Due Date: October 21th, 2023 at 11:00 pm

Link to Notebook: Click here.

Question 1 (30). (Implementing Perceptron and MLP with NumPy)

(1.1) Implement a perceptron, follow the provided Colab Notebook for instructions. (5)

(d) Using the provided data and graph function, train the perceptron for 3 epochs using the learning rate of 0.001. Show your results (decision boundary graph):

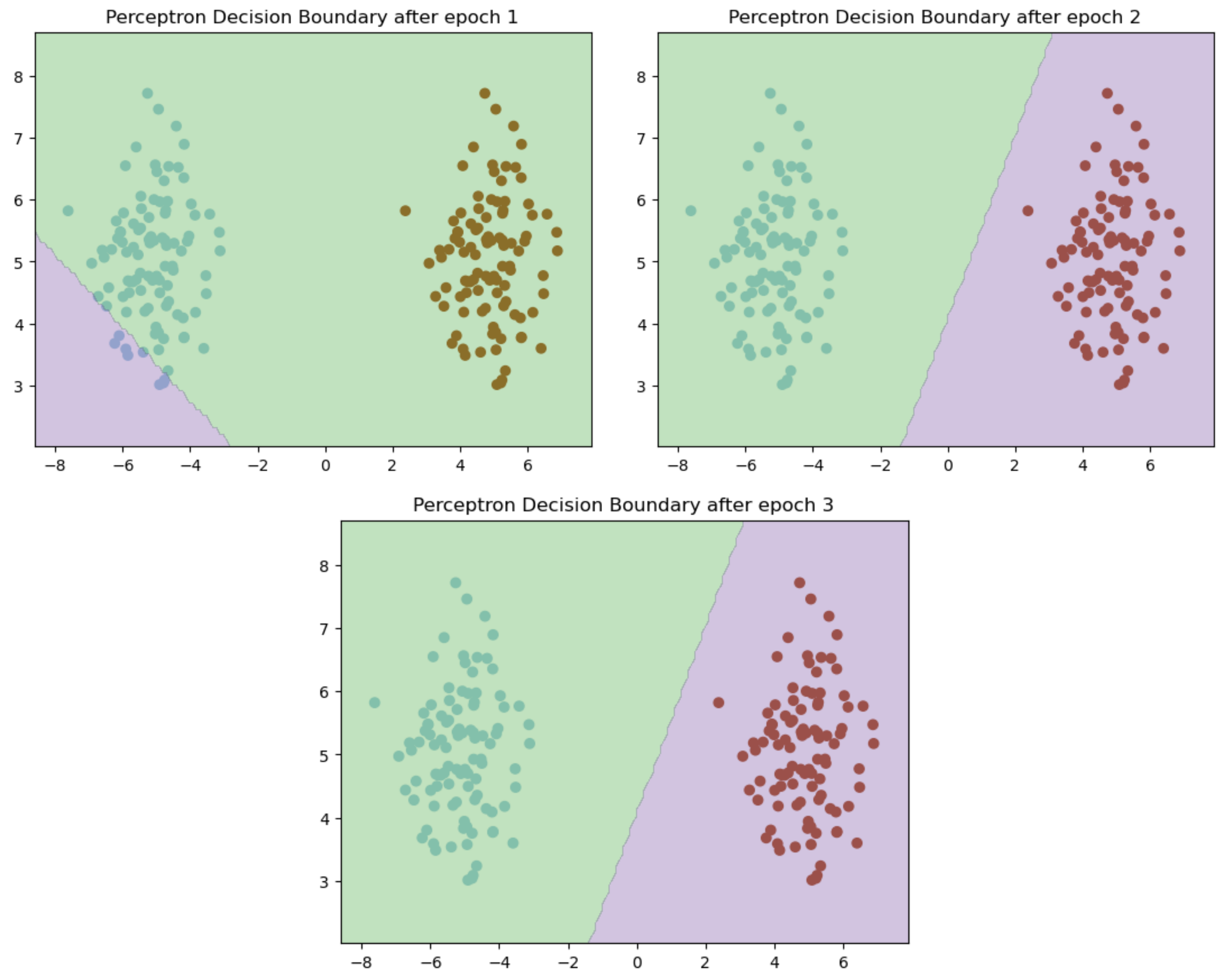
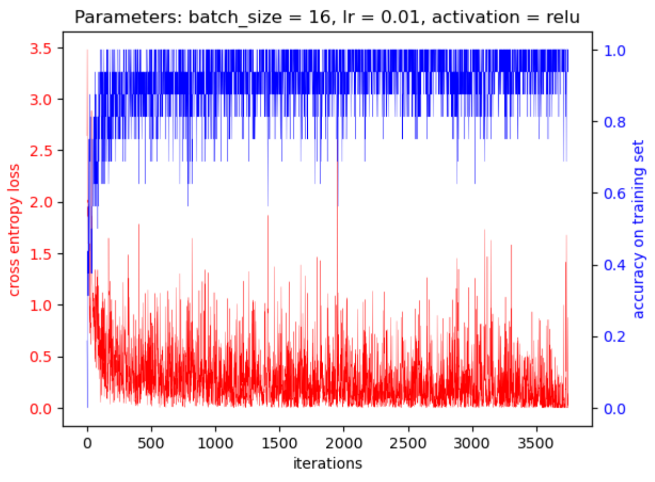


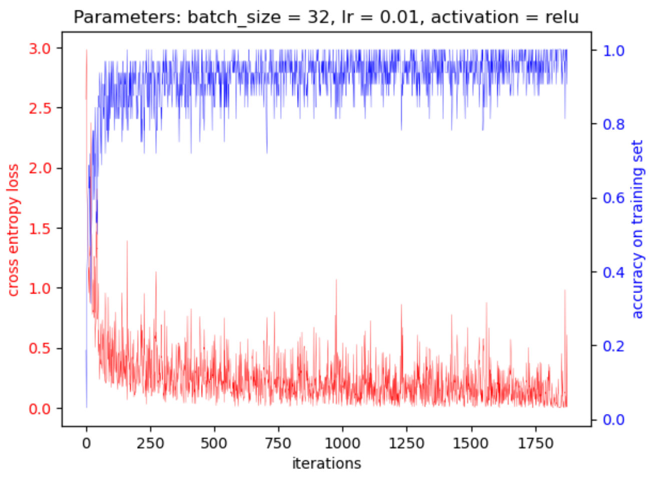
Figure 1 – Perceptron boundary after 3 epochs

