Deep learning workshop - Assignment 1

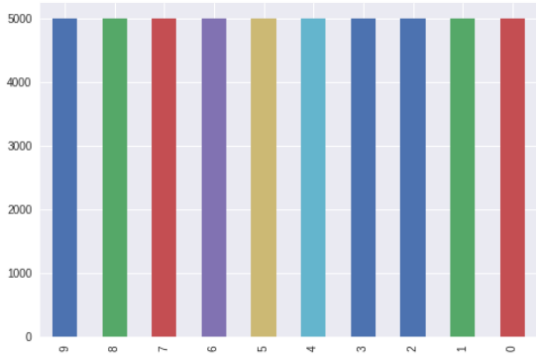
Doron Laadan.

Tal ben-senior.

We chose to work on the cifar-10 dataset after exploring the other options. As far as we could tell it was the most organize dataset and had a lot of previous works done on it.

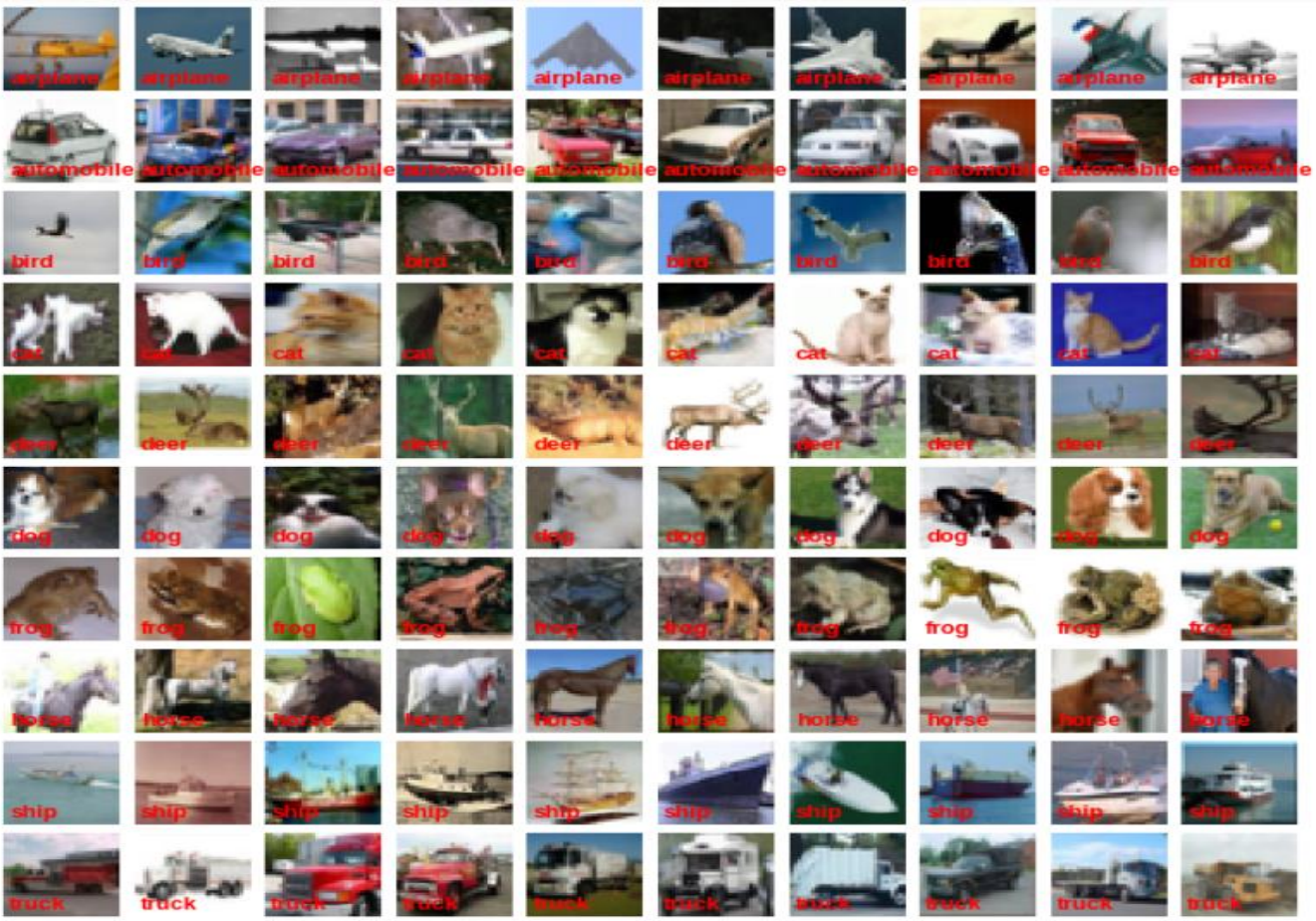
EDA:

a. The data set size is 60000 samples. 50000 samples in the training set and 10000 more on the test set. On the train set each class out of the ten classes has 5000 samples so we can say it is balanced.
b. Each sample in the data is 32X32X3 image, meaning each sample is a matrix of size 32X32 and with 3 channels (RGB). As it is well known and used dataset integrated into keras no pre-process is requires and data is ready to use when loaded.
we can use data augmentations as can be seen in the notebook. It is used as way to improve the model accuracy and generalize it. If we look at samples from the dataset, we can understand that not all forms of augmentations would be good. For example, vertical split might cause problem. Airplane usually surrounded by blue sky in the top while ship is surrounded with blue sea at the bottom, flipping the images might cause confusion while we train the model.
c. Yes. The data is balance, as mention before from each class there are 5000 examples.



d. As we mentioned before, cifar-10 is a widely used dataset (some would say overly). As such there are a lot of different studies, blogs and kernels working on it. The best results we found which we can consider benchmarks have 95%-98% accuracy on the test set.

e. Some examples of the different labels in the dataset



Basic ANN:

a. Our validation strategy is pretty straight forward, we used for the most part the simple validation split. For our first 2 versions of the model (the ones without data augmentation) we used the build in validation split hyper parameter in keras.fit function. We used a 80-20 split as it is common. Later when we used data augmentations, we split the data ourselves again using a 80-20 split. Because the data is well balanced and divided to test and train using those form of simple validation split were good enough for us and produce satisfying results later.

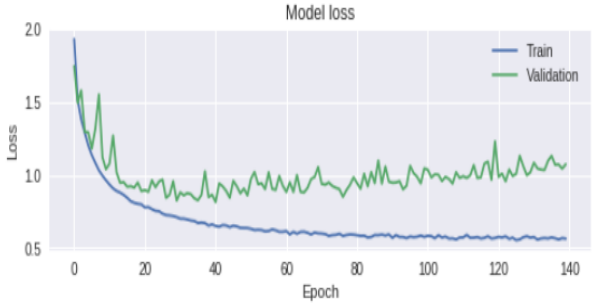
b. First run: for our first run we build a simple model based on the model shown in class with a little change. We then run the model for 140 epochs with no pre processing or change to data of any kind.

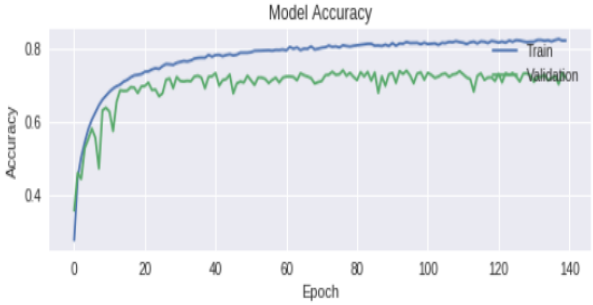
the model summary is:

Layer (type)	Output Shape	Param #
conv2d_1 (Conv2D)	(None, 30, 30, 16)	448
conv2d_2 (Conv2D)	(None, 28, 28, 16)	2320
max_pooling2d_1 (MaxPooling2	(None, 14, 14, 16)	0
batch_normalization_1 (Batch	(None, 14, 14, 16)	64
conv2d_3 (Conv2D)	(None, 14, 14, 32)	4640
conv2d_4 (Conv2D)	(None, 14, 14, 32)	9248
max_pooling2d_2 (MaxPooling2	(None, 7, 7, 32)	0
batch_normalization_2 (Batch	(None, 7, 7, 32)	128
dropout_1 (Dropout)	(None, 7, 7, 32)	0
conv2d_5 (Conv2D)	(None, 5, 5, 64)	18496
conv2d_6 (Conv2D)	(None, 3, 3, 64)	36928
dropout_2 (Dropout)	(None, 3, 3, 64)	0
max_pooling2d_3 (MaxPooling2	(None, 1, 1, 64)	0
flatten_1 (Flatten)	(None, 64)	0
dense_1 (Dense)	(None, 10)	650

