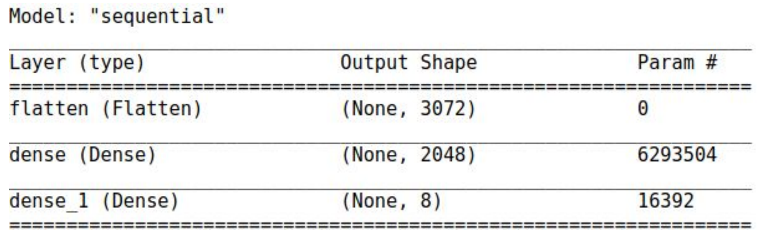
Master in
Computer Vision
Barcelona

M3 : Week 3

Group 4:
Aditya Sangram Singh Rana
Òscar Lorente Corominas,
Ian Riera Smolinska

Understanding Network Topology

The first part (task 1) of this project focus on understanding the baseline model and try to modify it to see how the different parameters and layers affect the accuracy.
We use the following model as a baseline.

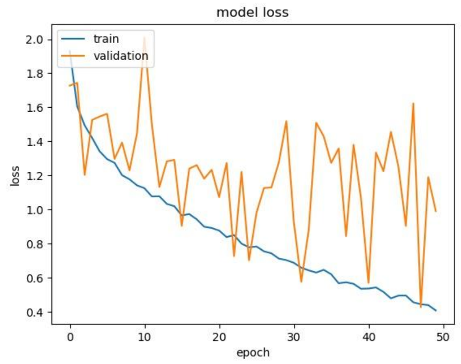


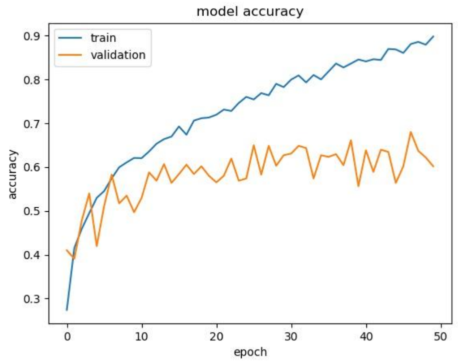
```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 3072)              0
_________________________________________________________________
dense (Dense)                (None, 2048)              6293504
_________________________________________________________________
dense_1 (Dense)              (None, 8)                 16392
=================================================================
```

We can observe a notable downgrade on accuracy between training and validation. This means that our model is overfitted to the training data.

Training longer than 12 epochs does not improve the validation accuracy.

We compute the model's accuracy and loss for each epoch.





	Train	Validation
Accuracy	0.90	0.61
Loss	0.41	0.99

Network Topology: Learning Rate

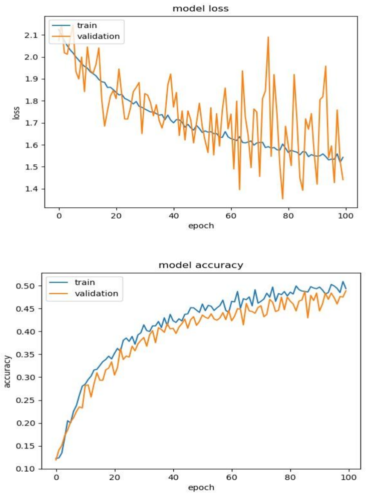
To try to improve the network and reduce the overfitting, we first changed the learning rate:

Learning rate	Train accuracy	Validation accuracy
1e-4	0.45	0.45
1e-3	0.63	0.55
1e-2	0.91	0.61
1e-1	0.78	0.57

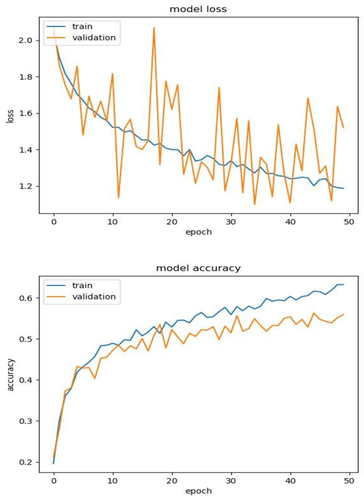
With a lower learning rate (lr), the curves are smoother, but it converges too slow (notice that we're using 100 epochs and the higher accuracy is much lower than the other ones with only 50 epochs).

