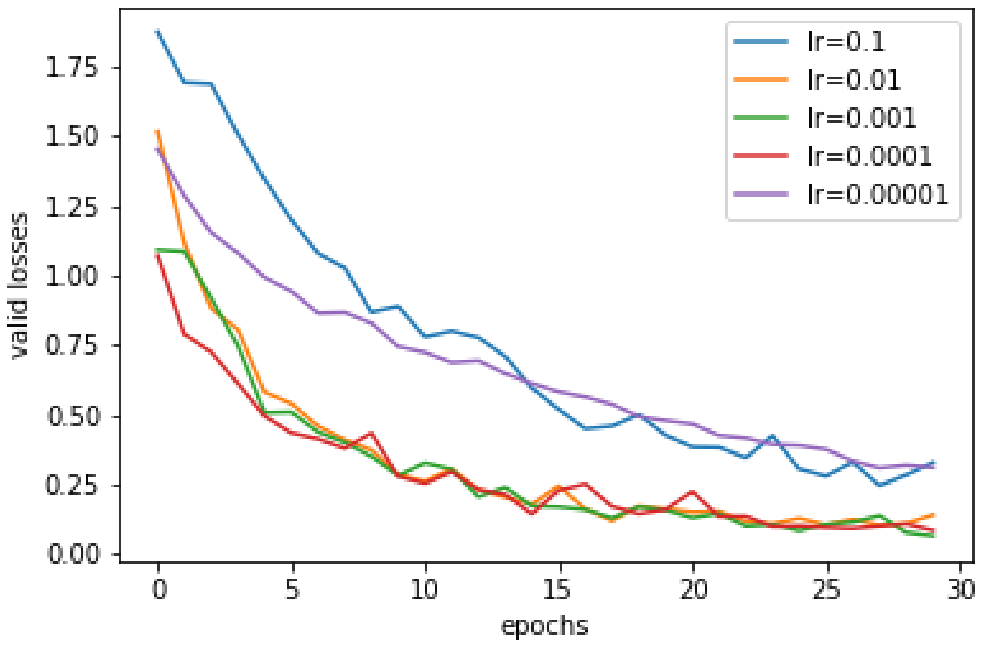

Figure 8: Resnet18 validation loss

1.3 4

we investigate the effect of patch size in mlpmixer. we perform experiments with 3 different patch size (4,8,16). Then in the Figure 9, 10, 11, 12 we provide four figures corresponding to training loss, validation loss, training accuracy, and validation accuracy, where the x-axis is the number of epochs and y-axis is either loss or accuracy. we further use the legend to denote the patch size being used.

After keeping track of number of parameters through each layer we find that, there is a relationship between number of patches and number of parameters which is described below:

all layers consists of:
proj(conv2d)
number of blocks * norm1
number of blocks *[mlp token(Fc1, Fc2)]
number of blocks * norm2
number of blocks *[mlp channel(Fc1, Fc2)]
norm3
Fc3

We count the number of parameter for each layer:
c=hidden dimension p= patch size

$proj.weight = 3cp^2$
$proj.bias = c$
$norm1.weight = c$
$norm1.bias = c$
$mlp - token.Fc1.weight = (\frac{img-size}{p})^2 * \frac{c}{2}$
$mlp - token.Fc1.bias = \frac{c}{2}$
$mlp - token.Fc2.weight = \frac{c}{2} * (\frac{img-size}{p})^2$
$mlp - token.Fc2.bias = (\frac{img-size}{p})^2$
$norm2.weight = c$
$norm2.bias = c$
$mlp - channel.Fc1.weight = c * 4c$
$mlp - channel.Fc1.bias = 4c$
$mlp - channel.Fc2.weight = 4c * c$
$mlp - channel.Fc2.bias = c$
$norm3.weight = c$
$norm3.bias = c$
$Fc3.weight = numberofclass$
$Fc3.bias = numberofclass$

so the total number of parameters is:

$3cp^2 + c + number - of - blocks * [(\frac{img-size}{p})^2 * \frac{c}{2} + \frac{c}{2} + \frac{c}{2} * (\frac{img-size}{p})^2 + (\frac{img-size}{p})^2 + 2c + c * 4c + 4c + 4c * c + c] + 2c + c * number - of - classes$
$= 3cp^2 + 3c + number - of - blocks * [(\frac{img-size}{p})^2 * (c+1) + 8c^2 + \frac{19c}{2}] + 2c + c * number - of - classes + number - of - classes$

As the default parameter we set c=256 and img-size = 32 we have:

total number of parameters = $768p^2 + numberofblocks * [\frac{263168}{p^2} + 526720] + 3338$

for 4 blocks we have for the number of parameters:
patch size = 4: 2188298
patch size = 8: 2175818
patch size = 16: 2310938

patch size =4
number of parameters:
has 2188298 total parameters, of which 2188298 are learnable.

performance:
training the model with the default parameter yields minimum training loss of 0.13499164895454366, minimum validation loss of 0.16779906626325103, maximum training accuracy of 0.9525017806267816, and maximum validation accuracy of 0.9537109374999999.