Total params: 72,922
Trainable params: 72,826
Non-trainable params: 96

the training process:

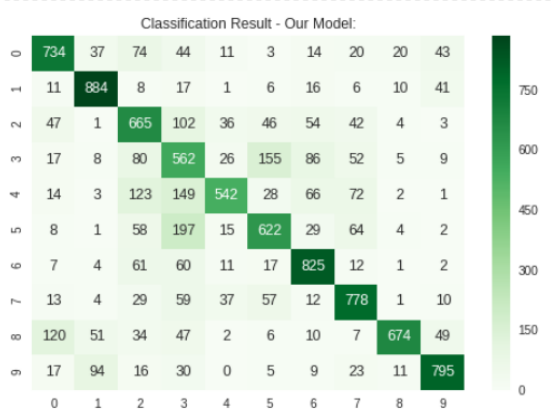




Training set accuracy: 0.82145
Validation set accuracy: 0.7197
10000/10000 [==============================] - 1s 118us/step
Test set accuracy: 0.7081

Model accuracy on test set is: 70.81%

	precision	recall	f1-score	support
0	0.74	0.73	0.74	1000
1	0.81	0.88	0.85	1000
2	0.58	0.67	0.62	1000
3	0.44	0.56	0.50	1000
4	0.80	0.54	0.64	1000
5	0.66	0.62	0.64	1000
6	0.74	0.82	0.78	1000
7	0.72	0.78	0.75	1000
8	0.92	0.67	0.78	1000
9	0.83	0.80	0.81	1000
avg / total	0.72	0.71	0.71	10000



results:

82% on train, 71% on validation and 70% on test.

we can say several things

1. Since the test accuracy is worse than the train accuracy, we suspect we overfitted the model.
2. The model was least successful in identifying classes 3-5 and had the most success with class 2
3. The validation accuracy and the test accuracy are close, so we can confirm our validation process works.

c. Reasons that might cause misclassification:

1. Some photos of cats and dogs looks alike. Even for us it took a while to understand that a picture of a cat was really a cat and not a dog. This problem could be fixed by using augmentations and create more samples of the classes and thus making the model to learn the differences better.
2. We assume the featured learned by the network is poor due to the optimization method that have been chosen, the learning rate is too high to converge to a global minimum – which later we reduced at particular epochs checkpoints by the 'scheduler' function. (Later we realize that may using ReduceLROnPlateau callback would achieve better results)
3. As we mentioned above, labels 3-5 which are cat, deer and dog, may confuse the network because of their features similarity which may confuse human eye as well. Another interesting result is the network misclassify ship for an airplane, a reasonable assumption is the image background (sky – sea colors), though we could not explain why this attribute not works vise versa (mistaken airplane for a ship).

What can be done to improve results?

In order to improve the accuracy and reduce overfitting we read several blogs online and looked at several Kaggle kernals. We came up with these methods

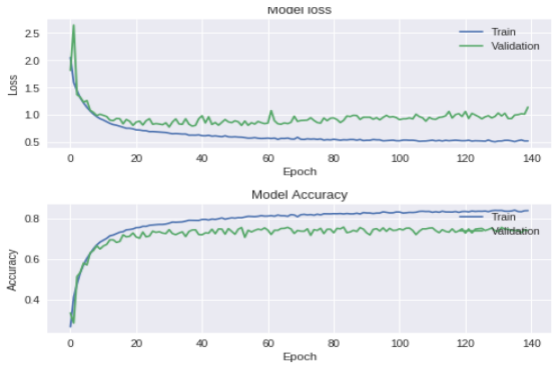
1. Data augmentation.
2. Changing optimizers.
3. More/less epochs.
4. Change layers.
5. Normalize the data

Before the project ended, we pretty much tried all of those methods mentioned above to improve results, some with luck and some without. First, we prioritize the list as follow

5->2->1->3->4

Second run: for our second run we used the same simple model based. We then run the model for 140 epochs after normalizing the data. The normalization was done using the well known method of standard score.

