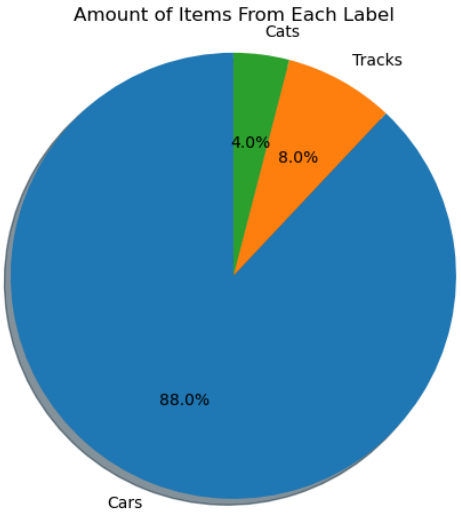
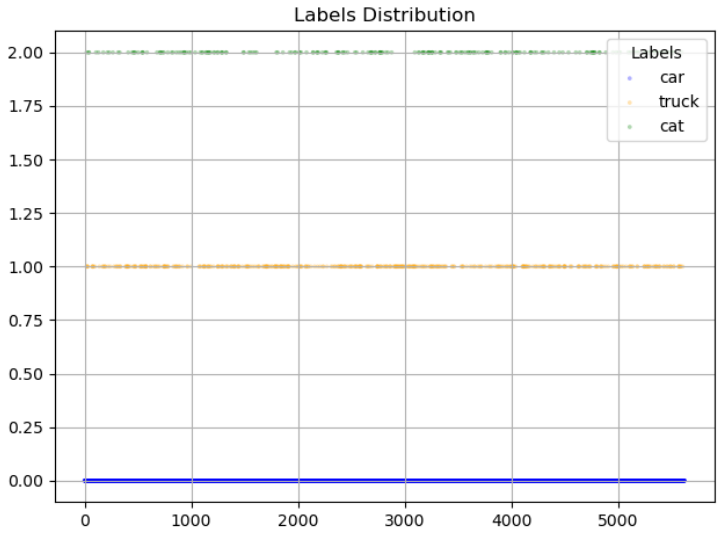
Training Neural Network

1. **Data**
   a. The code for this section can be found in "inspectData.py".
   b. Our data contains images from size 32x32x3.
   c. 5626 samples in total in the training set, labeled as follow :



   d. The distribution of samples types along our dataset is uniform:



   e. We can see that the data is imbalanced. This is an important fact and we will need to take special actions to deal with it as we saw in the recitation. In addition, the data is not sequentially ordered – important fact for us to know when we train the model.
   f. I tried to find some outliers by calculate mean and std of images and see extreme values but I didn't find any exceptional results.
   g. The test data is very similar to the training data in aspects of division for classes and uniform distribution along the data set. This fact may lead

us to miss situations of over fit or situations of bias towards the large class.

2. **Evaluation**

The evaluation script (evaluation.py) loads the pre trained net and tests it with the test set. This is a classification problem so I initially decided to evaluate the model by simply checking if its highest output for a given input matches the input label. My score for the model was (correct / total) = 88% for the pre-trained model. This fact seemed suspicious because I already knew my data is imbalanced.

I decided to check the performance of the model by classes – means to check which classes it recognizes better. The pre-trained model succeeded on 100% of the cars but 0% of trucks and cats. I realized that my evaluation method has a problem and I decided to change its score to be the average of scores it gets by classes. This gave me score of 34%.

This method felt a bit not based so I returned to research and I found that a good evaluation for an imbalance data problem is G-mean:
score = recall (a) * recall (b) * recall(c) ** (1/3)
When **recall (a) = True Positive / True Positive + False Negative.**
G-mean values are 0 to 1, while higher value implies better model.
This score gives **zero** for the pre-trained model, which is quite reasonable.

I saw many metrics for evaluation. In our case, I do not find any class prediction more important than another does. This observation made 'False Positive' and 'False Negative' cases to be equally bad for us, thus many metrics that trying to prefer one failure upon the other are irrelevant for us.

By the way – Which metric will you choose for the competition?

3. **Training**

   a. I started with naïve training loop (training.py) and I found out that my model still strongly influenced from the imbalanced data. As I raised the number of epochs, there was an improvement on prediction of Trucks, but still Cats remains very week category.
   For 100 epochs in the naïve training, we get Accuracy: 90.4%, Average Accuracy: 55% and G-Mean: 0.47.
   b. My next step was to use regularization (by adding decay weight to the optimizer) to make my model better handle with imbalanced data as we saw in the recitation. For instance, the results were the same or even worst, but later on, when I improved my training with other aspects, I found the regularization to be good tool to deal with the imbalanced data.