(1.2) Implement MLP, follow the provided Colab Notebook for instructions. (25)

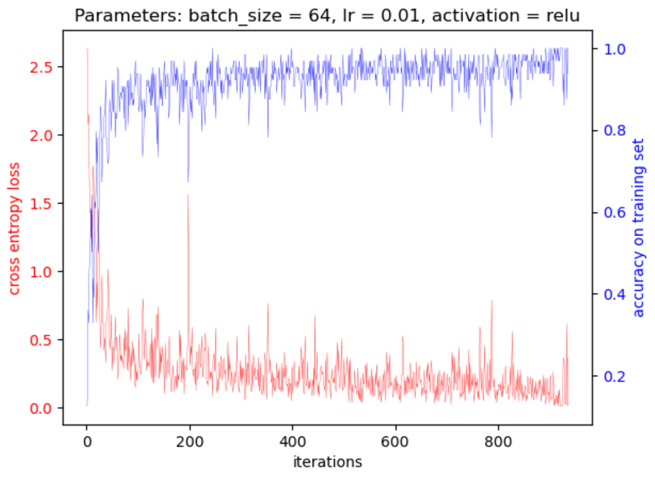
(j) Graphs:



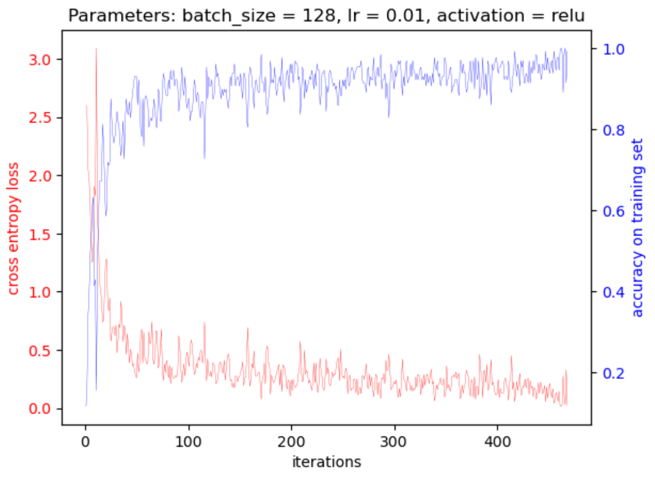
(a) Batch Size: 16



(b) Batch Size: 32



(c) Batch Size: 64



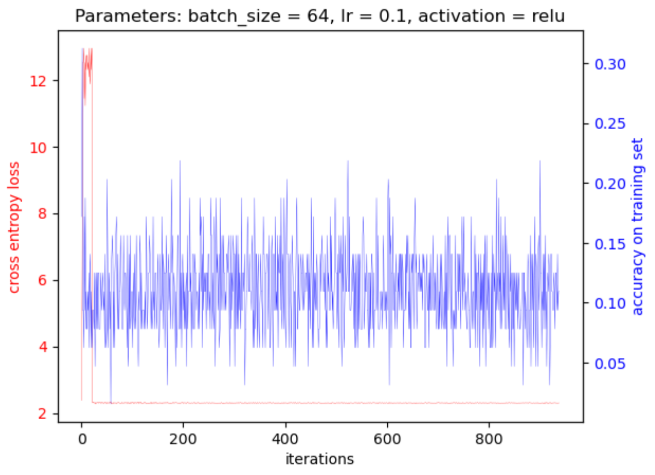
(d) Batch Size: 128

FIGURE 2 – Training graphs for different batch sizes with default learning rate of 0.01 and ReLu as activation function.

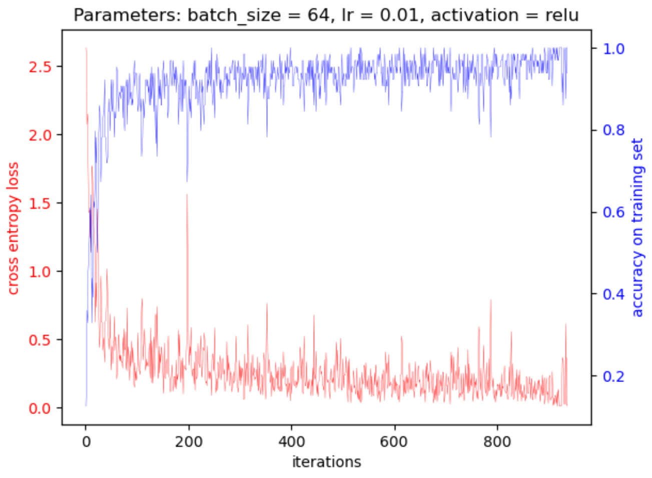
Observations: After a short training duration, all models show signs of rapid overfitting on the training data. We notice a fast convergence to high training accuracy and low loss.
Furthermore, the smaller batch sizes, it gets noiser during training process. This is clearly observed in batch size 16 through harsh fluctuations in both loss and accuracy : past the 3500th iteration, we can see loss spikes nearing 1.75 and corresponding dips in accuracy to about 0.75.

As the batch size increases, the training process becomes more stable and smooth. The larger batches, like 128, we notice minimal oscillations, providing a more predictable training trajectory.
Despite this, all models seem to achieve peak accuracy and bottom out their losses at the same rate.

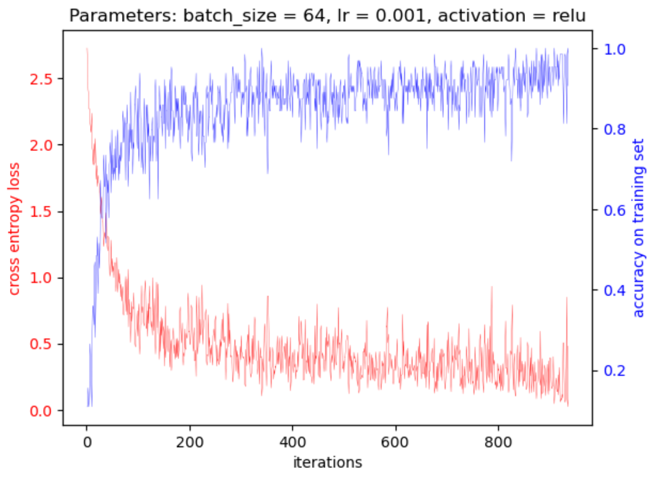
(k) Graphs:



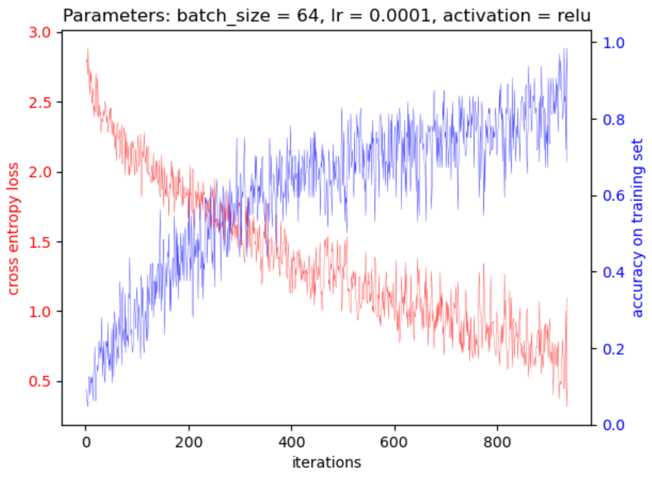
(a) Learning Rate: 0.1



(b) Learning Rate: 0.01



(c) Learning Rate: 0.001



(d) Learning Rate: 0.0001

Figure 3 – Training graphs for different learning rates with default batch size of 64 and ReLu as activation function.

Observations: At a high learning rate of 0.1 we have significant oscillations at a bad accuracy not going over 0.20, and the loss shoots up to 12 initially to stagnate at juste above 2. Clearly it overshot the optimal point, it will actually never reach it since there are convergence issues because it oscillates around the point.

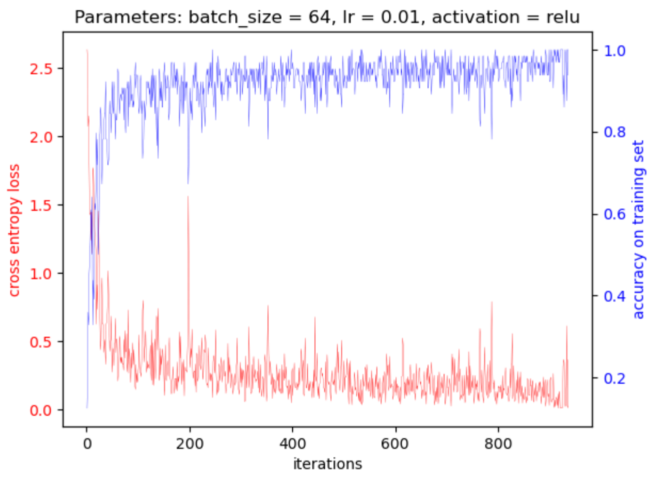
Going lower with a rate of 0.01, as we expected considering the previous rate being clearly too big, it is more stable. After just a few iterationns, it rapidly converges and seems to reach nearly a perfect accuracy and perfect loss after >800 iterations.

At a learning rate rate of 0.001, we have a much smoother descent in training loss but with a higher ocsillation than 0.01. It learns at a steady pace and still manages to raech near 1 accuracy and 0 loss after 800 iterations, but the convergence, as expected, is slower than with a slightly higher learning rate.

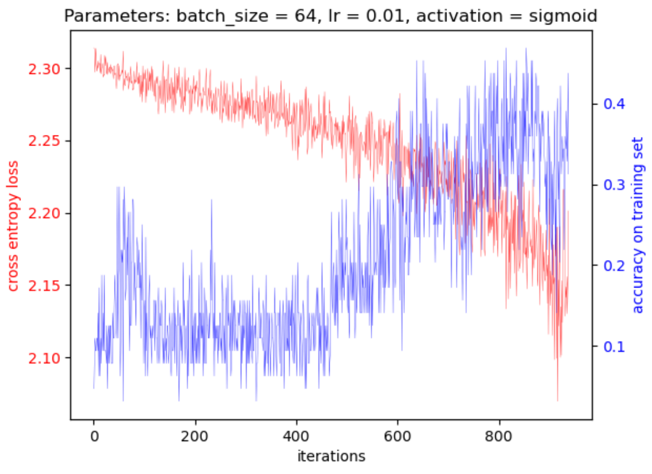
At our lowest learning rate of 0.0001, the decrease in training loss is very gradual. It doesn't converge at much as 0.01 and 0.001 and has a higher ocsillation once again but the difference from 0.001 to 0.0001 is less noticeable than from 0.01 to 0.001.

In conclusion, higher learning rate will lead to fast but potentially unstable training, while a low learning rate might result in very steady descent but slow convergence. From these observations, a learning rate of 0.01 seems to be the most effective on this particular MLP architecture and dataset.

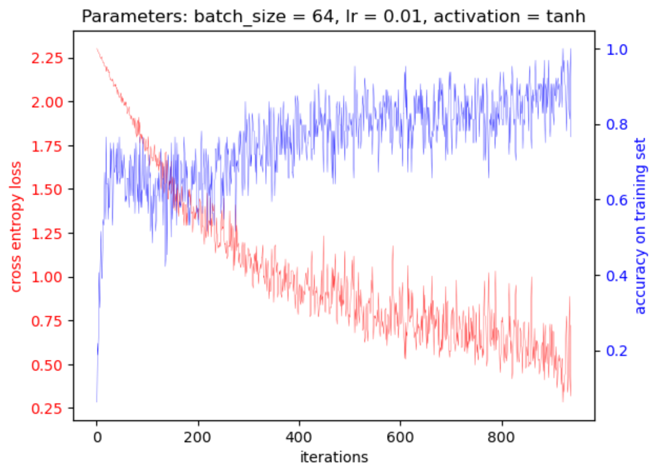
(l) Graphs:



(a) Activation: ReLu



(b) Activation: Sigmoid



(c) Activation: Tanh

FIGURE 4 – Training graphs for different learning rates with default batch size of 64 and ReLu as activation function.

Observations: We've already gone over the model with ReLu as activation, batch size of 64 and learning rate of 0.01 a few times. Just like before before we see an effective learning.

For the sigmoid activation, the loss ends at a much higher value after 800 iterations of around 2.1. There seems to be a start of fluctuations reaching the end, but a general downward converging trend indicates learning is occurring. The accuracy shows fluctuations, but the trend is upward, indicating that the model's predictions are improving. However, at the start we see a rise then drop, after that is stagnates then rises and end up by dipping again. Furthermore, the accuracy does not reach above 0.4. All of this shows us that using the sigmoid is not ideal in our scenario.