The model's performance on test set was:
test loss: 0.9951095875305468
test accuracy: 0.7590981012658228

running time:
average training running time was 28.467593216896056
average validation running time was 1.8428839445114136.

patch size = 8
number of parameters:
has total 2175818 parameters, of which 2175818 are learnable.

performance:
training the model with the default parameter yields minimum training loss of 0.17196392852738726, minimum validation loss of 0.15679807299748064, maximum training accuracy of 0.9392806267806265, and maximum validation accuracy of 0.9593749999999998.

The model's performance on test set was:

test loss: 1.0431543513189387
test accuracy: 0.7425830696202529

running time:
average training running time was 26.78816312154134
average validation running time was 1.8016459465026855,
and test run time was 4.9402642250061035

patch size =16
number of parameters:
has 2310938 total parameters, of which 2310938 are learnable.

performance:
training the model with the default parameter yields minimum training loss of 0.3243845482582379, minimum validation loss of 0.29736201912164684, maximum training accuracy of 0.8891337250712258, and maximum validation accuracy of 0.9109375000000002.

The model's performance on test set was:
test loss: 1.2919815061967583
test accuracy: 0.654865506329114

running time:
average training running time was 26.214262564977005
average validation running time was 1.8799697637557984,
and test run time was 2.768500804901123.

As it can be seen in the Figures 13, 14, and 15 there is no direct relation between patch size and number of parameters or patch size and running time but from the Figures it can be seen that patch size = 8 has the lowest number of parameters and the lowest average validation time while for the average training time patch size = 16 has the lowest value and then 8 and finally 4.

1.4 5

In this part we experiment hyperparameter tuning of Resnet18 architecture. We explore model's performance on test and validation sets on a define range of our hyperparameters.

batch size range= [64,128,256,512, 1024]
optimizer range =["sgd", "momentum", "adam", "adamw"]
lr range = [0.01, 0.025, 0.05, 0.075, 0.001, 0.0025, 0.005, 0.0075, 0.0001,0.00025, 0.00075]
momentum range = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
weight decay range = [0.01,0.075,0.001,0.005,0.0075,0.0001,0.00025,0.0005,0.00075,0.00001,0.000025,0.000075]