The training process:




```
Training set accuracy: 0.8384
Validation set accuracy: 0.7405
10000/10000 [==============================] - 1s 125us/step
Test set accuracy: 0.7326
--------------------------------------------------------------------
Model accuracy on test set is: 73.26%
```

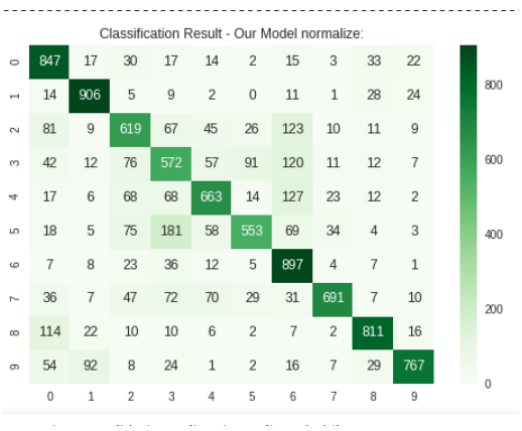
the results:

83% on train, 74% on val and 73% on test.

As we can see we improved the results a bit. But we might still have a problem of overfitting.

More over the model is still misclassify the same labels as before.

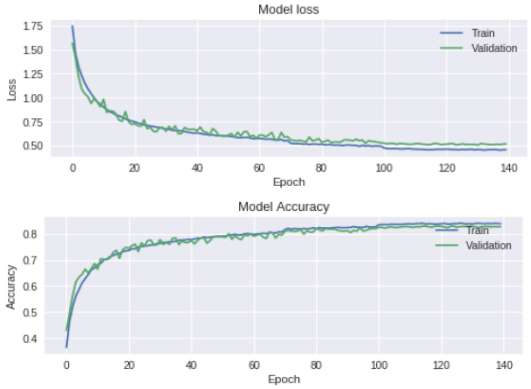
	precision	recall	f1-score	support
0	0.69	0.85	0.76	1000
1	0.84	0.91	0.87	1000
2	0.64	0.62	0.63	1000
3	0.54	0.57	0.56	1000
4	0.71	0.66	0.69	1000
5	0.76	0.55	0.64	1000
6	0.63	0.90	0.74	1000
7	0.88	0.69	0.77	1000
8	0.85	0.81	0.83	1000
9	0.89	0.77	0.82	1000
avg / total	0.74	0.73	0.73	10000



Third run: for our third run we used the same simple model based. We then run the model for 140 epochs after using normalization as before and data augmentation. We split the augmented data ourselves and run the model. The normalization was done using the well-known method of standard score. And the augmentation parameters are:

```
datagen = ImageDataGenerator(horizontal_flip=True,
                             width_shift_range=0.125,
                             height_shift_range=0.125,
                             fill_mode='constant',
                             cval=0.)
```

The training process:

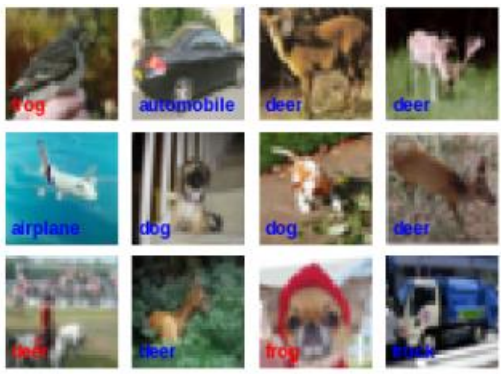



```
Training set accuracy: 0.8380216538192207
Validation set accuracy: 0.8267212978392866
10000/10000 [==============================] - 1s 130us/step
Test set accuracy: 0.8216
-----------------------------------------------------------------
Model accuracy on test set is: 82.16%
```

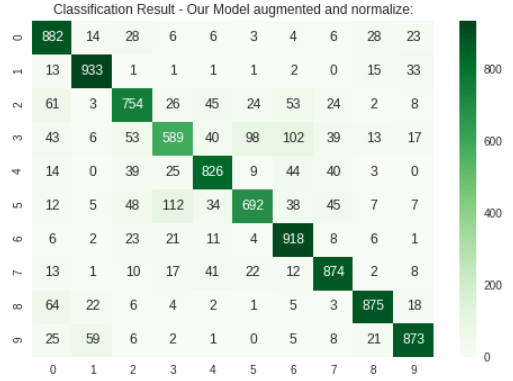
the results:

83% on train, 82% on val and 82% on test.

