**236606 Deep Learning – HW3**

Polosukhin Volodymyr 961144094

Bagrov Natan 203995121

**Part 1: A lightweight CNN.**

We propose a convolutional network with trainable parameters, that achieves an accuracy of over the test set.

Our proposed architecture is:

Layer (type) Output Shape Param #

=================================================================

conv2d\_1 (Conv2D) (None, 32, 32, 16) 448

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_1 (Batch (None, 32, 32, 16) 64

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_2 (Conv2D) (None, 32, 32, 16) 2320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_2 (Batch (None, 32, 32, 16) 64

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_1 (MaxPooling2 (None, 16, 16, 16) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_3 (Conv2D) (None, 16, 16, 16) 2320

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_3 (Batch (None, 16, 16, 16) 64

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_4 (Conv2D) (None, 16, 16, 32) 4640

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_4 (Batch (None, 16, 16, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_2 (MaxPooling2 (None, 8, 8, 32) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_5 (Conv2D) (None, 8, 8, 32) 9248

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_5 (Batch (None, 8, 8, 32) 128

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

conv2d\_6 (Conv2D) (None, 8, 8, 64) 18496

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

max\_pooling2d\_3 (MaxPooling2 (None, 4, 4, 64) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

flatten\_1 (Flatten) (None, 1024) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

batch\_normalization\_6 (Batch (None, 1024) 4096

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dropout\_1 (Dropout) (None, 1024) 0