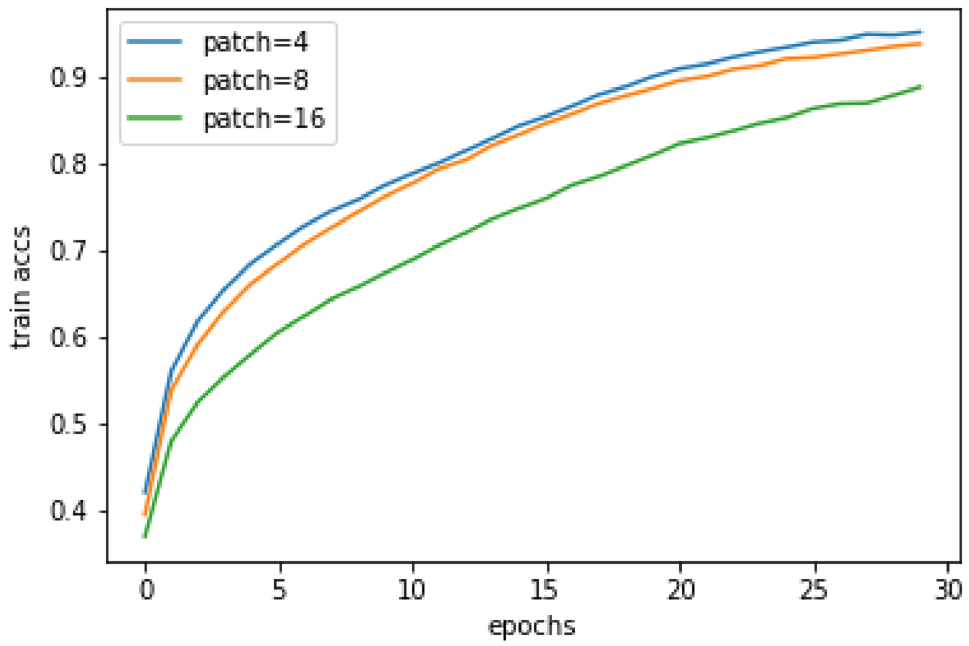

Figure 9: MLPMIXER training accuracy

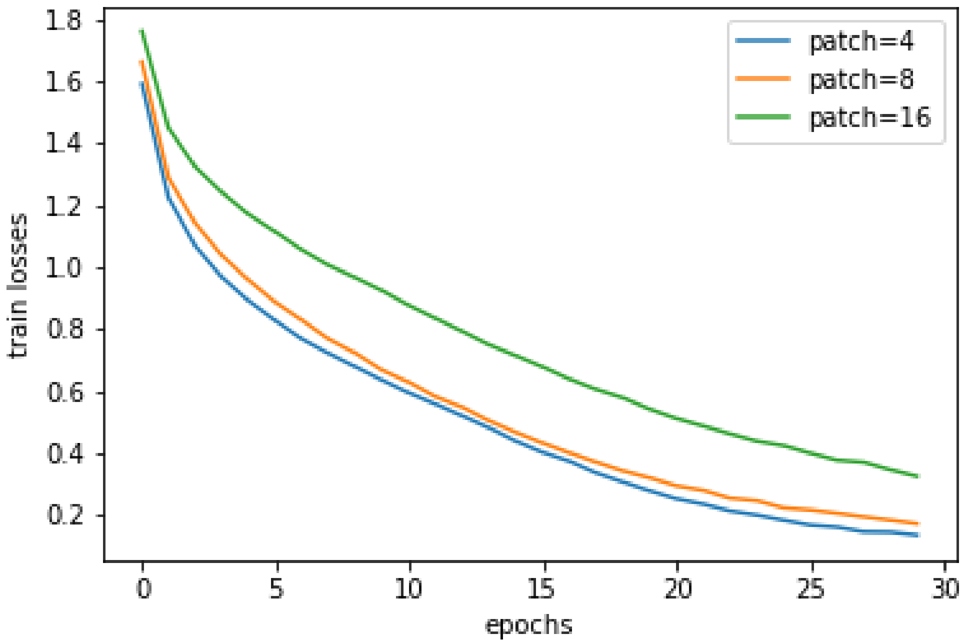

Figure 10: MLPMIXER training loss

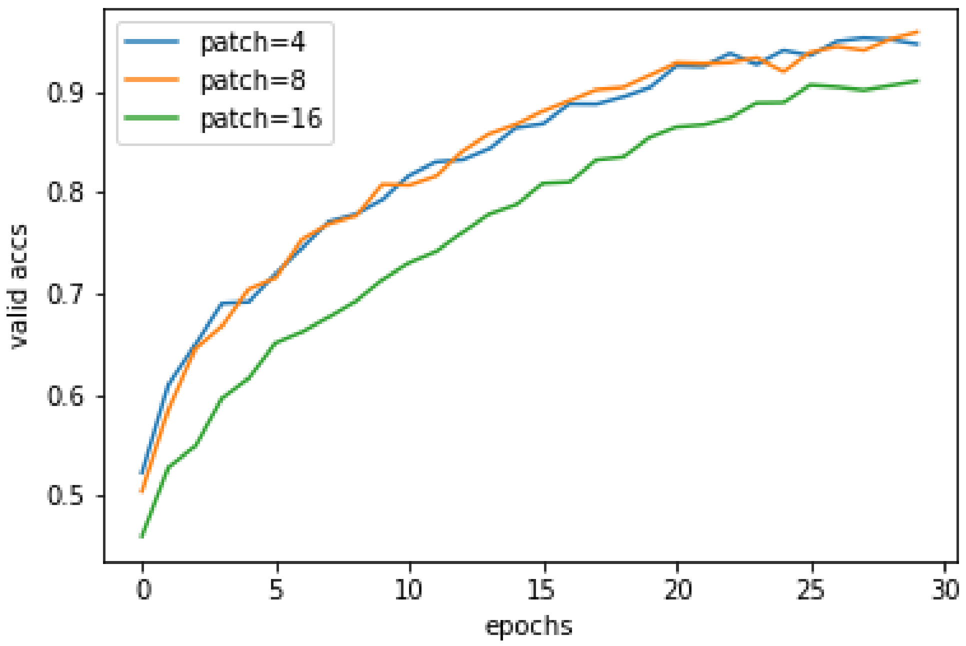

Figure 11: MLPMIXER validation accuracy

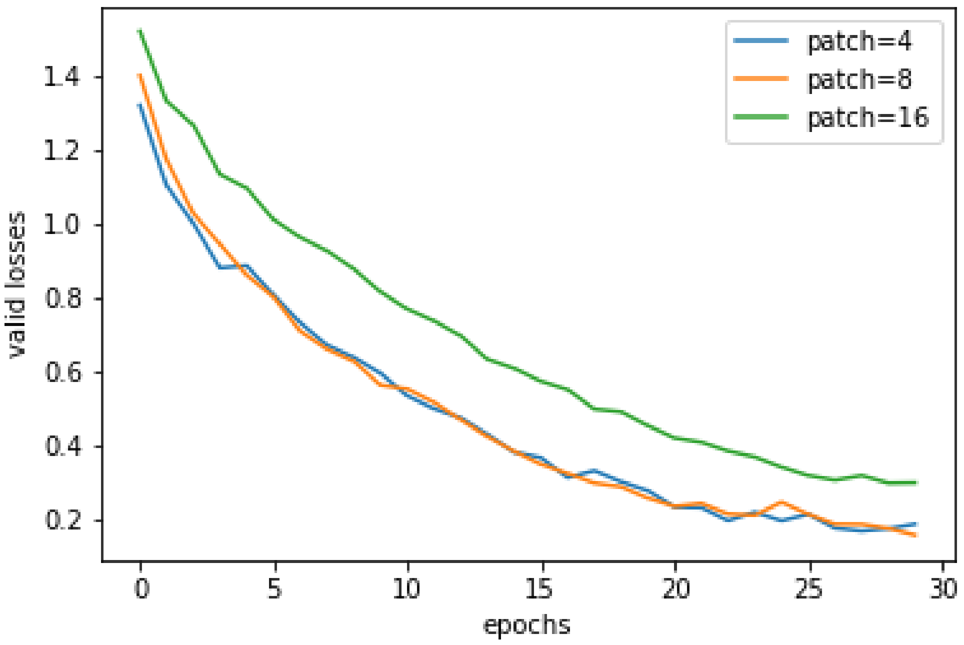

Figure 12: MLPMIXER validation loss

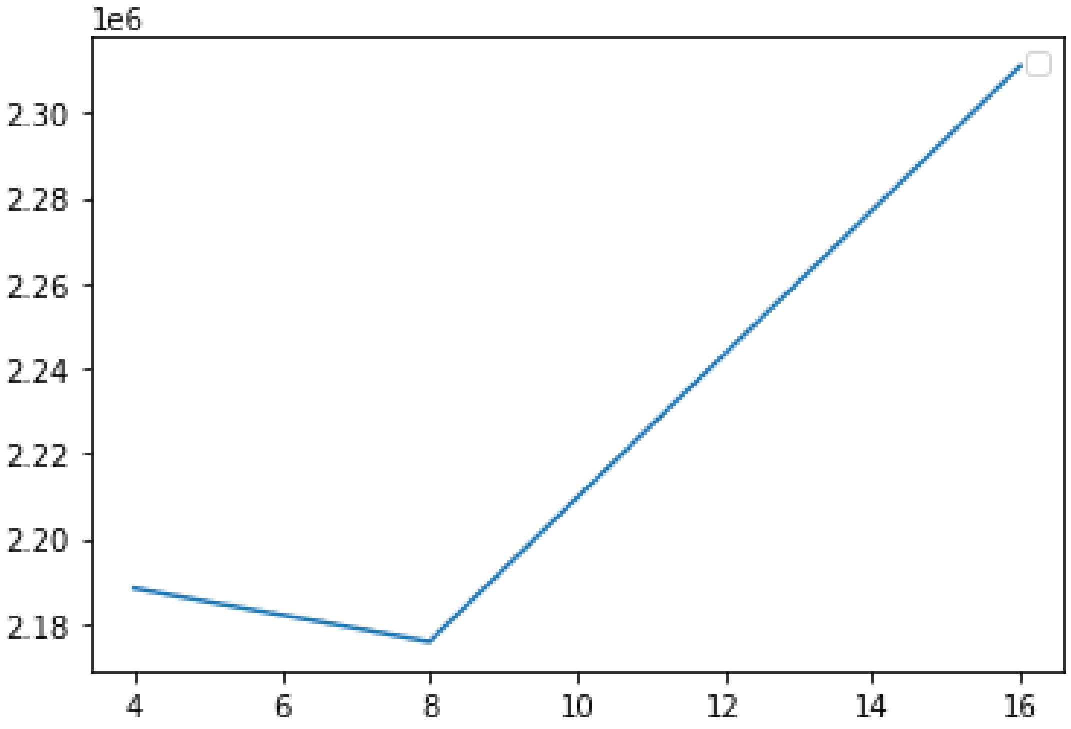

Figure 13: relation between patch size and total number of parameters

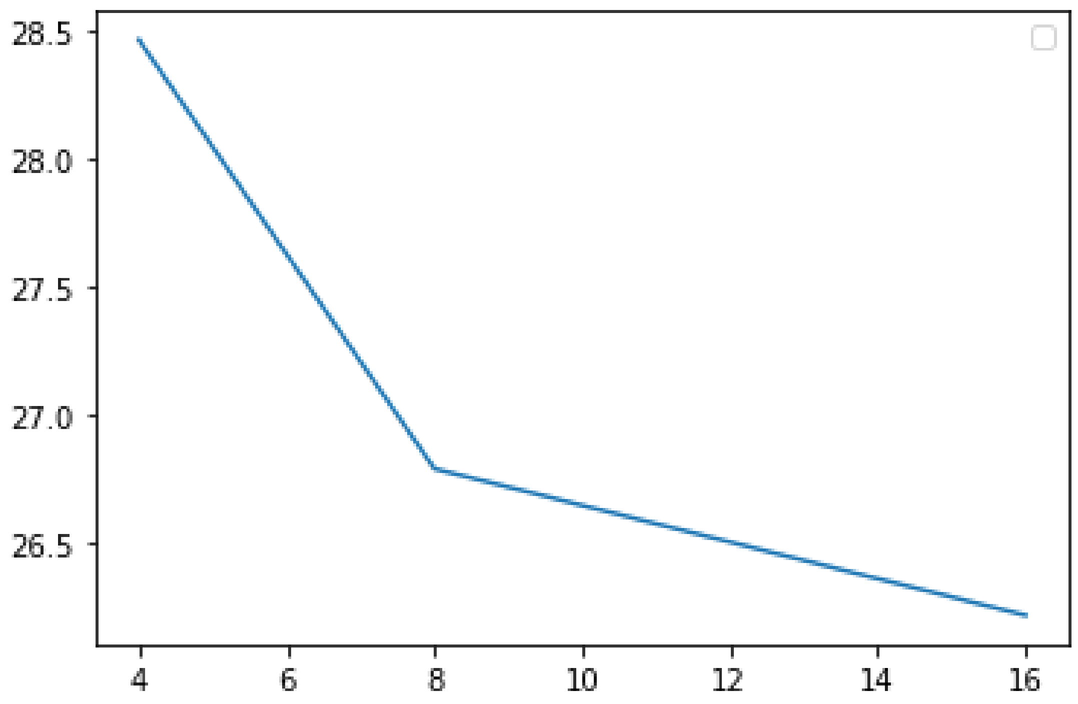

Figure 14: relation between patch size and average running time of training