Ok. Now were getting there! First the accuracy is far better than before on both the test and validation set but not on the train set so we probably don't have a problem of overfitting anymore. Second the results across all the classes have gone up. Our methods for improving the model worked.



	precision	recall	f1-score	support
0	0.78	0.88	0.83	1000
1	0.89	0.93	0.91	1000
2	0.78	0.75	0.77	1000
3	0.73	0.59	0.65	1000
4	0.82	0.83	0.82	1000
5	0.81	0.69	0.75	1000
6	0.78	0.92	0.84	1000
7	0.83	0.87	0.85	1000
8	0.90	0.88	0.89	1000
9	0.88	0.87	0.88	1000
avg / total	0.82	0.82	0.82	10000



Not all the modification we tried documented, therefore we would like to mention some notes:

- At first that we tried to train the model we overfitted the data, therefore we tried methods such as dropouts and l2 regularization. While dropout help us to reduce overfitting (yet not enough) and increased the accuracy a bit, l2 regularization reduce the accuracy drastically, we tuned twice the weight decay parameter, first with 0.01 value and then with 0.001 for each layer. We assume the l2 regularization prevent the network to fit the training by punishing too much the weight matrix.
- We also set kernel initializer for 'he initializer' as a thought it would help the training phase converge faster to global minimum. While we have not noticed any significant improvement, it didn't reduce performance and therefore we did not remove that initialization method.
- After the 2 convolution layers we used batch normalization. We believe that normalize the output of those layers relative to our data is essential for the same reason that data normalization improved the network performance. At the last convolution layer,we chose not to use batch normalization – the thought is - at the last layer the network has crucial knowledge for the classification phase and thus we shouldn't interrupt (yet we chose to use dropout as we found it helpful to reduce overfit).
- As we noticed we have difficulty to improve our model accuracy we increased the layers filter size to overfit the data a bit, the results lead to 3 percent improvement (79 to 82), while the train accuracy 'overfit' the test by 1 percent.
- All of the notes mentioned were done by smaller amounts of epochs (20-30) and the learning rate scheduler changed relatively. The significant modification though, was made by the origin number of epochs – 140 and they are the experiments that mentioned above.

Transfer learning

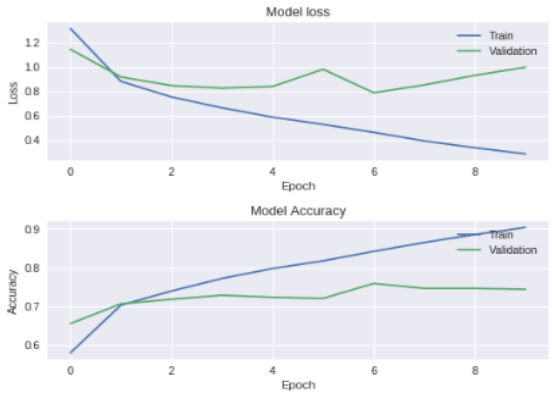
In this part we are trying to use transfer learning to build a model. First we looked at several of the already available models in keras. After some reading and checking we figure out that both resnet50 and inception would probably give the best results but they are both much too large to use "off the shelf" methods and would require using bottleneck solutions and preprocessing so we decided against using them.

We then tried VGG16 and VGG19, first we tried VGG16 for a couple of times and didn't get the best results (50% accuracy on some runs) but with VGG19 we got more then 70% accuracy on the first run, so we kept going with it.

To build the transfer model we took the VGG19 model from keras with imagenet weights. We did not include the top of the model and we define its input shape to be of the same shape of the cifar-10 dataset(32X32X3). We then tried 2 things. First, we took all the layers besides the last one and added a couple of our own as could be seen in the picture. We then run the model for 10 epochs, while we set the imagenet architecture parameters as untrainable, and trained just the layers we add to the model:

```
x = model.layers[-1].output
x = Flatten()(x)
x = Dense(1024, activation="relu")(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
x = Dense(512, activation="relu")(x)
x = BatchNormalization()(x)
x = Dropout(0.5)(x)
predictions = Dense(10, activation="softmax")(x)
```

The training process and the results:

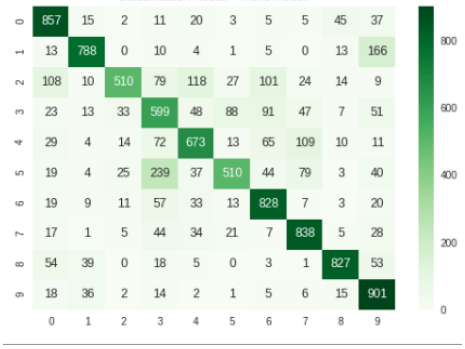



```
Training set accuracy: 0.9032
Validation set accuracy: 0.7441
10000/10000 [==============================] - 7s 739us/step
Test set accuracy: 0.7331
```


```
Model accuracy on test set is: 73.31%
              precision    recall  f1-score   support

           0       0.74      0.86      0.79      1000
           1       0.86      0.79      0.82      1000
           2       0.85      0.51      0.64      1000
           3       0.52      0.60      0.56      1000
           4       0.69      0.67      0.68      1000
           5       0.75      0.51      0.61      1000
           6       0.72      0.83      0.77      1000
           7       0.75      0.84      0.79      1000
           8       0.88      0.83      0.85      1000
           9       0.68      0.90      0.78      1000

avg / total       0.74      0.73      0.73     10000
```