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

dense\_1 (Dense) (None, 10) 10250

=================================================================

Total params: 52,266

Trainable params: 49,994

Non-trainable params: 2,272

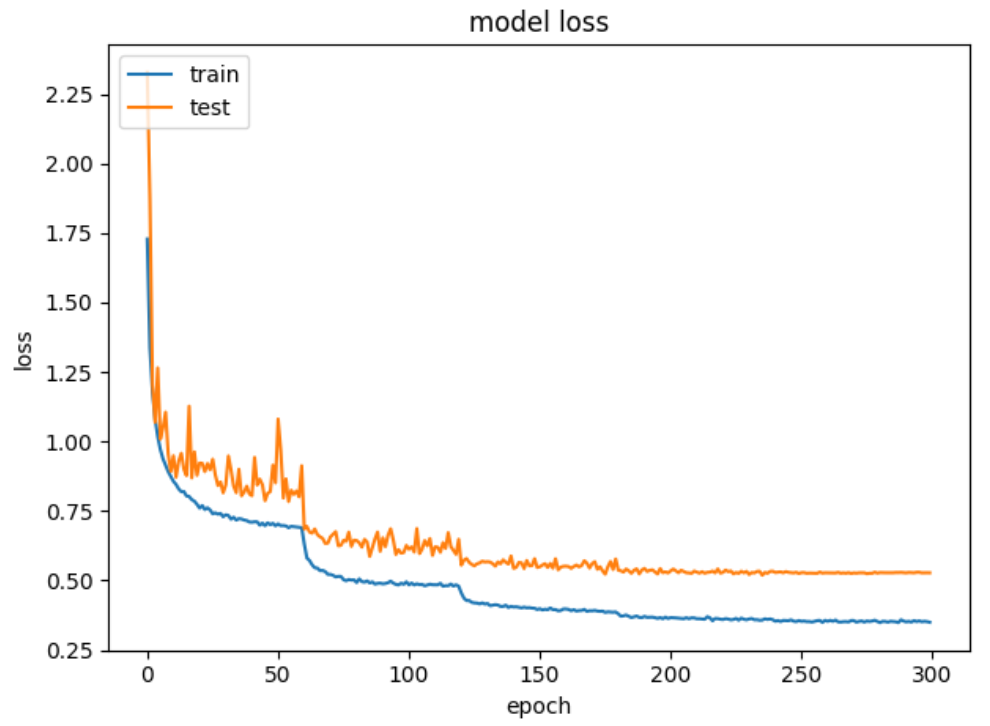
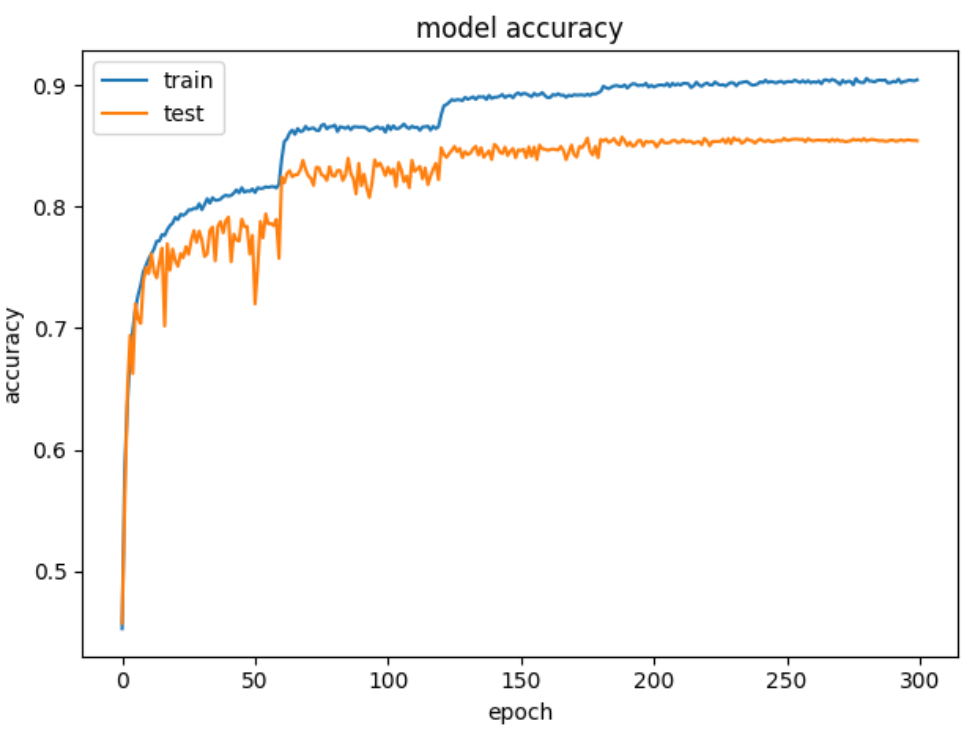
* The dropout layer had actually (in the proposed hyper-parameters that will be shown below) a drop-rate of zero, thus, it can be ignored.
* The convolutional kernels are of size 3x3.
* The pooling kernels are of size 2x2 with a stride of 2.

The hyper-parameters for the training procedure are:

**'batch\_norm'**: **True**,  
**'dropout'**: 0,  
**'weight\_decay'**: 5e-4,  
**'initial\_learning\_rate'**: 0.05,  
**'scheduler'**: step\_decay\_scheduler\_generator(initial\_lr=0.05, coef=0.2, epoch\_threshold=60),  
**'batch\_size'**: 256,  
**'augmentation'**: **True**,  
**'image\_data\_generator'**: ImageDataGenerator(  
 featurewise\_center=**False**, *# set input mean to 0 over the dataset* samplewise\_center=**False**, *# set each sample mean to 0* featurewise\_std\_normalization=**False**, *# divide inputs by std of the dataset* samplewise\_std\_normalization=**False**, *# divide each input by its std* zca\_whitening=**False**, *# apply ZCA whitening* rotation\_range=5, *# randomly rotate images in the range (degrees, 0 to 180)* width\_shift\_range=0.1, *# randomly shift images horizontally (fraction of total width)* height\_shift\_range=0.1, *# randomly shift images vertically (fraction of total height)* horizontal\_flip=**True**, *# randomly flip images* vertical\_flip=**False**) *# randomly flip images*

We used batch normalization, no dropout, an initial learning rate of 0.05 which was reduced by a factor of 5 per 60 epochs, L2 regularization with weight decay of 0.0005, a batch size of 256 and an augmentation with the parameters shown above.