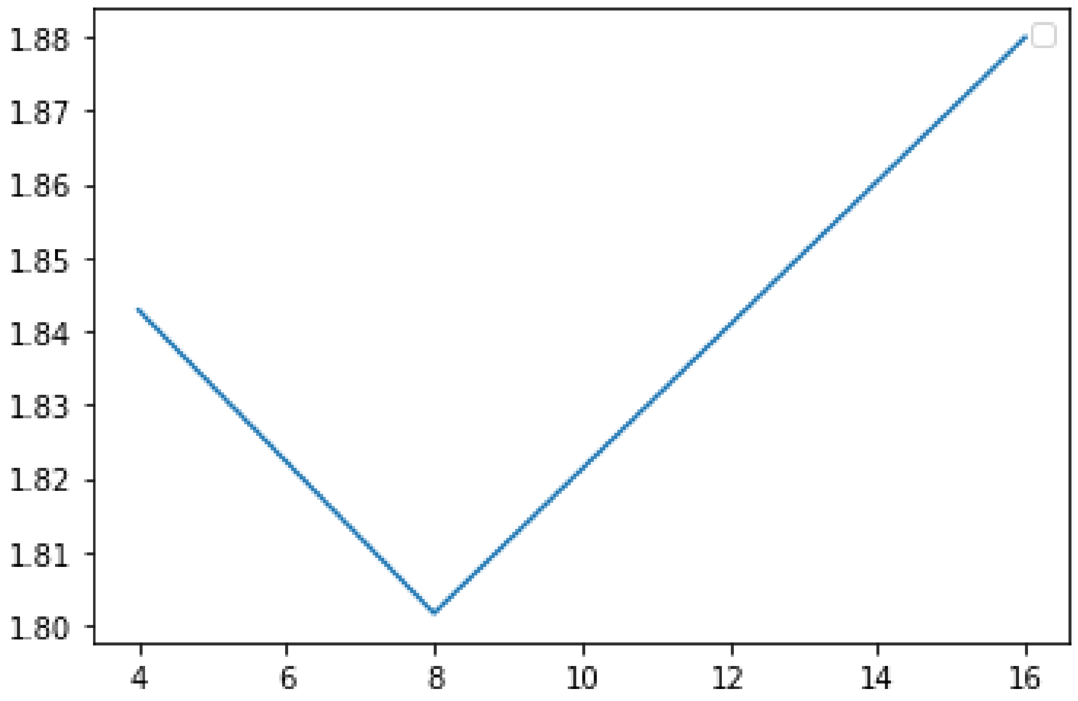

Figure 15: relation between patch size and average running time of validation

The table1 is the models we experiment for hyperparameter tuning of the Resnet18 architecture. We perform over 30 experiments of different settings for the hyperparameters. Table1 shows the 8 of the best ones we got.

model	batch size	learning rate	optimizer	momentum	weight decay	max train accs	max val acc	test acc
model1	64	0.01	sgd	0.6	2.5e-05	0.9700	0.9760	0.8533
model2	64	0.0025	adam	0.3	2.5e-05	0.97844	0.9837	0.8911
model3	512	0.001	adamw	0.6	0.00075	0.9701	0.9553	0.8720
model4	64	0.0075	sgd	0.9	0.00075	0.9671	0.9754	0.8647
model5	512	0.0001	adamw	0.9	0.0075	0.9694	0.9670	0.8794
model6	128	0.0025	momentum	0.5	2.5e-05	0.9345	0.9396	0.8203
model7	128	0.0025	adam	0.9	2.5e-05	0.9781	0.9800	0.8859
model8	128	0.0025	momentum	0.9	2.5e-05	0.9781	0.9800	0.8859

Table 1: hyperparameter tuning of Resnet18 architecture over 8 models with different settings for the hyperparameters.

