Image Classification Model using CNN

Yasmin Heimann, 311546915

November 11, 2020

1 Data Inspection and Preparation

We were provided with a data set of images (as apickle file), containing 5,625 images of cars, trucks and cats.

1. **Mis-Labeling.** At first, I inspected the data manually by visualizing the photos using the imshow() function of matplotlib.pyplot package. Then, I notices that some of the labels were incorrect. Due to the fairly moderate size of the data set, I decided to manually re-label the data and fix mis-labelings.

2. **Data Imbalance.** After fixing the data labels, I have extracted the data's distribution, by counting the number of appearances of each class in the data, which is shown as follows:

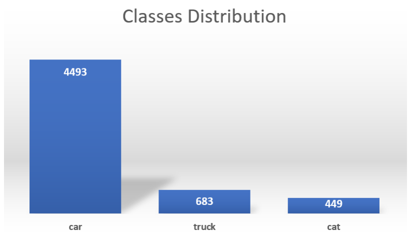


Figure 1: Classes Distribution in the Train Set

This distribution shows that our training data is imbalanced, which would cause our model to be biased towards the more frequent class - car, in this case.

To solve this issue, I have examined the following ways:

- **Weighted Sampler**
  - The trained data is sampled by the given probabilities of each class (label) in the data. That means, that if the class 'truck' is only 12% of the data, the sampler will sample it with a relatively higher probability than it will sample 'car', which is 80% of the data. That sampling is done to make sure each class is represented evenly in the training process, and by that to 'fix' the imbalance.
- **Weighted Training Loss**
  - The train loss recieves the data distribution as a parameter, and based on the distribution it creates a weight for each class - high weight for small classes and vice versa. According to the class weights, the loss function penalizes classes with low frequency much higher (as they have high weights). That is done to "force" the model to take the under-represened classes into account and forces the optimizer to optimize these samples too, in an even way, due to the weighted penalizing.
    The Loss used in the training is Cross Entrophy Loss.

Without any Imbalance fixing method, the best model accuracy reached was around ~60%. When Applying either the weighted sampler or weighted training loss, the accuracy of the model raised to be around ~87%. I chose to use the Weighted Training Loss as it shows slighly better results.

2 Pre-Trained Baseline Model Evaluation

2.1 Evaluation script and Metrices

The considerations taken in the evaluation step were as follows:

1. **Evaluation Mode:**

(a) The supplied network cotains droupouts, which are relevant only for the training step. Thus, I use the $model.eval()$ function to indicate the model that this is an evaluation call, and by that the dropout layers will be "turned off".

(b) When in evaluation mode, gradient calculations are useless, thus disabling it by the function no_gard() will result in reduced memory consumption for computations that would otherwise calculate the gradients with no further use.

2. **Evaluation Metric - Evenly Weighted Accuracy:** The test set distribution, similar to the train set, is biased towards the 'car' class, with 503 cars samples, 51 trucks and 71 cats. Thus, we shall use a weighted accuracy which takes the average accuracy, based on each class' accuracy. Then, the accuracy of the model is measured by counting the percentage of samples that the model was correct on, relatively for each classification class. That means that we get the success rate for each class, and then take the average success rate of all class.

   The calculation is as follows:

$$\forall i \in \{Classes\} \implies C_iA = Class - Accuracy = \frac{\sum_{x\in\{C_i - Samples\}} 1_{\{y=f(x)\}}}{|C_i - Samples|}$$

$$Total - Accuracy = \frac{\sum_i C_iA}{|\{Classes\}|}$$

   Where x is the image, y is its true label, f is the model and $f(x)$ is the model prediction.

3. **Confusion Matrix** - A matrix where each row represents the true class of a sample, and each column represents the predicted class by the model. The diagonal indicates the success of the model prediction, as the i, i cell tells that the true label matches the prediction. This evaluation method is used to describe the performance of a classification model on a set of test data, and indicates how many times it has got the right or wrong prediction for each class. In our model, this will be a 3×3 matrix.

2.2 Trained Baseline Model Evaluation

I ran the given trained model on the re-labeled test set (the supplied dev.pickle) and on the original train set to measure its training accuracy. The results are as follows:

Dataset	Total Accuracy	Car	Truck	Cat
Original Train Set	33%	100%	0%	0%
Re-labled Test Set	33%	100%	0%	0%

Confusion Matrix:

$$\begin{array}{c|ccc} & car & truck & cat \\ \hline car & 503 & 0 & 0 \\ truck & 51 & 0 & 0 \\ cat & 71 & 0 & 0 \end{array}$$

Table 1: Pre-Trained Model Success

We can see from the confusion matrix, that the given model had labeled all of the test images as car, which explains why it gets 100% accuracy on 'car' and 0% on the other classes. This results in a weighted loss of $\frac{1}{\#classes} = \frac{1}{3} = 33\%$.