Not good enough. Not only are we overfitting the model (90 acc on train, and 74 on val) but the results are actually worse than our model from before.

The model overfit to the training data, therefore the following attempts were made:

- Dropout
- L2/L1 regularization
- Batch Normalization
- Changing optimization methods – Adam / RMSprop / Momentum

All of those attempts failed, the result was quite the same, the test data converge to 73-76 percent accuracy on the test set. Where dropout and batch normalization did not change the test accuracy its failed to reduce the overfit as well.

The optimization methods did not change the accuracy as well, while RMSProp just reduce the test accuracy by 3-4 percent.

L2/L1 regularization reduce the overfit, yet we failed to overcome the plateau problem, the methods that were tried are changing the optimization methods to RMSProp / adam/momentum. For momentum we used ReduceLROnPlateau callback with patience of 2 epochs – while the learning rate is reduced the training accuracy did not improved.

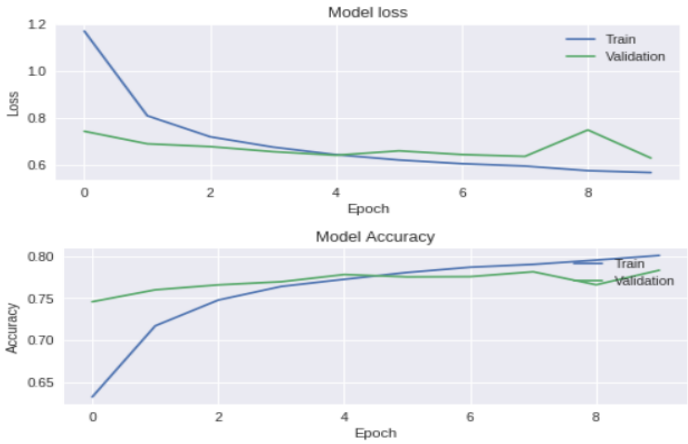
- All the methods were tried with and without data augmentation, data augmentation accuracy were always lower, we couldn't explain it.

The best result was achieved when we used VGG19 architecture instead, with the same settings that lead to the best performance (the same experiments were made for this architecture as well with no success).

The "best" transfer model we got:

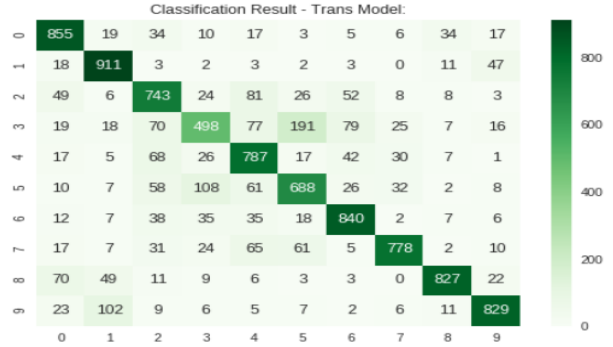
Layer (type)	Output Shape	Param #
input_1 (InputLayer)	(None, 32, 32, 3)	0
block1_conv1 (Conv2D)	(None, 32, 32, 64)	1792
block1_conv2 (Conv2D)	(None, 32, 32, 64)	36928
block1_pool (MaxPooling2D)	(None, 16, 16, 64)	0
block2_conv1 (Conv2D)	(None, 16, 16, 128)	73856
block2_conv2 (Conv2D)	(None, 16, 16, 128)	147584
block2_pool (MaxPooling2D)	(None, 8, 8, 128)	0
block3_conv1 (Conv2D)	(None, 8, 8, 256)	295168
block3_conv2 (Conv2D)	(None, 8, 8, 256)	590080
block3_conv3 (Conv2D)	(None, 8, 8, 256)	590080
block3_conv4 (Conv2D)	(None, 8, 8, 256)	590080
block3_pool (MaxPooling2D)	(None, 4, 4, 256)	0
block4_conv1 (Conv2D)	(None, 4, 4, 512)	1180160
block4_conv2 (Conv2D)	(None, 4, 4, 512)	2359808
block4_conv3 (Conv2D)	(None, 4, 4, 512)	2359808
block4_conv4 (Conv2D)	(None, 4, 4, 512)	2359808
block4_pool (MaxPooling2D)	(None, 2, 2, 512)	0
flatten_2 (Flatten)	(None, 2048)	0
dense_2 (Dense)	(None, 1024)	2098176
batch_normalization_3 (Batch	(None, 1024)	4096
dropout_3 (Dropout)	(None, 1024)	0
dense_3 (Dense)	(None, 512)	524800
batch_normalization_4 (Batch	(None, 512)	2048
dropout_4 (Dropout)	(None, 512)	0
dense_4 (Dense)	(None, 10)	5130