As it can be seen model2 with the written hyperparameters settings yields the best validation accuracy and the best test accuracy score. Figure 16 is the visualization of the kernel of the first layer of model2 which contains sixty-four 3×3 small images, where each image represent the kernel corresponding to the 64 output channels. After getting the weights of the first convolutional layer, we standardize the weight values between 0 and 1 and then simply visualize the kernels as small images. Figure 17 is after applying averaging across the channels to have gray scale images.

the type of features learned in the first layer of ResNet18 by the visualization of the kernels are likely to be simple visual patterns and edges that are common in natural images of objects in the CIFAR-10 dataset.

As we already knew, CIFAR-10 is a dataset of 32 x 32 RGB images of 10 different classes, such as airplane, automobile, bird, cat, deer, dog, frog, horse, ship, and truck. The images are relatively small and contain simple shapes and textures, which means that the kernels of the first layer of ResNet18 are likely to be specialized in detecting edges, corners, and textures that are relevant to these objects.

For example, some features may be sensitive to edges and corners that are common in the shape of airplanes or cars, while others may be sensitive to textures that are common in the fur of cats or dogs. The kernels may also be sensitive to color gradients or patterns that are common in the background or foreground of the images.

1.5 6

In this part we experiment hyperparameter tuning of MLPMIXER architecture. We explore model's performance on test and validation sets on a define range of our hyperparameters.

batch size range= [64,128,256,512, 1024]
optimizer range =["sgd", "momentum", "adam", "adamw"]