Even if the best results are obtained with lr=1e-2, we have still a lot of overfitting, so we'll try to improve it with other methods.

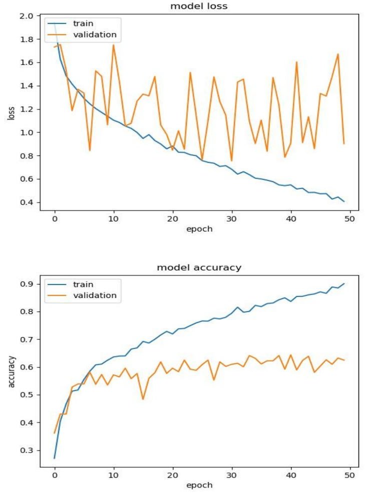
1e-4



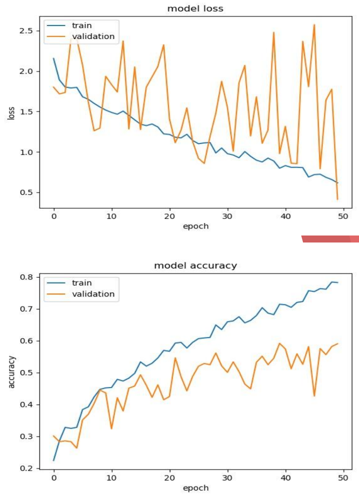
1e-3



1e-2



1e-1



Tricks: Cyclical Learning Rates

As we already studied the importance of the learning rate by changing it manually, we have also used some SOTA tricks to automatically find the best learning rate, momentum and optimizer using:

1. **The One Cycle Policy**[1] - allows us to train our networks at much higher learning rates, and thus they converge in lower number of epochs.
2. **Learning Rate Finder**[2] - allows us to find the best learning rate for our models.

In the following experiments, the number of epochs is much lower (~10), as we use these tricks to automatically find the best hyperparameters (learning rate, momentum...), so we can train our model faster.

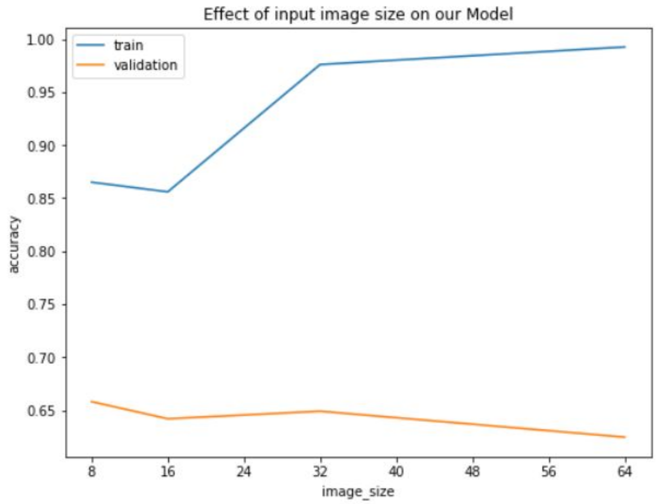
References

1. [1803.09820] A disciplined approach to neural network hyper-parameters: Part 1 -- learning rate, batch size, momentum, and weight decay
2. [1506.01186] Cyclical Learning Rates for Training Neural Networks

Network topology: Image Size

Let's change the size of the input images:

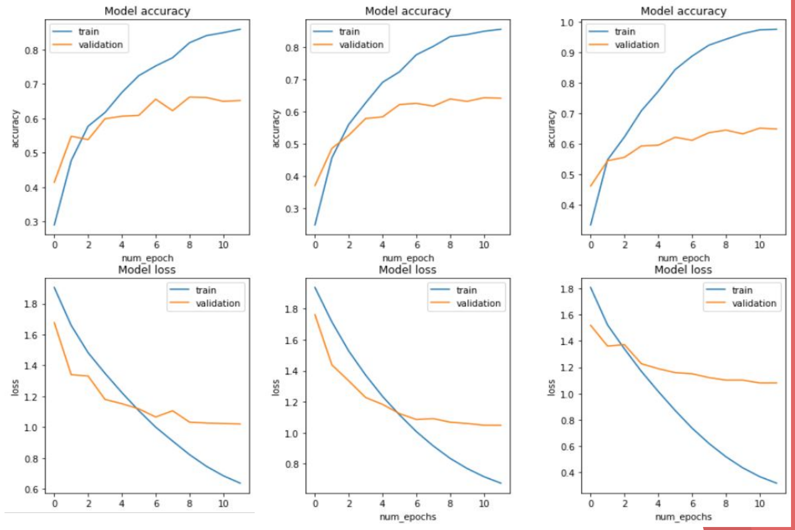
Image size	Train accuracy	Validation accuracy
8x8	0.87	0.64
16x16	0.85	0.62
32x32	0.96	0.63