c. I decided to try to improve my data. For instance, one of the team members said that he found a lot of labeling errors of Cat class, which is already the smallest. I fixed some of the labels and tried to train my model again and I got slightly better results but still very bad predictions for trucks and cats.
d. My next step was to try to perform weighted sampling, so the model will be trained on the same amount of samples from each class. I used some package wrapping 'sampler' ("MultiLabelBalanceRandomSampler") and sample samples uniformly by taking each time a sample from the least sampled class. This also did not help much.
e. I applied some transformations to the data, tried to train the model on regular data, and then transformed data. I got surprising results: now the model identified 100% of cats, but only 30-40% of trucks and cars (G-mean 0.51).
f. I applied learning rate decay with stepLR scheduler and I did not see significant improvement.
g. I found a critical error in my process – I saved the fixed data as images, mean that it was transformed and normalized in order to be saved as image but I trained my model on the raw test set. I saved the test set as images, loaded it and normalized it and now I got much better results: Accuracy: 82.4%, Average Accuracy: 82.6%, G-mean: 0.826.
h. I noticed that my sampler is maybe too strict. It samples the exact same amount from each class and I suspected that it has too strict behavior. I used instead in WeightedRandomSampler with different weights (by class) and my results improved.
i. I found out that the process of hyper parameters tuning may be very long and exhausting. After many tries and combinations of the above stages, I reached a nice results of 82.4%, Average Accuracy: 82.6%, G-mean: 0.826.

4. **Learning Rate Analysis**

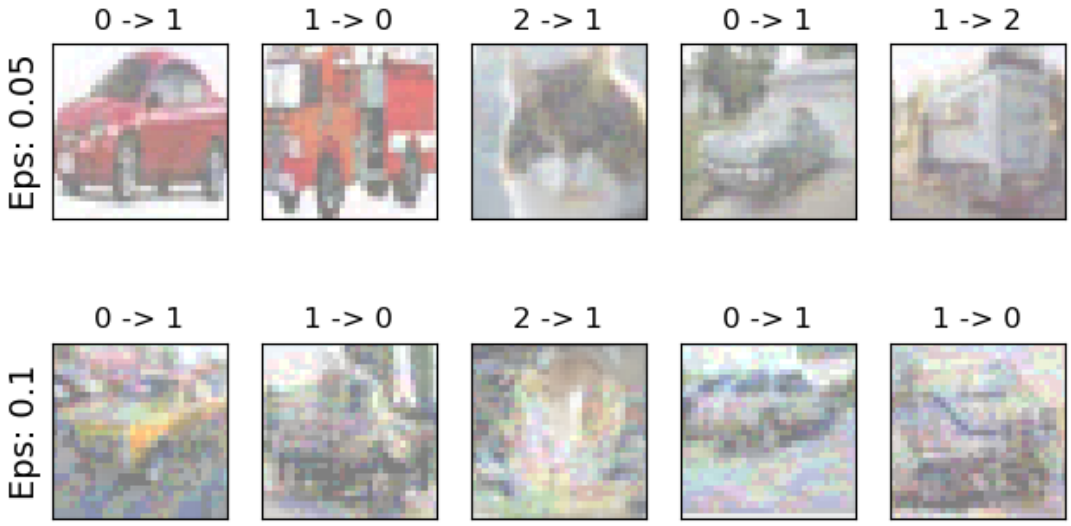
I plot the log of loss values according to learning rates of (0.5, 0.05, 0.005, 0.0005, and 0.00005). It is easy to see that for too high / too low learning rates the model does not converges (the log function maintains the linearity, I use it from visual reasons – to avoid too large difference between values).

This phenomenon occurs because of the fact that when trying to optimize the loss function, we calculate the gradient and then we take a step against it and the learning rate controls the size of the step that we are taking. Too large steps against the gradient might cause our function to "jump" between different local minima sockets or even reach to high loss value in the opposite direction. Too low values may be stuck in local minima with no enough power to get out. Other option is that the very small value will converge and lead us eventually to the minimum but it will take very long time – so in the seen time, it looks like it does not converges.

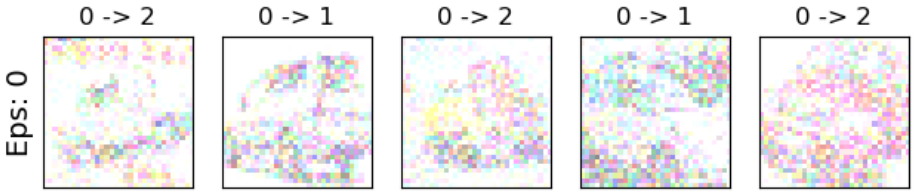
5. **Adversarial Model**

I implemented two methods of Adversarial examples:

a. Adversarial model with different epsilons (strength of the noise). For each (x, y) – image, label, I ran the original model and predicted p1. If the model predicted right label, I taking the gradients of the loss calculated during the prediction and creates a matrix of signs of the gradient. Then I perform: Image + gradient_sign * epsilon, means adding epsilon in the direction of the gradients. Then I performing another prediction on the image+noise and check if the model was right. The results for this method were really good: 10.5% accuracy and 12.8% average accuracy:



b. My second approach was to try to implement the method we saw in class. I initialized a random noise and for each iteration, if the model predicted well, I added the noise to the image and calculated the loss of noise+image against wrong label (original label + 1). Then I added the gradients to the noise (multiplied by learning rate). Finally, I ran gain over the data and tried to predict image+noise. The accuracy of the model was 63.8 with average of 47.5, but the noise was too strong:


0 -> 2
0 -> 1
0 -> 2
0 -> 1
0 -> 2
Eps: 0