The training progress is visualized below:

**

Several observations:

1. As can be deduced from the graphs, the learning rate scheduler reduced that rate per 60 epochs (the accuracy/loss graphs have a ‘step’ per 60 epochs).
2. In epoch 126, the model reached the accuracy of 0.85 for the first time.
3. The maximum accuracy of is achieved at epoch 189.
4. The training procedure achieves its convergence in approx. epoch 200.
5. The training loss in the end of the training procedure is (and the accuracy is ), which means that there are improvements that can be done. The achieved accuracy on the test-set is enough, though ☺.
6. It seem that we don’t have an over-fitting.

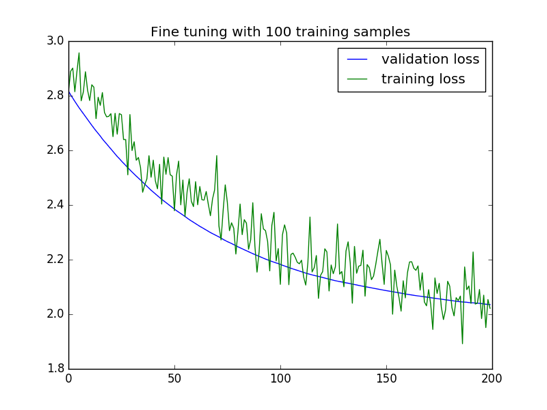
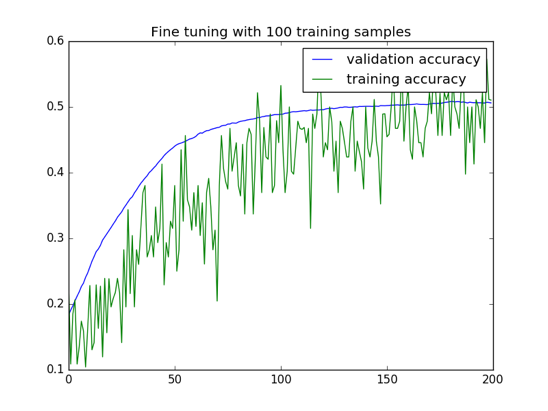
The python script is located under ‘lightweight\_cnn/light\_cnn.py’ and the model parameters are located under ‘lightweight\_cnn/saved\_models/light\_cnn\_weights.189-0.857.h5’. There is an option to train the model from scratch or to load the trained parameters.

**Part 2: Transfer Learning.**

Section 1:

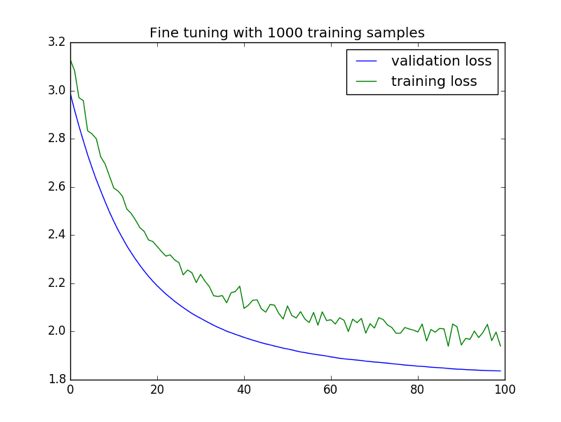
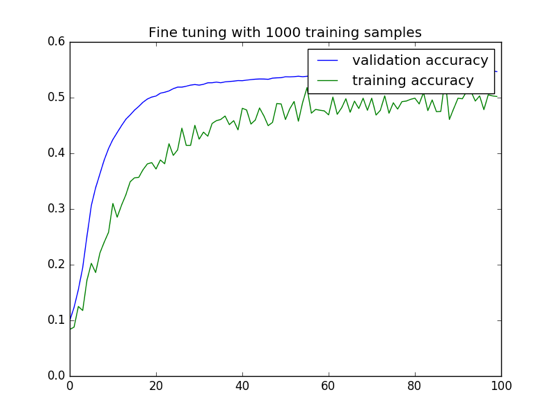
We trained the network with Adam optimizer with learning rate of and weight decay and batch sizes of 8, 32 and 128 for 100, 1000 and 10000 training samples respectively for 100-200 epochs (validation accuracy converged there), also we used augmentation of rotations until , horizontal and vertical shifts until , zoom of and horizontal flips (we think that vertical flips does not make sense e.g. rotating automobile).