Total params: 13,219,402
Trainable params: 2,631,178
Non-trainable params: 10,588,224




```
Training set accuracy: 0.8008
Validation set accuracy: 0.7832
10000/10000 [==============================] - 6s 575us/step
Test set accuracy: 0.7756
-----------------------------------------------------------------
Model accuracy on test set is: 77.56%
```

	precision	recall	f1-score	support
0	0.78	0.85	0.82	1000
1	0.81	0.91	0.85	1000
2	0.70	0.74	0.72	1000
3	0.67	0.50	0.57	1000
4	0.69	0.79	0.74	1000
5	0.68	0.69	0.68	1000
6	0.79	0.84	0.82	1000
7	0.88	0.78	0.82	1000
8	0.90	0.83	0.86	1000
9	0.86	0.83	0.85	1000
avg / total	0.78	0.78	0.77	10000



So, we improved our accuracy on the test and validation to 77%-78%, which is better than most models but still worse than our model from the first section.

We were also able to reduce the overfitting issue.

We suspect several things that led to this but couldn't actually improve upon.

1. VGG19 / VGG16 are complex architectures for classification tasks such as CIFAR10, and though it converges faster, it fit the training too much.
2. ImageNet task is more complex as well and the weights trained for this task are less fit to CIFAR10 which his input dimension is much smaller.
3. Our hyper parameters tuning was not good enough to fit the transfer task.

Feature Extraction

We used the best transfer model from before for this section.

We omitted the last layer from the model and compile it. We then use the model to predict on the train set and the test set to create out features.

What we got are two sets.

Features_train in the shape of (50000,512) meaning for each sample of the 50000 we now have 512 new features.

Features_test in the shape of (10000,512) meaning for each of the 10000 we now have 512 new features.

The labels for each sample remain the same as before. We then used the features on 3 different "classic" ML algorithms:

logistic regression, random forest, KNN and SVM.

For logistic regression we used the default hyper-parameters.

For random forest we try several hyper parameters setting to try to improve its accuracy and reduce overfit – in the end (it doesn't even matter) we used:
n_estimators=50, max_depth=6, criterion='entropy', min_samples_leaf=3.

and in knn we set n_neighbors=5.

The results are as shown:

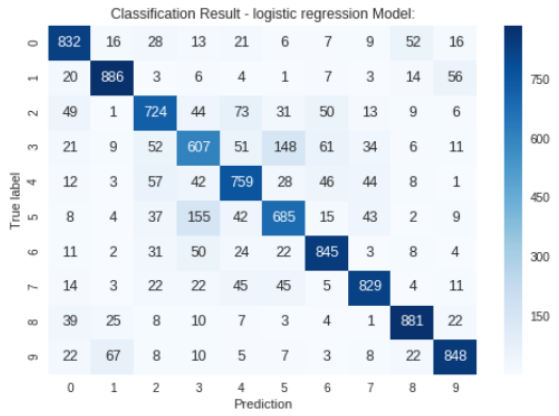
Logistic Regression:

```
model accuracy on test set is: 78.96%
             precision    recall  f1-score   support

          0       0.81      0.83      0.82      1000
          1       0.87      0.89      0.88      1000
          2       0.75      0.72      0.74      1000
          3       0.63      0.61      0.62      1000
          4       0.74      0.76      0.75      1000
          5       0.70      0.69      0.69      1000
          6       0.81      0.84      0.83      1000
          7       0.84      0.83      0.83      1000
          8       0.88      0.88      0.88      1000
          9       0.86      0.85      0.85      1000

avg / total       0.79      0.79      0.79     10000

------------------------------------------------------------------
0.87992
0.7896
```