For the tanh activation the loss initially decreases significantly and then shows slight fluctuations, but doesn't converge as much as we would like after 800+ iterations and only finishes at around 0.25. The training accuracy increases noticeably at the beginning and then shows minor fluctuations. While the model with tanh activation is learning, it might not be as stable as the one with ReLU activation but definitely more stable than the sigmoid.

ReLu activation appears to be the most effective among the three for this particular dataset and MLP architecture. The model converges rapidly, attains a high training accuracy and obtains low loss. Sigmoid Activation seems to be the least effective, with the model showing poor learning: high, stable loss and low unstable accuracy.

Question 2 (20)**. (Implementing CNN layers with NumPy)**

Question 3 (50)**. (Implementation and experimentation)**

3. Just a note here, my early stopping wasn't the greatest and it kept running even though the validation accuracy dropped in the notebooks. However the model, losses and accuracies were saved on early stopping :)
4. Before starting to make my own model, I did some research on some famous architectures to find some inspiration: MobileNets, EfficientNet, InceptionNet, ResidualNet.
I didn't want to go too crazy or weird, and simply wanted to improve VGG16 in terms price to performance. VGG16 has many parameters and takes a long time to train ! For batch size of 64, it takes 1h to train on a A100 and uses 17GB of VRAM which is surprising seeing how that it isn't a deep network. Therefore, I decided to take some inspiration from MobileNets and ResidualNets because it keeps a similar pipeline of VGG16 but allows us to go deeper. Furthermore, we would need to mess around with batch size, image sizes and other hyperparameters.
5. I set out to design an architecture where blocks would have residual skips, and in addition to that, there would be skips between blocks. This design choice allowed for a deeper architecture compared to VGG16, aiming to address the vanishing gradient problem inherent in deep networks. Initially, I made a straightforward architecture consisting of blocks with the following input-output pairs: [(64,128),(128,128),(128,256),(256,512),(512,1028)]. Each of these blocks was built with 4 layers, utilizing depthwise and pointwise convolutions. With the foundation in place, I went on to experiment on varying image sizes to see if changing input dimensions could enhance performance. It required tweaks to the initial convolutional layer, exploring different receptive fields via varied padding and stride configurations. Given that CIFAR10 images are inherently 32x32, it seemed logical to adjust the channel size from VGG16's typical 224x224 down to 96x96, especially when one of my goals was to be more economical in terms of computational resources. Having a balance between resource usage and larger input sizes was important, especially as I wanted for the architecture to delve deeper and therefore needed more data. While I wished to have explored this further, I was limited by the amount of computational units I had on Google Colab.

The model's performance showcased a promise, rivaling VGG16's outcomes but with a significantly reduced training duration: just 15 minutes compared to an hour. Moreover, it operated efficiently, consuming only 7GB of GPU VRAM. This efficient utilization showed that I would have headroom to augment the network's depth and capacity with a new goal of achieving a better performance than VGG16 now.

As I increased the depth of the network architecture, I observed a rise in GPU VRAM usage. However, what caught my attention was a slight drop in the model's performance, probably due to the fact that it now needed more epochs to converge. Furthermore, realized that my approach of using double skipping – both inside and outside the blocks – might be redundant things and reducing the model's overall efficiency. Considering the added depth, it made sense to also upscale the data intake to 128x128 images. So, I decided to reconfigure the design. Instead of just having a single block with residual skipping (along with an internal skip), I experimented with two blocks of skipping in a deeper model setup. Additionally, I varied the number of depthwise separable convolution layers within each block. The results were intriguing: the losses plummeted to very low values, but, surprisingly, the accuracy didn't see a proportional improvement.

I wanted to try a more modern architecture so I explored a simplified Inception module, even though the main goal remained parameter reduction and I knew just having a few branch on an inception model could rapidly increase the number of parameters. Throughout its process, I looked out for the pooling frequency, ensuring that spatial dimensions were retained for effective feature extraction. The Inception-inspired design proved great in just 11 minutes of training, although it also raised concerns of potential overfitting. The model save file was 1.3GB compared to the around 500MB marks of the other models !

Going a bit more in depth in my choices:
- For the pointwise convolution I chose a simple 1x1 convolution to allow the build of new features through the channel combination.
- In the depth wise block, when the input channels don't match output channels, I applied a 1x1 convolution to the input to align the dimensions. It mimics ResNet by allowing the addition of the original input to the output.
- I sometimes used SiLu to provide slightly better performance than the traditional ReLU

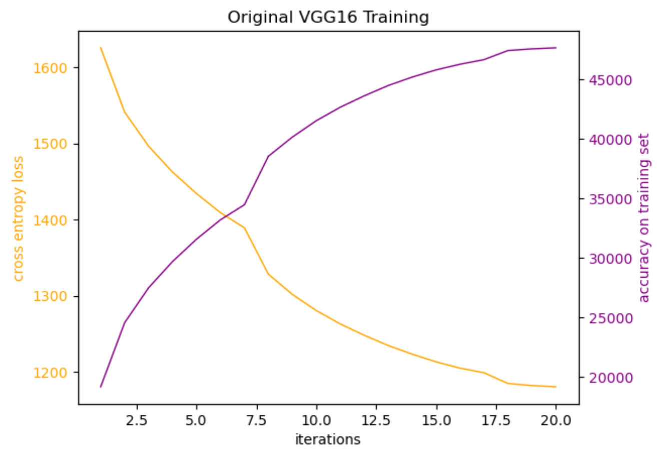
- Choosing a variable number of depthwise separable convolutions in a block made more sense, as not all layers need the same amount of complexity or refinement.

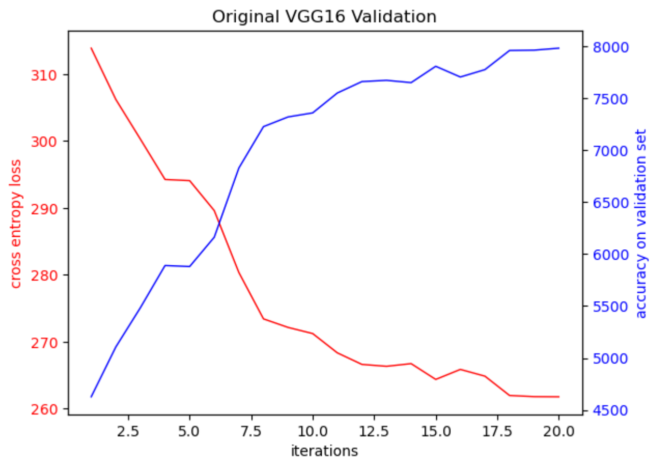
- in the overall architecture, we have a standard 3x3 convolution (for a good capturing of initial low level features), followed by batch normalization and a ReLU activation. The following poolings reduce the spatial dimensions, preparing the tensor for deeper processing.
- Multiple residual blocks are defined with varying input/output channel sizes. Which has a purpose of refining features.
- Pooling layers are spaced out equally (every 2 blocks) to reduce spatial dimensions and increase receptive field.
- Larger residual connections are introduced that skip over two blocks, making the network potentially easier to train by providing more direct paths for the gradients.