For 100 samples, after 200 epochs, network reached accuracy of 0.5062 with accuracy and loss graphs below:



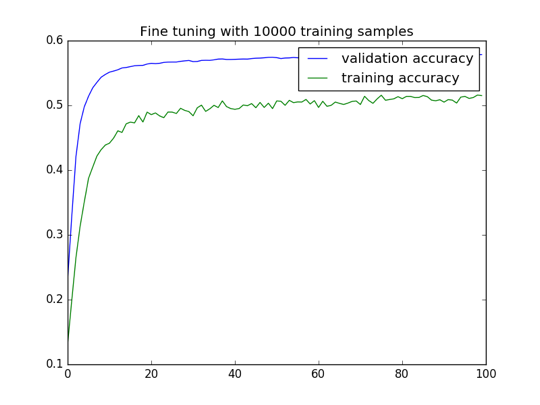
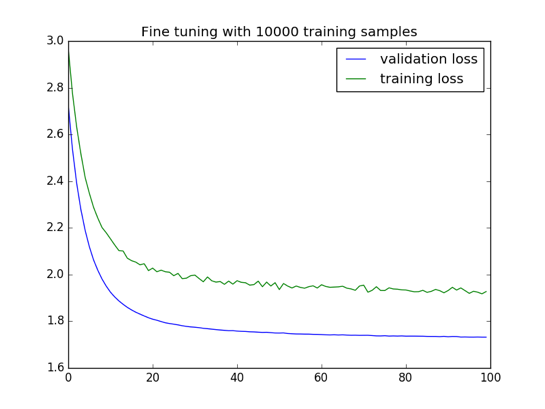
You can find trained weights in file cifar10vgg\_finetuning\_100\_trainset.h5

For 1000 samples, after 100 epochs, network reaches accuracy of 0.5484 with accuracy and loss graphs below:



You can find trained weights in file cifar10vgg\_finetuning\_1000\_trainset.h5

For 10000 samples, after 100 epochs, network validation accuracy reached 0.5785, see the accuracy and loss graphs below:



You can find trained weights in file cifar10vgg\_finetuning\_1000\_trainset.h5

The results summary you can see at the table below

|  |  |  |
| --- | --- | --- |
| Number of samples | Train accuracy | Test accuracy |
| 100 | 0.5104 | 0.5062 |
| 1000 | 0.4962 | 0.5484 |
| 10000 | 0.5151 | 0.5785 |

What is interesting, that train accuracy and loss are worse than validation, it can be explained by taking average on batches for calculating train accuracy and augmentation. It is also clear that increasing amount of data improves performance.

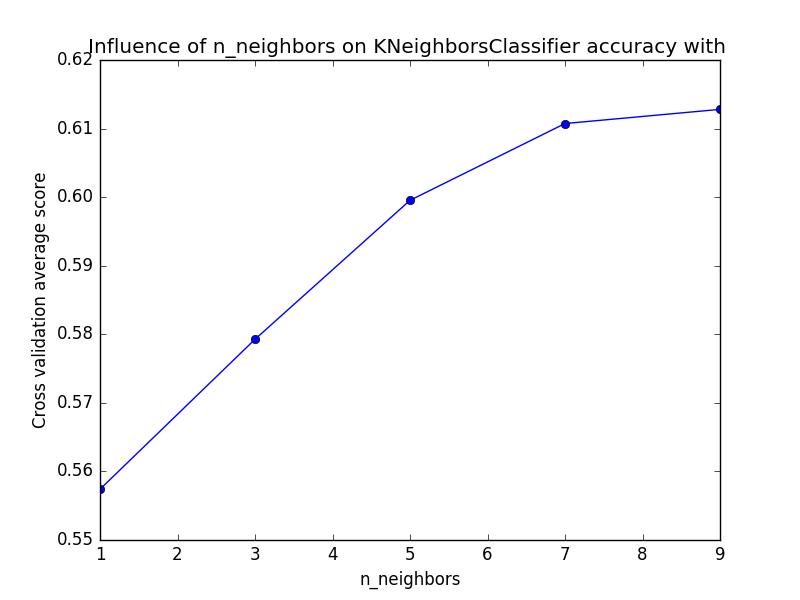
Please find the implementation under transfer\_learning/fine\_tuning.py