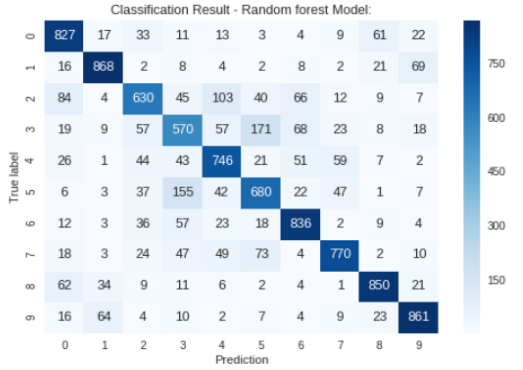


Random forest:

```
0.82016
0.7638
model accuracy on test set is: 76.38000000000001%
              precision    recall  f1-score   support

           0       0.76      0.83      0.79      1000
           1       0.86      0.87      0.87      1000
           2       0.72      0.63      0.67      1000
           3       0.60      0.57      0.58      1000
           4       0.71      0.75      0.73      1000
           5       0.67      0.68      0.67      1000
           6       0.78      0.84      0.81      1000
           7       0.82      0.77      0.80      1000
           8       0.86      0.85      0.85      1000
           9       0.84      0.86      0.85      1000

avg / total       0.76      0.76      0.76     10000
```



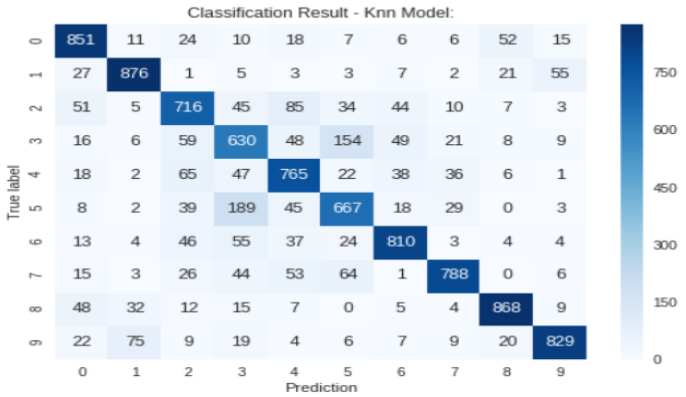
Knn:

```
              precision    recall  f1-score   support

           0       0.80      0.85      0.82      1000
           1       0.86      0.88      0.87      1000
           2       0.72      0.72      0.72      1000
           3       0.59      0.63      0.61      1000
           4       0.72      0.77      0.74      1000
           5       0.68      0.67      0.67      1000
           6       0.82      0.81      0.82      1000
           7       0.87      0.79      0.83      1000
           8       0.88      0.87      0.87      1000
           9       0.89      0.83      0.86      1000

avg / total       0.78      0.78      0.78     10000
```

```
0.8777
0.78
```



We can see that all the model performances are mediocre. None of them past the 80% accuracy and they are all at least a little over fitted. Random forest gave the worst result but was the least overfitted (is the one model we focused most for tuning because is the algorithm we most familiar with) and logistic regression had the best result but was over fitted, if we need to suggest what architecture one should select from those three we will recommend random forest because we believe he generalize best.

- Logistic regression lead to the best training result, this fact led us to the thought that may gradient based algorithm will lead also to good performance, therefore we tried SVM too. We have not documented his performance because the algorithm running time took much time to train and the result were relatively poor.

All the model achieves worse result compare to our model from part 2. But got pretty much the same result as our transfer model. that lead us to believe that the features extracted from the transfer model which were not that good or at least as good as the model itself, so the classic machine learning algorithms couldn't improve. It's important to note we didn't use any preprocessing methods for the features after the extraction and did very little hyper parameter optimization both of which can be critical for a machine learning algorithm to works best.

Final remarks

The work was hard. Building a right model and tuning to get optimal results was no easy task and we've ended up getting as far as 82% while the best result is beyond 90%. We learned that using the right augmentation and normalization is crucial for improving the model and enable him to learn.

We also notice that the choice of optimizer and metrics affect the model creation a lot.

Finally we saw how we can use existing solutions like VGG19 to get result and extract features from a network for a novel algorithm although we can see that for unstructured data ANN works better and easier then the classic approach.