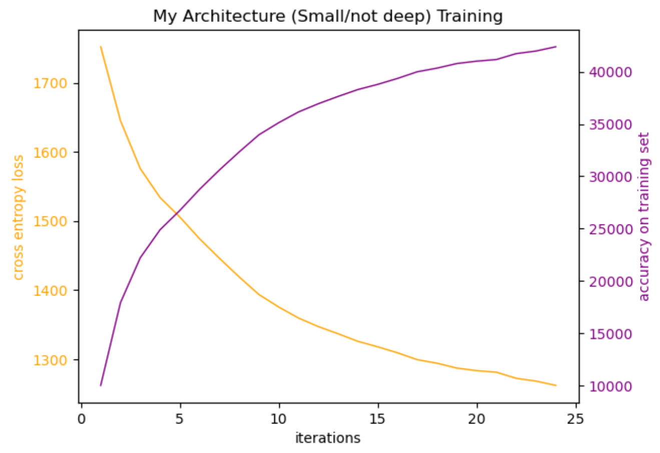
- Before the final fully connected layers, a global average pooling is used to reduce the spatial dimensions to 1x1, reducing the number of parameters all while focusing on the more defining features.
- MLP: A two layered fully connected network that further refines features before classification seemed more than enough comapred to VGG16's bigger MLP. I added dropout to reduce the risk of overfitting, even though it doesn't seem that necessary in this case.
- I chose the Kaiming and Xavier initilizations. Kaiming because I learned that it performs better with ReLu activation.

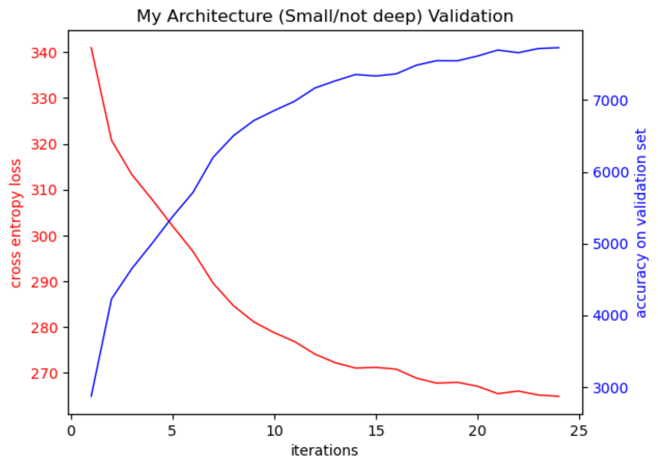
Overall, the nuances between depth, complexity, and performance, really showed me that there are many considerations to take in when creating an architecture and that it takes a lot of patience as well as computational ressources. It showed me that deep learning isn't just about powerful models ; it's also about smart decisions, sometimes simple simplicity and learning from trial and error. I would've really enjoyed exploring more but, as previously mentionned, was held back by a linited amounts of Computing Units Google Colab.

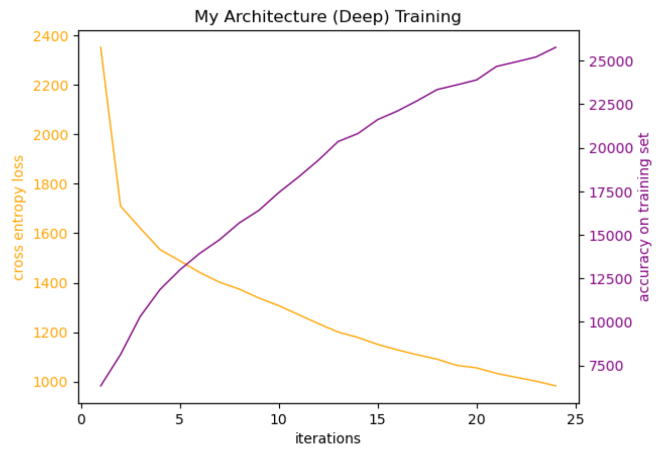
6. Here are the different architectures compared individually between their training loss/accuracy and validation loss/accuracy:

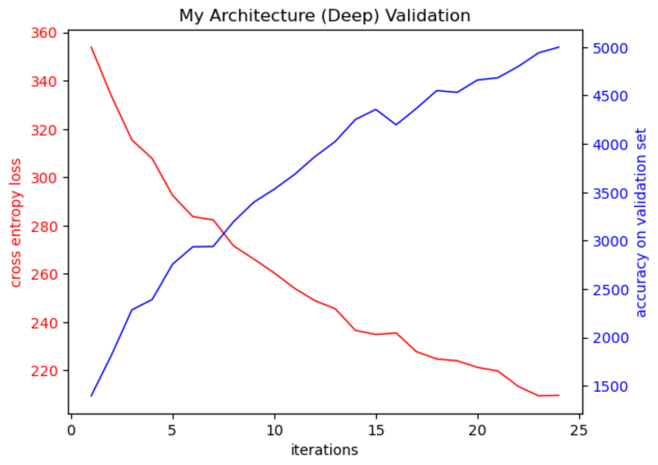


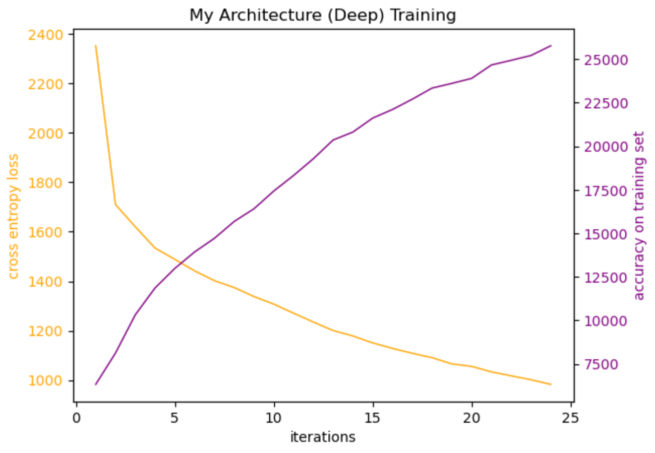


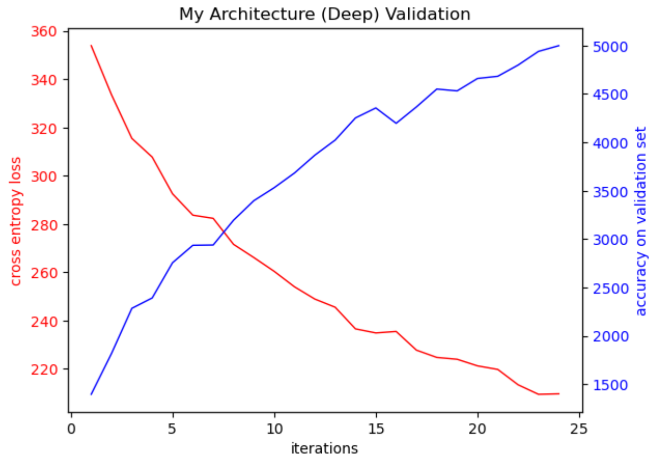


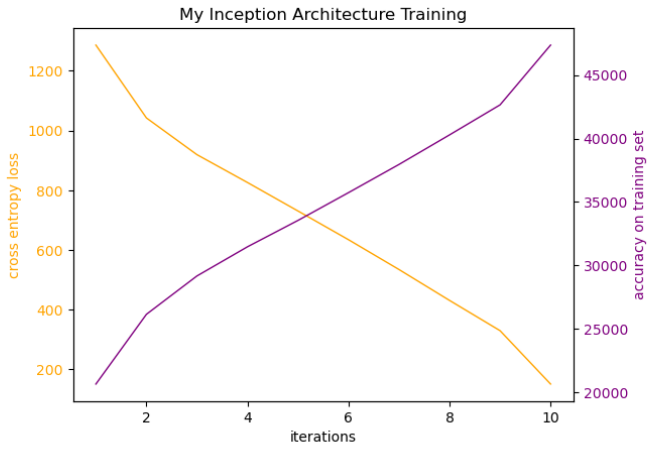


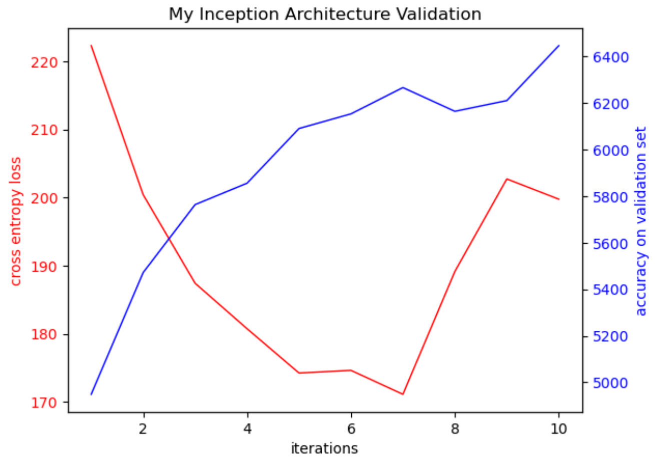




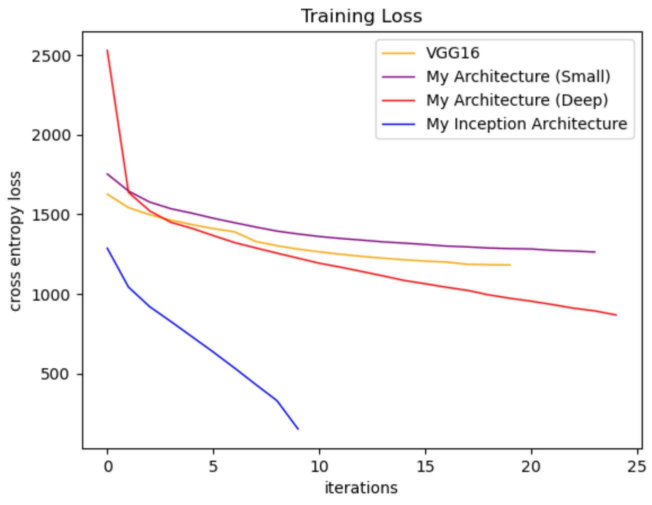

My Architecture (Deep) Training
cross entropy loss
accuracy on training set
iterations

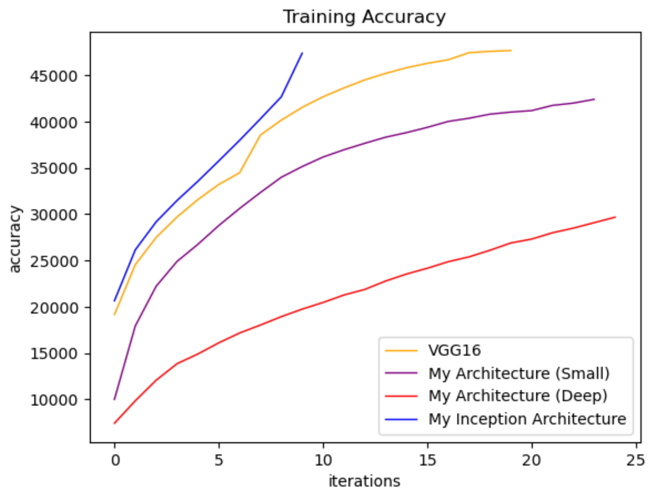

My Architecture (Deep) Validation
cross entropy loss
accuracy on validation set
iterations

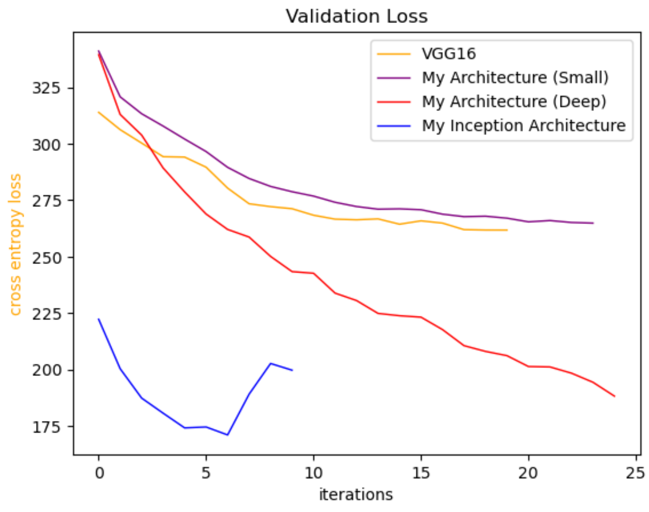

My Inception Architecture Training
cross entropy loss
accuracy on training set
iterations