Section 2:

We tested performance with default k=5, and then decided to search for optimal k between odd number in range 1-10 using 3-fold cross validation (which worked very slow, and saturated on 9, that is why we did not test other values), and found out that optimal k is 9, see results in table below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Default (k=5) | | Fine tuning (k=9) | |
| Number of samples | Train accuracy | Test accuracy | Train accuracy | Test accuracy |
| 100 | 0.438520 | 0.427900 | 0.449240 | 0.443000 |
| 1000 | 0.513880 | 0.507300 | 0.532640 | 0.524800 |
| 10000 | 0.595920 | 0.572300 | 0.601100 | 0.581600 |

And here is graphs of influence that different k has on validation accuracy:



From the table above, we can conclude, that increasing amount of data, as expected, improves accuracy. There is the reverse side of the medal – more data we have, slower kNN predict the label for new sample. It is actually almost inapplicable to use kNN with such big number of features (512).

We can see that this embedded approach works worse than the first one (training single layer), though it is faster in required training time and usually slower in prediction time (since in fine tuning we need to run topless network and new trained layer, which is just another multiplication, and here run topless network and afterwards search in kd-tree).

Section 3:

Given the fact that the model was trained to classify CIFAR-100 classes and given the fact that most of the CIFAR-10 classes related somehow to CIFAR-100 classes, we would like to propose a **probabilistic approach** to classify CIFAR-10.