8x8

16x16

32x32



We don't observe any significant improvement in the validation results by changing the input image size. With bigger image sizes (32x32) we have even more overfitting, as the difference between the train and validation accuracy is larger.

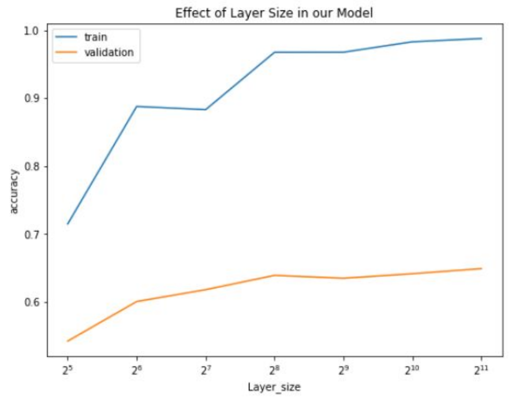
Furthermore, using images of size 64x64 (or even more) the model has to learn **a lot** of parameters, so we had to redefine some layers to make it work (and not run out of memory). For this reason, it's not worth it to try with higher values than 32x32.

Network Topology: Layer Sizes

Let’s keep the depth of the network constant and only change the size of the hidden layer.

We try sizes = [32, 64, 128, 256, 512, 1024, 2048] and stop there as we see no further improvement regarding the overfitting problem.

Layer size	Train accuracy	Validation accuracy
32	0.71	0.56
64	0.88	0.60
128	0.87	0.62
256	0.96	0.64
512	0.96	0.63
1024	0.98	0.64
2048	0.98	0.65

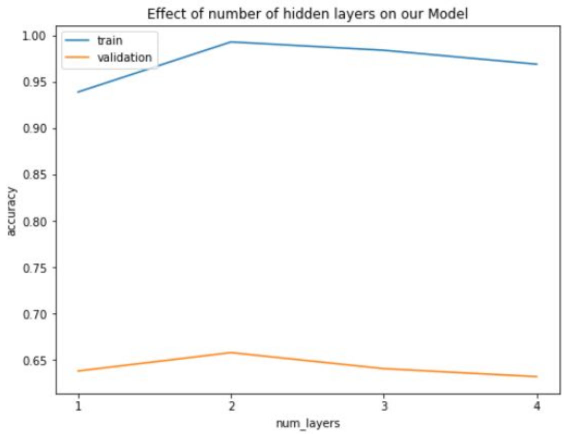


- ➔ Using bigger hidden layers lead to better results, which is expected, as the model’s representational capacity increases.
- ➔ However, all the models compared here still show high overfitting to the training dataset.

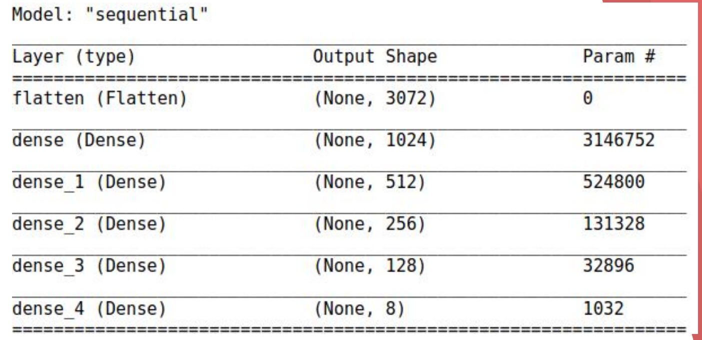
Network Topology: Increasing Depth

Let's add more hidden layers to our network. We start with a first dense layer (1024), and continue adding layers of half the size of the previous layer to test different models. All models are trained until they show no more improvement.

For example
Layers 1 = [1024] (single hidden layer)
Layers 2 = [1024, 512]
Layers 3 = [1024, 512, 256]
...