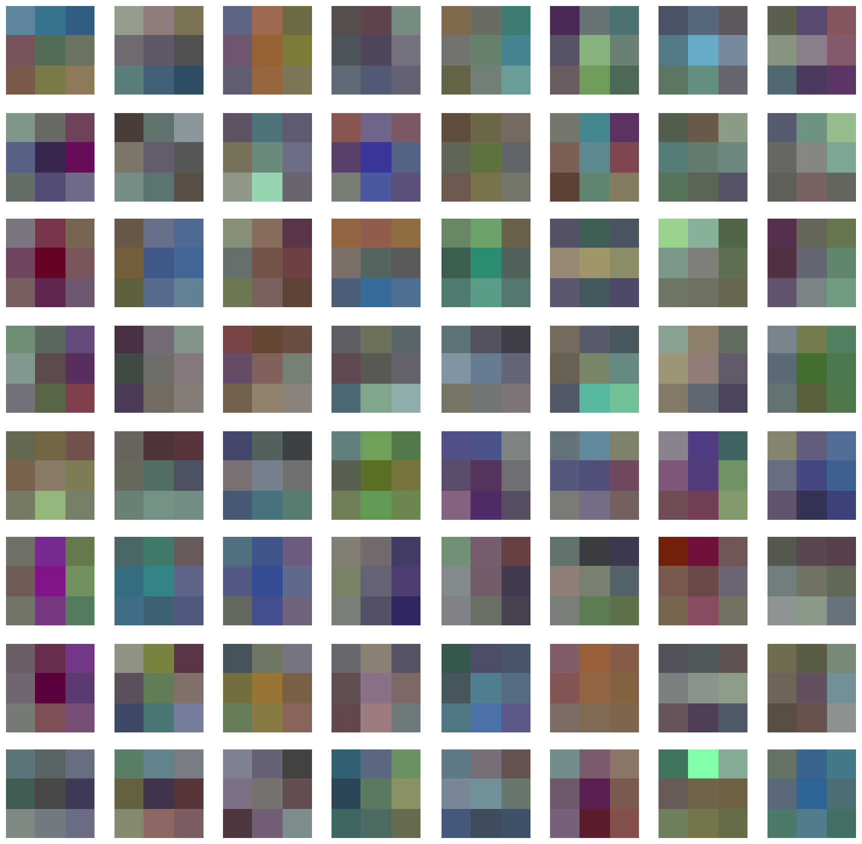

Figure 16: visualization of the kernel of the first layer of Resnet18

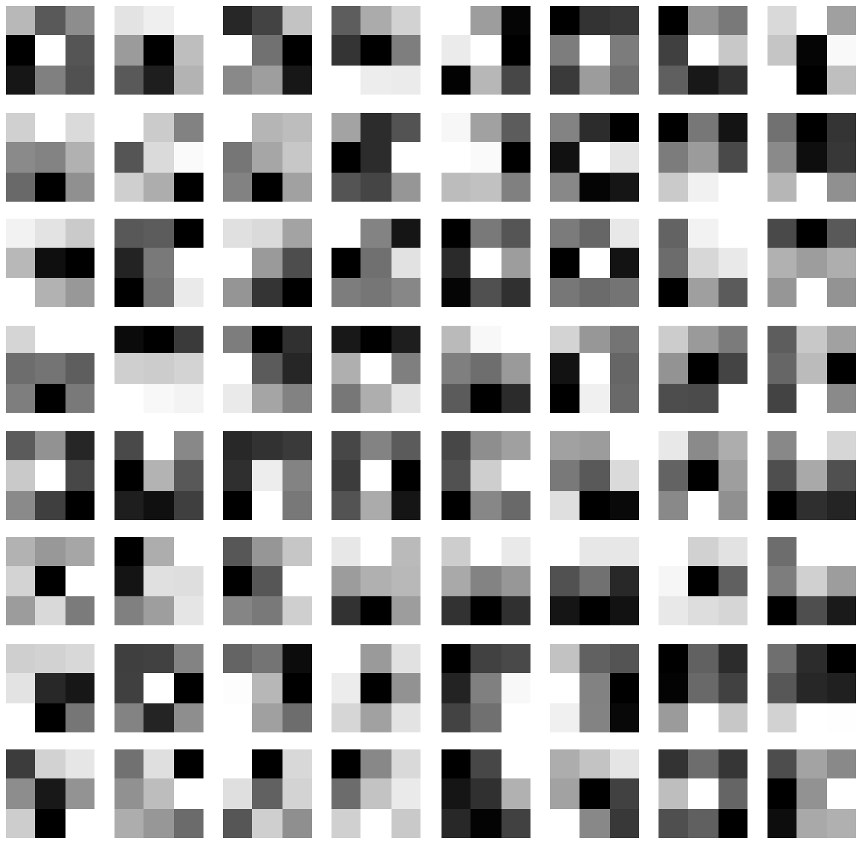

Figure 17: gray scale visualization of the kernel of the first layer of Resnet18

lr range = [0.01, 0.025, 0.05, 0.075, 0.001, 0.0025, 0.005, 0.0075, 0.0001,0.00025, 0.00075]
momentum range = [0.1,0.2,0.3,0.4,0.5,0.6,0.7,0.8,0.9]
weight decay range = [0.01,0.075,0.001,0.005,0.0075,0.0001,0.00025,0.0005,0.00075,0.00001,0.000025,0.000075]
embed dim range =[64 : 300]
num blocks range =[2,3,4,5,6]
drop rate range =[0.1,0.2,0.3,0.4]

The table below is the models we experiment for hyperparameter tuning of the MLPmixer architecture. We perform over 30 experiments of different settings for the hyperparameters. Table1 shows the 8 of the best ones we got.

model	batch size	learning rate	optimizer	momentum	weight decay	embed-dim	number of blocks	drop rate	max train accs	max val acc	test acc
model1	128	0.0075	adam	0.7	0.01	128	4	0.2	0.7578	0.7941	0.7289
model2	128	1e-3	adamw	0.9	5e-4	256	4	0.0	0.95250	0.9537	0.7590
model3	32	0.001	adamw	0.9	0.0005	256	4	0.0	0.9099	0.9124	0.7637
model4	128	0.005	adamw	0.85	0.005	64	4	0.2	0.7404	0.784	0.7339
model5	64	0.005	adamw	0.95	1e-05	150	4	0.0	0.8715	0.8892	0.74452
model6	64	0.00075	adam	0.8	0.0001	275	4	0.0	0.95208	0.9501	0.76054
model7	64	0.0001	adam	0.95	7.5e-05	200	4	0.0	0.801	0.813	0.722
model8	128	0.00075	momentum	0.8	2.5e-05	200	3	0.0	0.92784	0.9343	0.756230

Table 2: hyperparameter tuning of MLPmixer architecture over 8 models with different settings for the hyperparameters.

As it can be seen model3 with the written hyperparameters settings yields the best test accuracy score. Figure 18 is the visualization of the first layer of token-mixing MLP in the first block of model3 which contains 128 8×8 small images. After getting the weights of the first convolutional layer, we standardize the weight values between 0 and 1 and then simply visualize the kernels as small images.

the visualization of the weights of the token-mixing MLP in the first block of the MLPmixer architecture for CIFAR10 dataset might represent higher-level and more abstract features that are relevant to the objects in the CIFAR10 dataset.

The MLPmixer architecture is designed to process the input image as a sequence of patches, which are then transformed by the token-mixing MLP to generate a sequence of higher-level representations. The weights of the token-mixing MLP determine how the information is mixed across different patches and which features are emphasized in the sequence of representations.