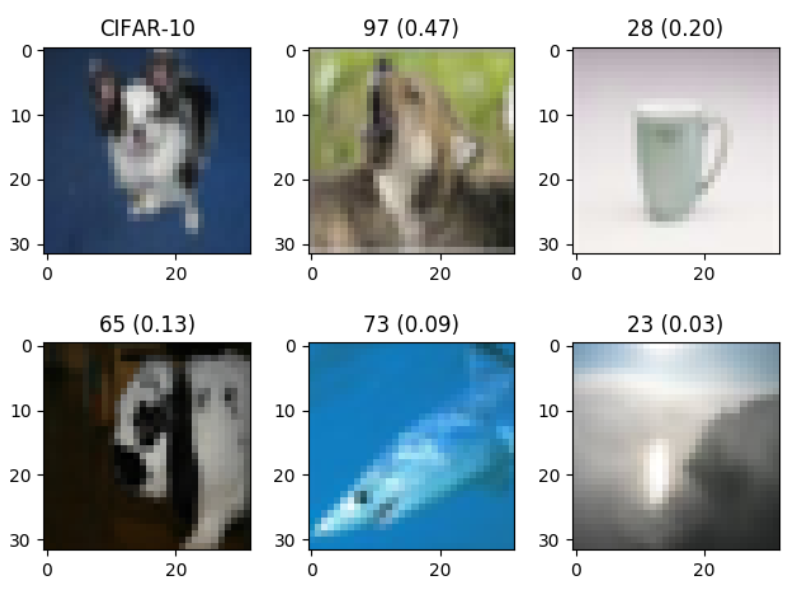
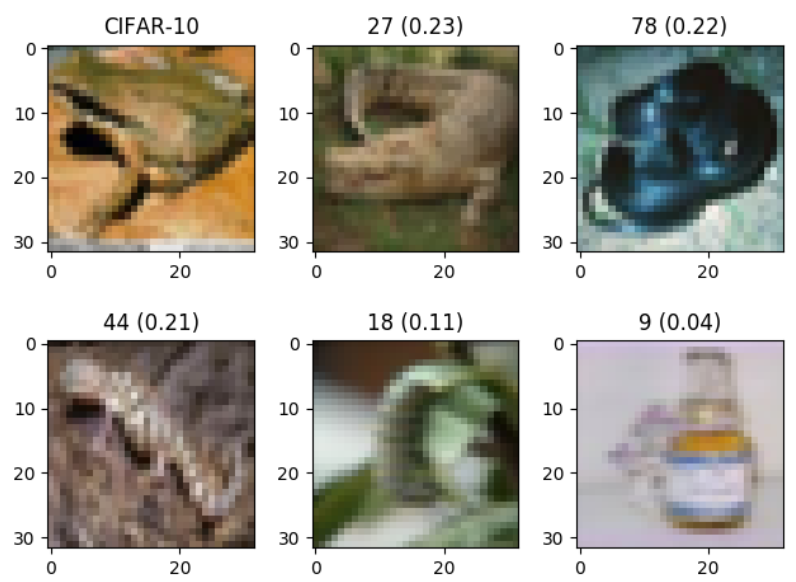
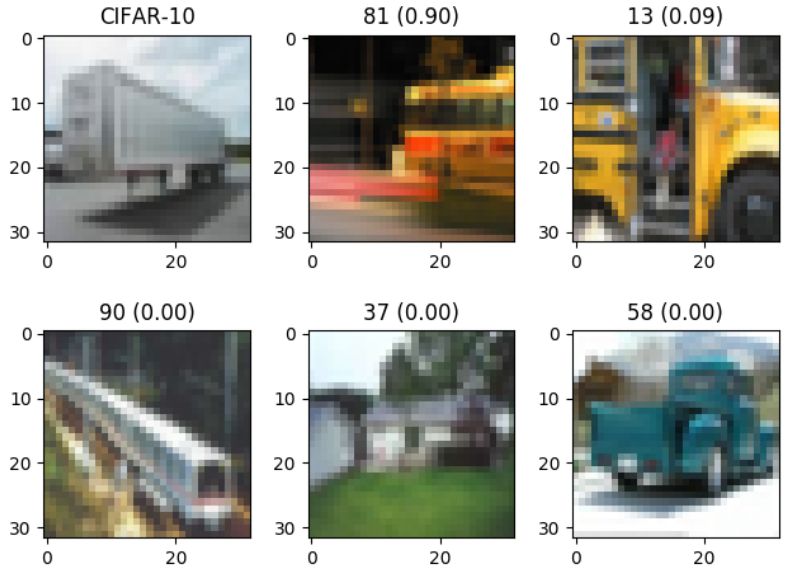
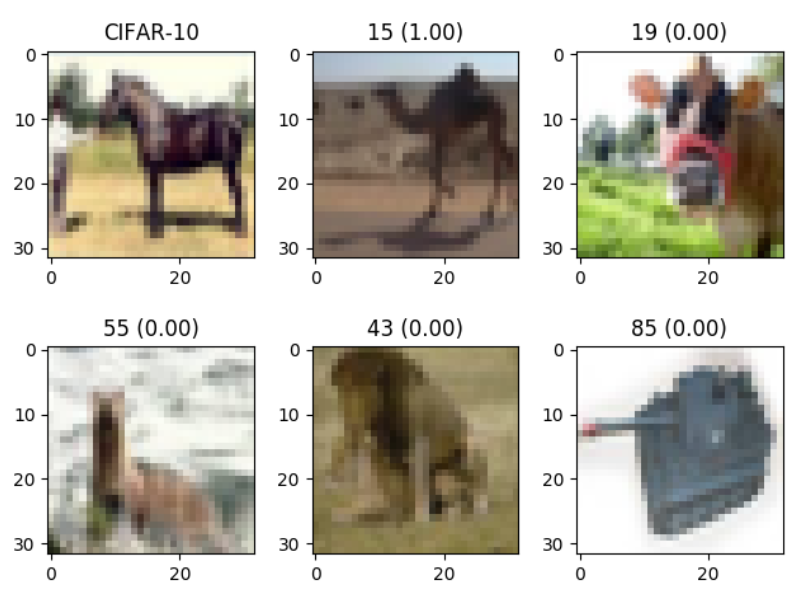
We take the output of the model, which is 100 soft-max activations, and consider them as probabilities, denoted as (where is a sample (32x32x3) and is a vector of 100 probabilities, a probability at index is the probability of to be labeled as ).