Example model with four hidden layers



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 3072)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              3146752
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
dense_2 (Dense)              (None, 256)               131328
_________________________________________________________________
dense_3 (Dense)              (None, 128)               32896
_________________________________________________________________
dense_4 (Dense)              (None, 8)                 1032
=================================================================
```

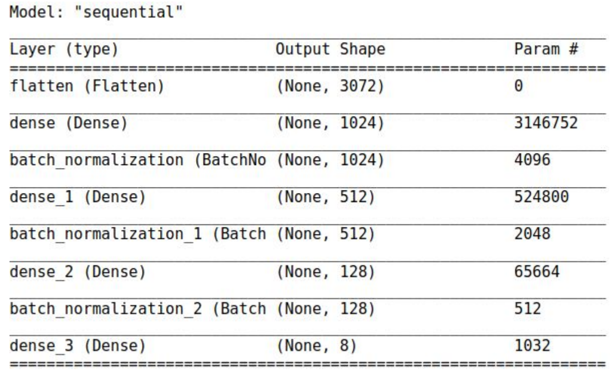
→ For this particular dataset, simply adding more hidden layers does not offer any major advantage.

→ We observe no major difference with respect to overfitting,

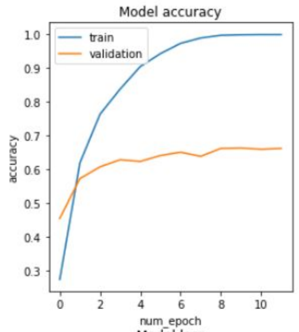
Network Topology: Adding BatchNorm

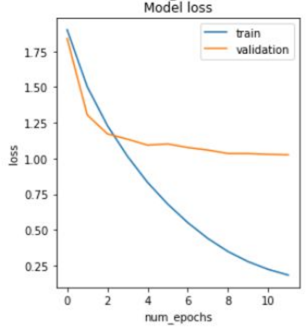
One possible solution to our overfitting problem could be adding normalization (e.g. BatchNorm), which helps in the training of deeper models.

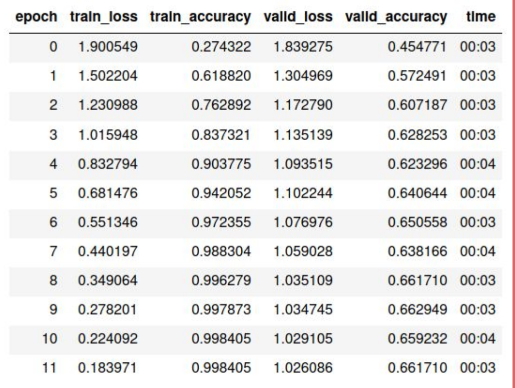
Let's try adding BatchNormalization to the following configuration:



```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 3072)              0
_________________________________________________________________
dense (Dense)                (None, 1024)              3146752
_________________________________________________________________
batch_normalization (BatchNo (None, 1024)              4096
_________________________________________________________________
dense_1 (Dense)              (None, 512)               524800
_________________________________________________________________
batch_normalization_1 (Batch (None, 512)               2048
_________________________________________________________________
dense_2 (Dense)              (None, 128)               65664
_________________________________________________________________
batch_normalization_2 (Batch (None, 128)               512
_________________________________________________________________
dense_3 (Dense)              (None, 8)                 1032
=================================================================
```







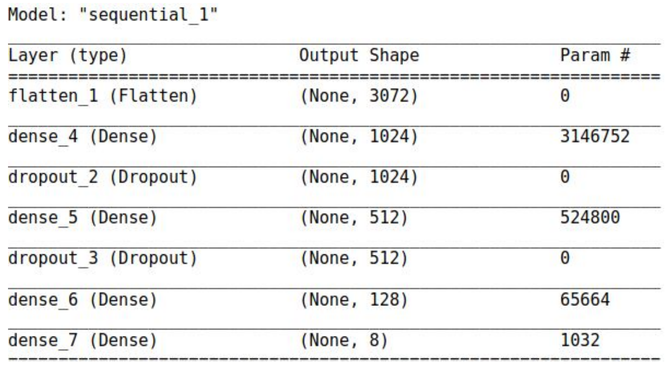
epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	1.900549	0.274322	1.839275	0.454771	00:03
1	1.502204	0.618820	1.304969	0.572491	00:03
2	1.230988	0.762892	1.172790	0.607187	00:03
3	1.015948	0.837321	1.135139	0.628253	00:03
4	0.832794	0.903775	1.093515	0.623296	00:04
5	0.681476	0.942052	1.102244	0.640644	00:04
6	0.551346	0.972355	1.076976	0.650558	00:03
7	0.440197	0.988304	1.059028	0.638166	00:04
8	0.349064	0.996279	1.035109	0.661710	00:03
9	0.278201	0.997873	1.034745	0.662949	00:03
10	0.224092	0.998405	1.029105	0.659232	00:04
11	0.183971	0.998405	1.026086	0.661710	00:03

→ BatchNormalization slightly improves our results, but it's not enough to deal with the overfitting.

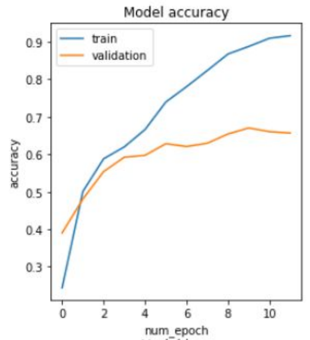
Network Topology: Dropout

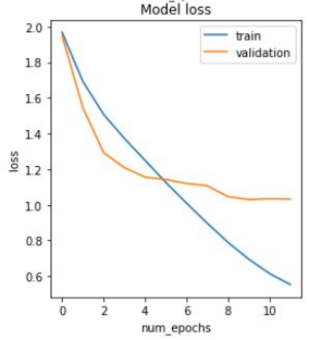
Another possible solution could be using some regularization, for example with dropout.

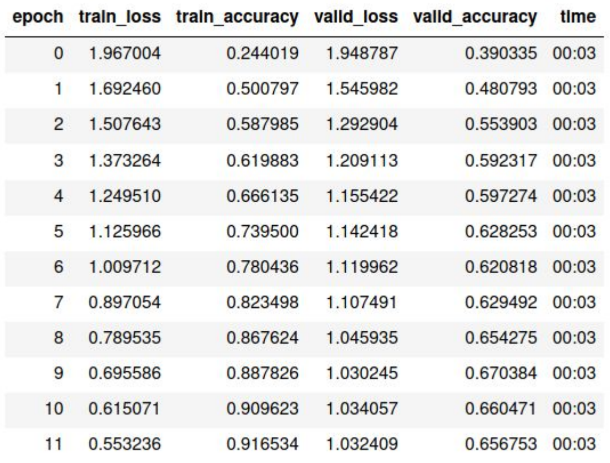
Let's try adding dropout to the following configuration:



```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 3072)              0
_________________________________________________________________
dense_4 (Dense)              (None, 1024)              3146752
_________________________________________________________________
dropout_2 (Dropout)          (None, 1024)              0
_________________________________________________________________
dense_5 (Dense)              (None, 512)               524800
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_6 (Dense)              (None, 128)               65664
_________________________________________________________________
dense_7 (Dense)              (None, 8)                 1032
=================================================================
```







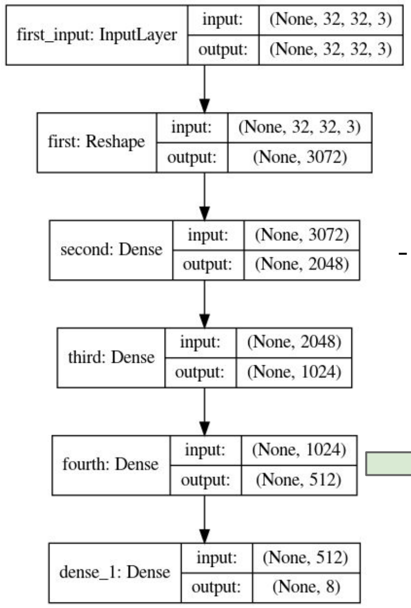
epoch	train_loss	train_accuracy	valid_loss	valid_accuracy	time
0	1.967004	0.244019	1.948787	0.390335	00:03
1	1.692460	0.500797	1.545982	0.480793	00:03
2	1.507643	0.587985	1.292904	0.553903	00:03
3	1.373264	0.619883	1.209113	0.592317	00:03
4	1.249510	0.666135	1.155422	0.597274	00:03
5	1.125966	0.739500	1.142418	0.628253	00:03
6	1.009712	0.780436	1.119962	0.620818	00:03
7	0.897054	0.823498	1.107491	0.629492	00:03
8	0.789535	0.867624	1.045935	0.654275	00:03
9	0.695586	0.887826	1.030245	0.670384	00:03
10	0.615071	0.909623	1.034057	0.660471	00:03
11	0.553236	0.916534	1.032409	0.656753	00:03

➔ Dropout slightly improves the results, but again, it's not enough to deal with the overfitting.

Extracting Deep Features + SVM

The second part (tasks 2 and 4) of this project focus on extracting deep features from a hidden layer given an input image, and then train an SVM classifier. First, let's present the methodology using a configuration with three hidden layers:

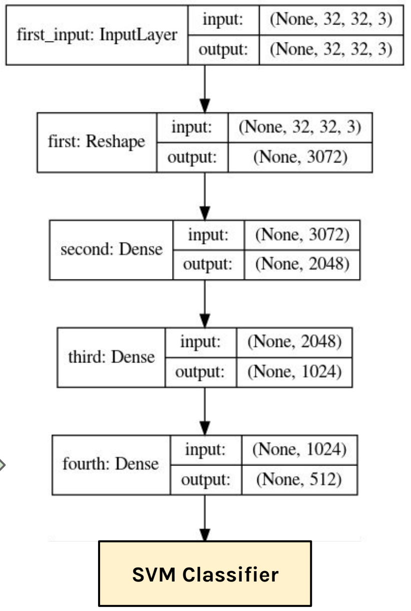
End-to-end model


first_input: InputLayer | input: (None, 32, 32, 3) | output: (None, 32, 32, 3)
first: Reshape | input: (None, 32, 32, 3) | output: (None, 3072)
second: Dense | input: (None, 3072) | output: (None, 2048)
third: Dense | input: (None, 2048) | output: (None, 1024)
fourth: Dense | input: (None, 1024) | output: (None, 512)
dense_1: Dense | input: (None, 512) | output: (None, 8)

Note: We can also extract features from previous hidden layers, but they will be less abstract/general compared to the deeper hidden layer, so the results are expected to be worse