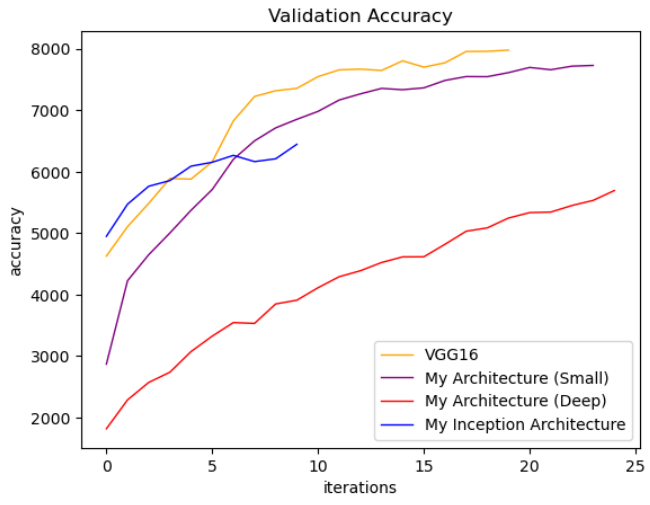

My Inception Architecture Validation
cross entropy loss
accuracy on validation set
iterations

And here they are compared to each other:









7. **What we can observe:**
   - Training loss:
     - For my deep architecture, we notice the loss starts high but rapidly catches up with VGG16 and performs slightly better after around 4 epochs. Reaches a low loss of 1000.
     - My smaller architecture is on par with VGG16, following it closely but at a slightly lower loss. They both stay around 1300-1500 and don't decrease much.
     - The inception architecture has an extremely rapid descent to 0 after only 10 epochs !
   - Training Accuracy:
     - The training accuracy is interesting here. My deeper architecture seems to need more time to converge and we see it doesn't seem to be plateauing anytime soon.
     - My smaller architecture follows once again VGG16 but finishes with a lower accuracy (>450000 vs around 42000).

  - The inception architecture has an rapid ascension, and reaches the same accuracy as VGG15 but in only 10 epochs instead of 25. Not to mention it took 11mins of training instead of 1h.
- Validation Loss:
  - Here we really see the bigger depth of my model shine. It reaches just under 200 in loss in 25 and seems to be able to keep going.
  - My smaller architecture follows once again VGG16 and they nearly have the same loss. They both plateau at around 20epochs, showing their limitations.
  - The inception starts extremely low and descends to 175 before worsening at 6. Past 6 epochs, the validation loss might be oscilating. Either hyperparameters could be improved as we are overshoot the optimal point or we are overfitting.
- Validation Accuracy:
  - Just like in the training accuracy, the deeper model would like to keep going and only reaches 5500 accuracy after 25epochs.
  - My smaller architecture follows once again VGG16 but here, the gap between them seems slightly smaller than in the training accuracy. Furthermore, VGG16 seems to be slightly more unstable than my architectures (deep and small, not inception).
  - Suprisingly, inception here performs much worse than VGG16 and my small model ! It finishes around 6500 after 10 epochs. There might be some overfitting ?

Analysis:

First, we need to remember a few things: VGG16 took 1 hour to train, whereas my architecture that mixes depthwise separable convolution with residual block architectures took only 13 minutes for the smaller depth and 16 minutes for the deeper model. Furthermore, the difference in parameters between my smaller architecture and VGG16 is substantial. I don't have the exact number for my smaller architecture, but in terms of VRAM, we're looking at 17GB versus just under 7GB !

Due to the residual connections in my deeper architecture, we could potentially train it for more epochs. The trade-off is significant as we might achieve higher accuracy and lower loss in potentially the same training time as VGG16, using roughly 14GB of VRAM. However, the downside is that the intricate design of the residual connections combined with depthwise convolutions could make fine-tuning more challenging.

On the one hand, Inception might perform better with a larger dataset and more fine-tuning (that wasn't my primary focus here). On the other hand, its lack of depth might be the reason it began overshooting. We'd need more tests to confirm this, but its strength is evident from the rapid decreases in loss and swift gains in accuracy. Given the modest training time of 11 minutes and significant VRAM consumption, it's a trade-off to consider.

In conclusion, each architecture has its strengths and weaknesses. However, VGG16 seems rather cumbersome due to its long training time and extensive VRAM usage. While it does produce commendable results (so do the others), these outcomes can be achieved more economically through more efficient means, as demonstrated by my smaller architecture or inception.

8. Images:

Filter 1
Filter 2
Filter 3
Filter 4
Filter 5

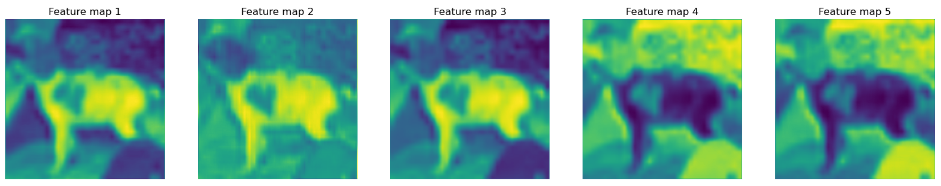
(a) Filters of the first convolutional layer of VGG16

Filter 1
Filter 2
Filter 3
Filter 4
Filter 5

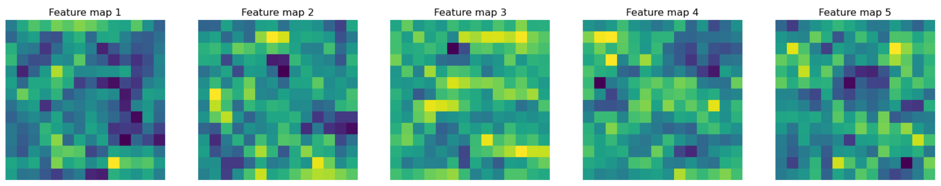
(b) Filter of the last convolutional layer of VGG16

They seem the same, and seem to not have much information but they differ a lot in depth. Here the depth can't be very well represented for the last layer (unless we make a tiny 3x512) so the depth is averaged. The first filters are of sizes (3, 3, 3), we have 64 of them and the last filters are of sizes (512, 3, 3), we have 512 of them
These filters are low level filters, responsible for detecting features in images such as edges or simple shapes. they will be more reactive to sharp changes in color or intensity in the input image.
However, the depth increase is because of the growing complexity of features being captured throughout the architecture. While early layers detect simple features directly from the image, deeper layers combine these simple features in various ways to detect more complex patterns. The large depth of the last layer shows how the model processes and combines a set of features extracted from the input image running through all the previous layers.