The original classifier classifies as ; Instead, in our approach, having a class (from CIFAR-10) we consider as and actually **model the likelihood of a sample to be classified to class (from CIFAR-100’s domain) given the fact that its true class is (from CIFAR-10’s domain).**

This set of possible labels and their labeling probability is exactly .

Some intuition: **even though the classes of CIFAR-10 and CIFAR-100 are distinct**, suppose we have a sample from CIFAR-10 and its true label is ‘truck’, we assume that this sample is to be classified as ‘pickup truck’ or ‘tank’ or ‘tractor’ or maybe even as ‘castle’ (classes of CIFAR-100), etc.

To illustrate our intuition, we show samples (each from a different class) from CIFAR-10, and their top 5 similarities from CIFAR-100 (top 5 activated soft-max neurons):

Observations:

A **frog** (CIFAR-10), looks like an **crocodile** (CIFAR-100) with a probability of 0.23, but also might look like a **snake** or even a **lizard**.

A **dog** (CIFAR-10), looks like a **fox** (CIFAR-100) with a probability of 0.47

A **truck** (CIFAR-10), looks like a **bus** (CIFAR-100) with a probability of 0.90

A **horse** (CIFAR-10), looks like a **camel** (CIFAR-100) with a probability of 1.00

The training algorithm:

1. Normalize the training-set according to the values calculated by the pre-trained model.
2. Feed the training-set into the pre-trained model and get soft-max activations.
3. Take an average on the activations that are truly labeled as (known from test set), for in number of classes and get probabilities.
   * the probability of a sample classified (by the pre-trained model) as (100 classes) to be classified as (10 classes).

The prediction algorithm (**1st approach**):

1. Normalize the test-set according to the values calculated by the pre-trained model.
2. Feed a sample (CIFAR-10) to the pre-trained classifier, output is a 100 vector of soft-max activations.
3. Predict (take argmax) the class of this label with respect to CIFAR-100 and get .
4. Output

In-fact, given Naïve Bayes formula: , since and is constant when predicting its label, we just take .