We extract the learnt (deep) features of the model from the last hidden layer (just before the softmax classification), and use them to train a SVM

SVM approach

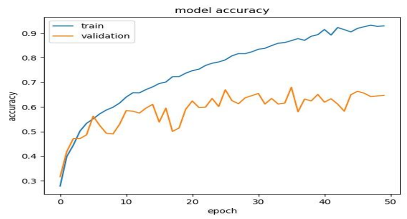

first_input: InputLayer | input: (None, 32, 32, 3) | output: (None, 32, 32, 3)
first: Reshape | input: (None, 32, 32, 3) | output: (None, 3072)
second: Dense | input: (None, 3072) | output: (None, 2048)
third: Dense | input: (None, 2048) | output: (None, 1024)
fourth: Dense | input: (None, 1024) | output: (None, 512)
SVM Classifier

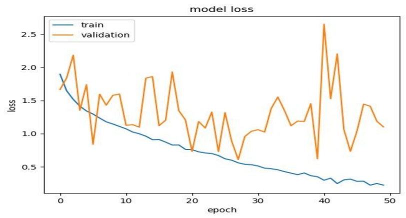
Deep Features with an SVM Classifier

Let's study the test accuracy obtained with the SVM classifier, and compare it to the end-to-end approach. We'll extract deep features from different hidden layers to see if our first assumption (that deeper layers works better) is correct. To do so, we'll use the configuration shown in the previous slide.

End-to-end

Train accuracy	Test accuracy
0.93	0.65





SVM

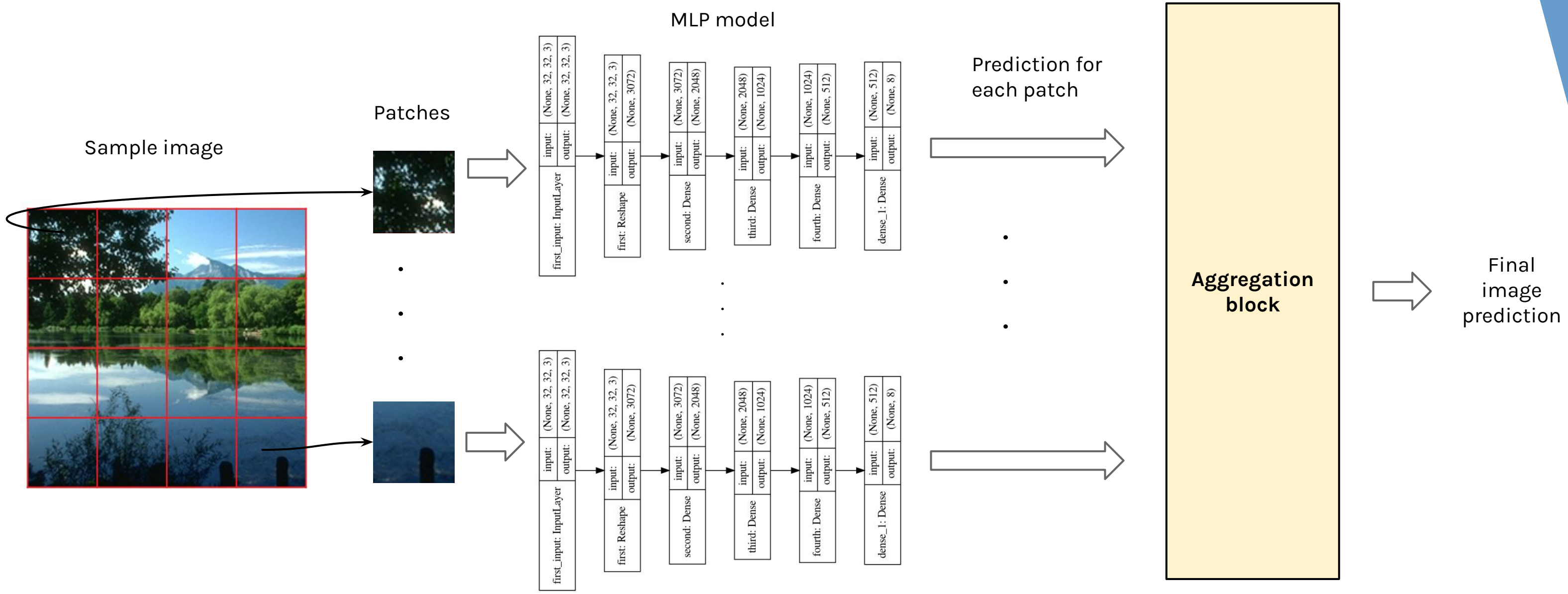
Hidden layer	Test accuracy
Second (2048)	0.41
Third (1024)	0.43
Fourth (512)	0.44

➔ As expected, end-to-end performs better than the SVM classifier.

➔ The deep features that work *better* for the SVM approach are the ones extracted from the last (deeper) hidden layer, as the features are more abstract/general.

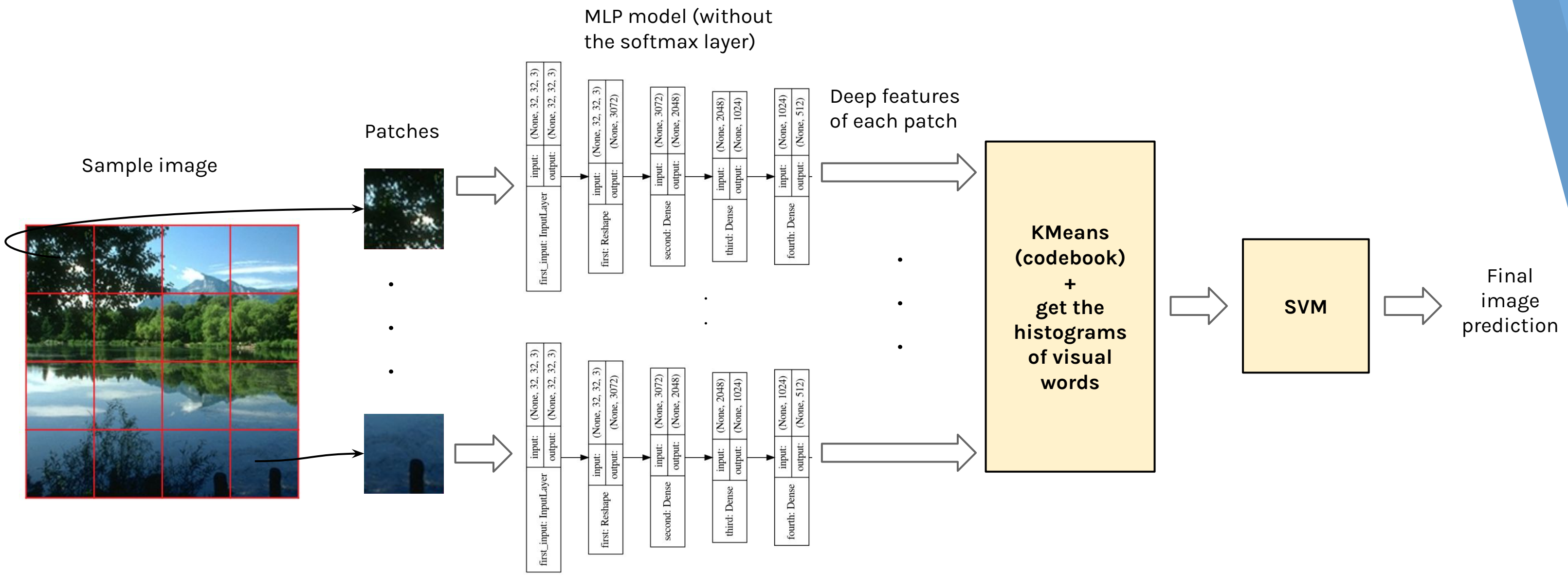
Using MLP Deep Features as a Dense Descriptor

The final part (tasks 3 and 5) of this project focus on dividing the image into small patches, extract the prediction for each patch and aggregate the final prediction. First, let's present the methodology with a simple example:



Using MLP Deep Features as a Dense Descriptor + Bag of Visual Words

As an alternative, the output patches can be used to create dense descriptors for a Bag of Visual Words approach. To do so, we extract the deep features from the last hidden layer (previous to softmax classification) for each image patch, and we concatenate these features to create a feature vector for each image. Then, we use KMeans to create a codebook and we train an SVM classifier with the histograms of visual words:



MLP Dense Descriptors and BoW

Let’s try the end-to-end approach with different patch sizes, to see if this parameter has an impact on the results:

Patch size	Test accuracy
8x8	0.56
16x16	0.65
32x32	0.72
64x64	0.66