In the case of the CIFAR-10 dataset, the features that are learned by the token-mixing MLP in the MLPmixer architecture might be related to the visual characteristics of the 10 object classes in the dataset, such as shapes, colors, textures, and patterns. For example, some weights in the token-mixing MLP might be sensitive to specific combinations of edges, corners, and textures that are common in certain classes of objects, such as cars or airplanes.

The MLPMixer replaces convolutional layers with a series of matrix multiplication and non-linear activation functions (MLP). This approach gained several success over other models.

One of the main reasons behind the success of the MLPMixer over other

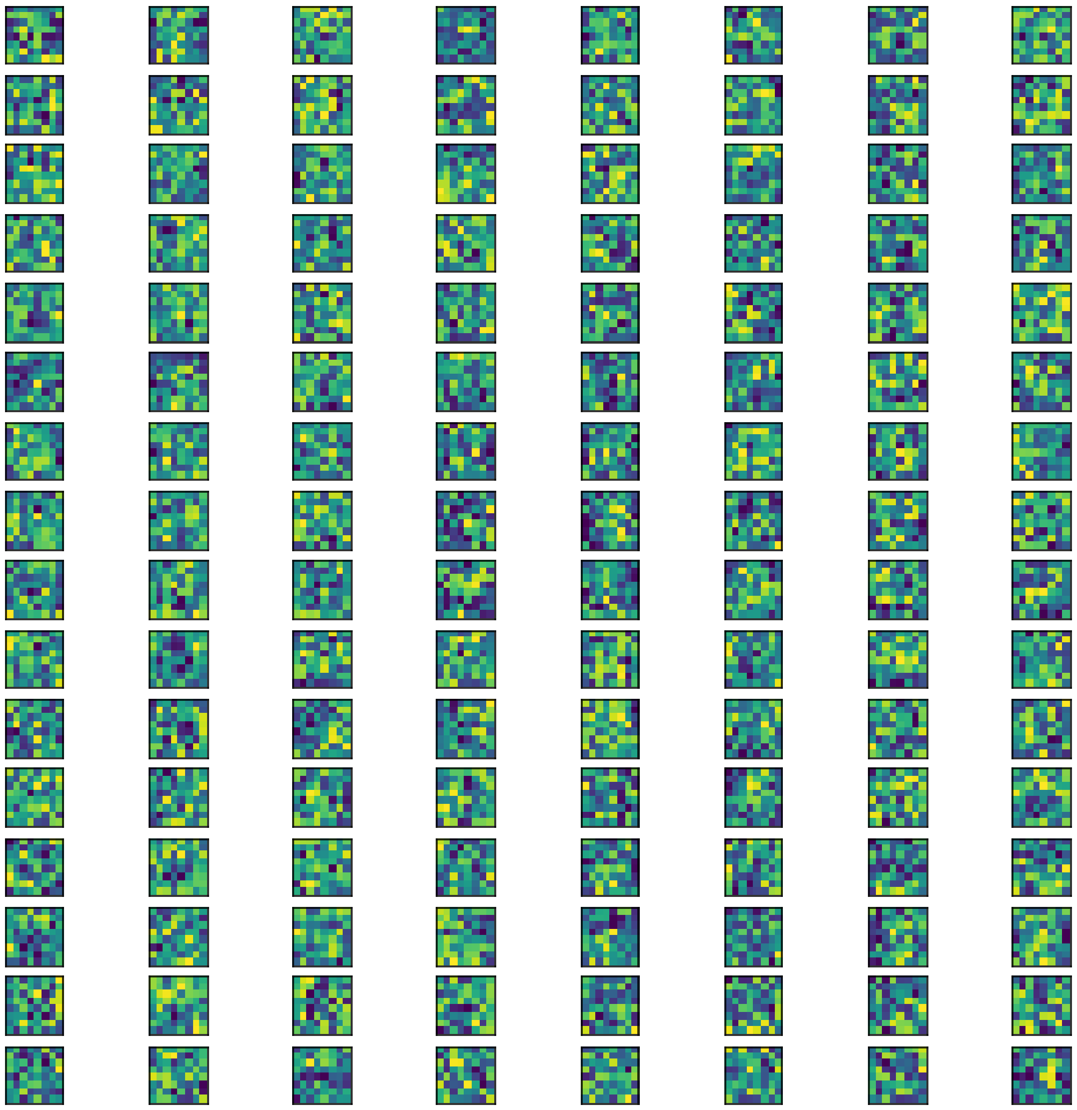

Figure 18: visualization of the first layer of token-mixing MLP in the first block

models especially normal MLP is its ability to effectively capture spatial information in images. While traditional MLPs are commonly used for sequential data processing, they are not well-suited for image data due to the lack of spatial awareness. In contrast, the MLPMixer uses token mixing, which mixes the information across the spatial and channel dimensions of the input, allowing the model to capture both local and global features of the input data. This allows the model to better understand the structure of images and achieve higher accuracy on image classification tasks.