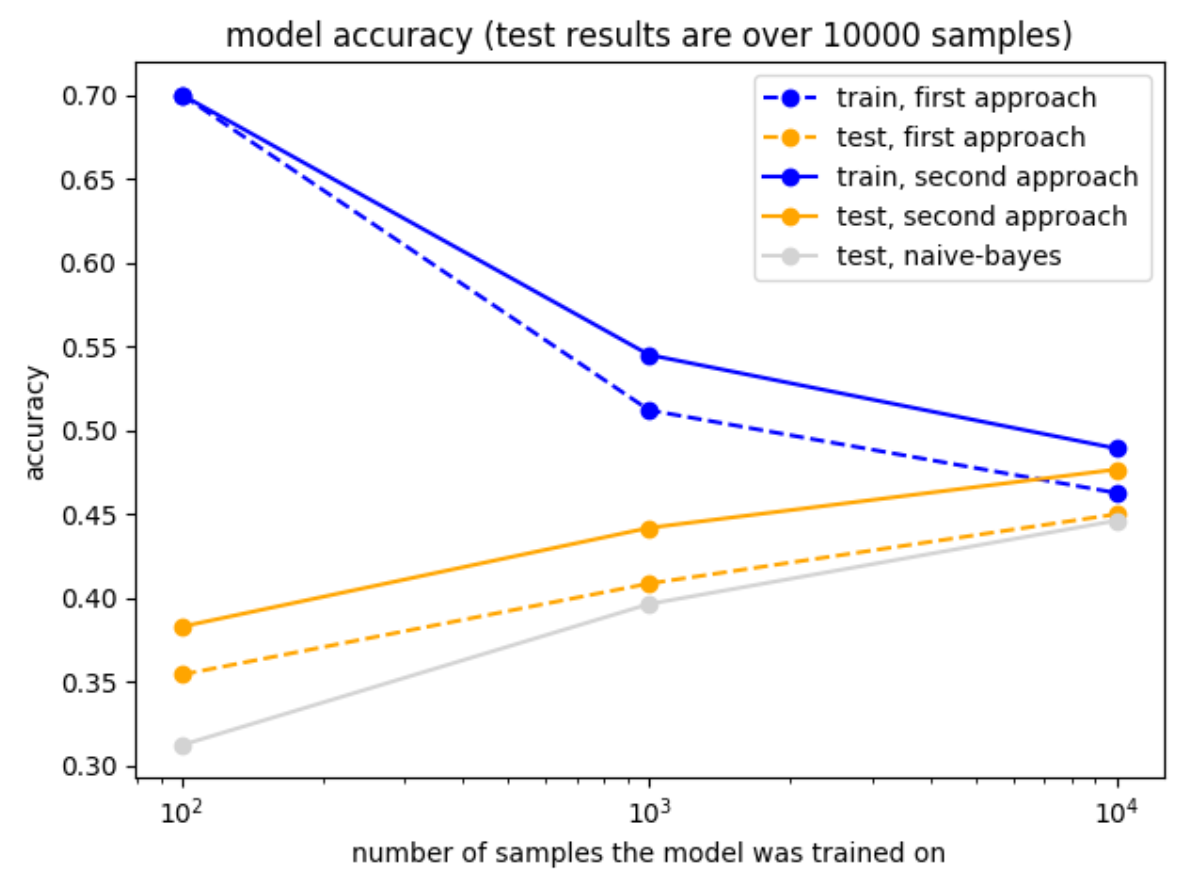
The prediction algorithm (**2nd approach, more gentle**):

1. Normalize the test-set according to the values calculated by the pre-trained model.
2. Feed a sample (CIFAR-10) to the pre-trained classifier, output is a 100 vector of soft-max activations.
3. Output (for all )

The difference between the first and second approaches is that in the first approach we first classify the sample to its CIFAR-100 class, and then take the CIFAR-10 class that gives us the maximum probability, where in the second approach we multiply the joint probabilities and take the maximum.

This model achieves accuracies of **, ,**  trained on 100, 1000, 10000 samples respectively using the first approach, and of **, ,** using the second approach.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Our approach (1st) | | Our approach (2nd) | | Naïve Bayes over one-hot |
| Number of samples | Train accuracy  over taken samples | Test accuracy over 10000 samples | Train accuracy  over taken samples | Test accuracy over 10000 samples | Test accuracy over 10000 samples |
| 100 | 0.7 | 0.354 | 0.7 | 0.383 | 0.312 |
| 1000 | 0.512 | 0.408 | 0.545 | 0.442 | 0.396 |
| 10000 | 0.462 | 0.45 | 0.489 | 0.477 | 0.446 |



As can be seen from the graph, this model actually outperforms the Naïve Bayes classifier (trained with one-hot labels) on small training sizes (100, 1000), and performs slightly better on large training sizes (10000). That makes sense since our model approximates the probabilities given soft-max and not approximating them from one-hot labels; when working with large amount of training data, the approximations get closer.

Compared to previous transfer learning approaches, the current approach underperforms both of standard previous approaches, and has another disadvantage, since assumes that there is logical connection between classes of old trained model and classes of new, but has also great advantage – it is computationally easiest, simple in implementation and has no hyper parameters to fine tune.