- ➔ We can observe that changing the patch size affects notably the performance of the classifier.
- ➔ Too small patch sizes (e.g. 8x8) leads to a really bad classification, as the *descriptors* are too specific.
- ➔ We obtain the best accuracy with a patch size of 32x32.

On the other hand, we also tuned the codebook size of the BoW approach:

BoW: codebook size	Test accuracy
64	0.60
128	0.65
256	0.70
512	0.68
1024	0.66

- ➔ With small codebook sizes (e.g. 64), the features are too general, so they are not representative enough to perform classification properly.
- ➔ With large codebook sizes (e.g. 1024), the visual words are too specific, which leads to a decrease in the performance.
- ➔ The best results are obtained with a codebook size of 128.
- ➔ In this case, the end-to-end approach (with a patch size of 32x32) performs slightly better than the BoVW classifier.

Conclusions

- A multilayer perceptron (MLP) is too simple for a classification problem as the one that we're facing here. We can try changing some parameters as the number of layers (to increase depth), the number of neurons per layer, adding normalization or regularization, and so on and so forth. However, the potential of the system is limited to the fact that we're using a simple MLP, so the results won't improve at all.

- Extracting deep features and using them to train an SVM classifier is not a good alternative, as the results are not satisfactory. In that case, using features from deeper hidden layers provides *better* (but still bad) results, as the features are more abstract/general compared to the other layers.

- Another approach is to divide each image in patches and extract deep features from each of them. In that case, we're using the output of MLP's last hidden layer as dense descriptors, which provides slightly better results. Using these dense descriptors to create a Bag of Visual Words (BoVW) provides similar but slightly worse results.

- After doing a lot of experiments, we can conclude that even if MLP is not the most suitable option here due to its simplicity, it provides an accuracy of ~72% in some cases. We're still far from the desired results, but we see a lot of potential in neural networks compared to the BoVW approach (which is more limited). For this reason, we want to keep trying things to see if we can surpass the BoVW + SVM system implemented on previous weeks.