9. Images:



(c) Filters of the first convolutional layer of vis

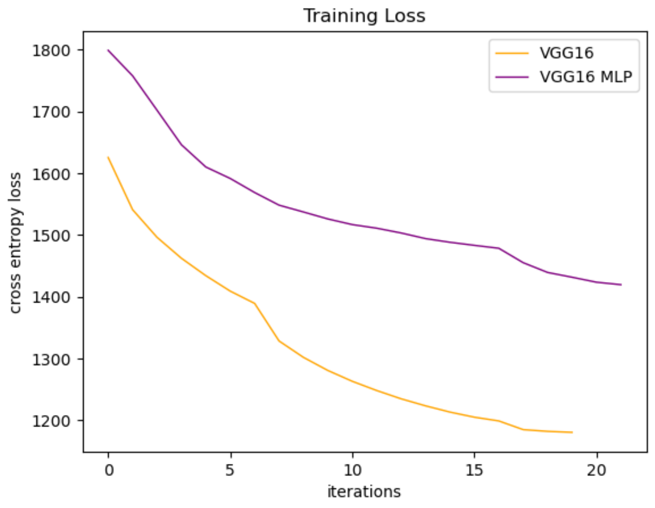


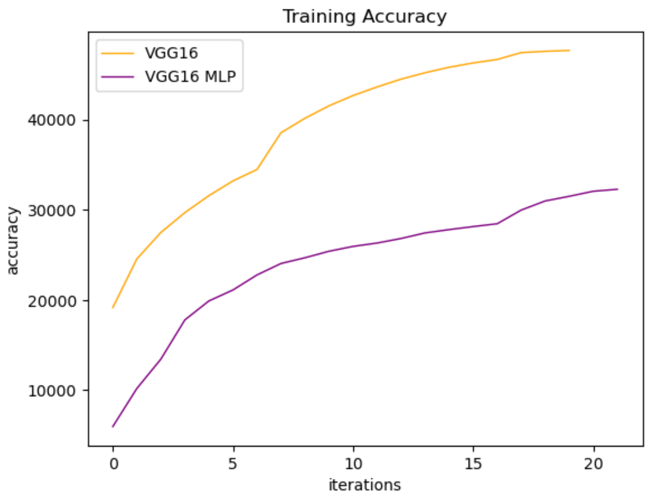
(d) Filters of the last convolutional layer of vis

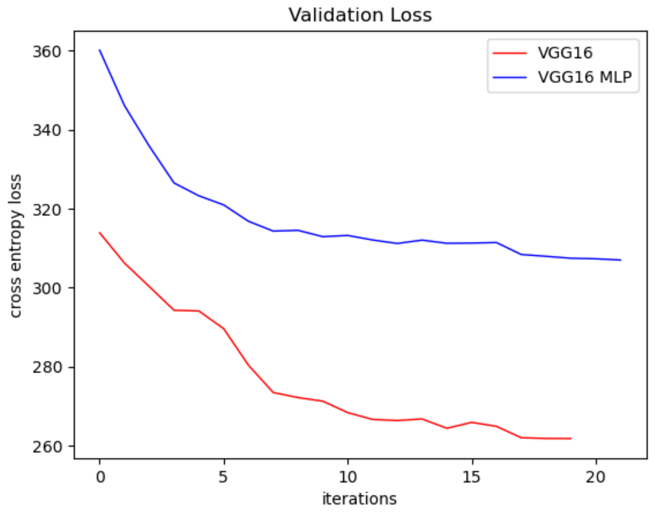
The images of the first layer, we observe emphasis on broad outlines and general structures. The details are clear, we notice the dog (that has a heart on his fur actually, very cute) and we can still distinguish original input image, there are just different colour depths.

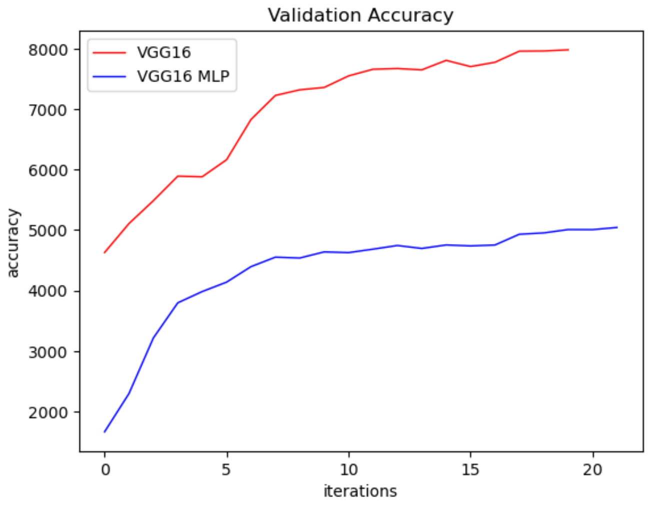
The feature maps from the last layer capture higher-level, they are more abstract and less visually interpretable from the original image. Instead of direct patterns like edges or colors, they represent a complex combinations of features, not necessarily noticeable here but in their depth.

10. Comparison graphs:









	Model 1	**Model 2**
Loss mean	317.81	277.21
Loss std	13.25	15.98
Accuracy mean	4324.41	6978.95
Accuracy std	861.68	1027.87
Number of parameters	176 784 778	134 309 962

Table 1 – Comparison of Model 1 and Model 2.

VGG16 has better performance in terms of both training accuracy and training loss, and also in terms of validation accuracy and calidation loss.
VGG16 is much smaller in terms of the number of parameters, making the MLP any bigger would be way too costly.

These results are to be expected as CNNs like VGG16 take advantage of the spatial hierarchies in the data. Because a filter is used across the entire image, it allows these architectures to share their weight through the convolution operation. Furthermore, while MLPs can be made deep like VGG16, the sheer number of parameters can grow exponentially, I had to try many different combinations of layers to just be able to run the training without crashing everything.

11. Submit a link to your Google Colab notebook in your PDF report (at the top) as well, make sure to make it publicly viewable !