As we saw on section 2, the model is most likely affected by the imbalanced data distribution, which makes the model biased towards the most frequent class - 'car'.

Conclusion The given model is predicting very poorly, and is not giving any insights on the image it tries to predict. In other words, it can't really predict an image label, as it recognizes only cars.

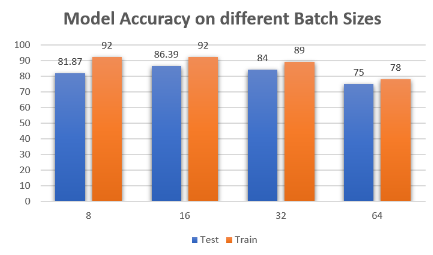
3 Hyper-parameters Tuning and Model Parameters

Throughout the model training, there are many hyper-parametes that should be learned and optimized for the specific network chosen. I have examined these parameters and present my conclusion in the following section.

3.1 Batch Size & Epoc

Epoc is the number of passes of the entire training dataset the machine learning algorithm has completed. The batch size determines the number of training examples utilized in one iteration.

To optimize these parametes, for its usage in the final training, I have examined the affects of the epocs and batch sizes on the model accuracy, which has resulted in the following:



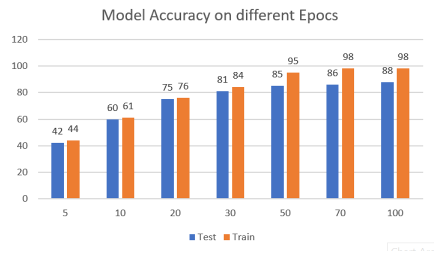


Figure 2: **Left:** Batch varying sizes trained using SGD with LR=0.001 and evaluated on the train and the test set. **Right:** Epocs varying sizes trained using SGD with LR=0.001 and evaluated on the train and the test set.

As for the batch size, the range between 16-32 performs the best. Low batch sizes takes more time to learn but can be more accurate. Larger batch sizes, like the 64, may lead to poor generalization in the model, which we can see is consistent with the 75% accuracy rate on the test set, that is much lower than in smaller batches.

As for the Epoc size, we can see that around 30-50 epocs we can already reach high accuracies.

3.2 Optimizer

For finding the best optimizer for the model, I have decided to check SGD (with momentum) vs. ADAM, as ADAM optimizer already combines the advantages of two other SGD extensions - RMSprop and AdaGrad. I simulated momentum rates and found 0.9 to consistently bring good results. I have trained the model twice - once for each optimizer, on the same parameters. I have evaluated the models on the test set (train was 98% accuracy for both) ,and the results are as follows:

#epocs	**#batches**	**LR**	**weight decay**	**momentum**
100	32	0.001	0.01	0.9 (SGD only)

$\rightarrow$

Optimizer	**Weighted Accuracy**	**Car**	**Truck**	**Cat**
SGD	87.73%	92%	80%	90%
ADAM	88.18%	92%	82%	90%

Table 2: SGD vs. ADAM

We can infere that both optimizers work well with the data and parameters given, thus an arbitrary choice can be made. I chose to use the ADAM optimizer as my final model to submit due its slightly better predictions.

3.3 Regulariozation

The weight_decay parameter in the optimizer adds a $L2$ penalty to the cost, which can effectively lead to smaller model weights. To optimize this parameter, I have simulated the following values:

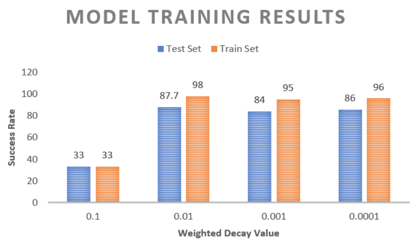


Figure 3: **Weighted Decay Values Optimization**: Training with different weighted decays to determine the best value for the model. The model was trained using 100 epocs, 32 batch size and SGD with LR=0.001 and momentum = 0.9.

From the figure above, we can see that 0.1 value is too high - results in very bad predictions, and in fact made the model to predict only cars. Values lower than 0.1 had shown good results, and should be used to train the model.

3.4 Data Augmentation on Images

Image Augmentation is the process of generating more images from the existing images, by using various transformations e.g. flipping the image, rotating etc. This process can help to enrich and enlarge the data set, while decrease overfitting.

I have suggested a few tranformations for usage in my code, for example a Color Jitter Transformation which randomly changes the brightness, contrast and saturation of the given image.

I have decided to not apply this method, as the given data, after re-labled, had shown good results in the learning process, so that additional images wasn't definitely necessary at this stage.

3.5 Hyper-Parameters Summary

Batch Size	Epocs	Weighted Decy	Optimizer
16-32	30-100	0.01-0.0001	SGD or ADAM

Table 3: Best Values and Parameters for Model Training

4 Learning Rate Optimization

The learning rate is a tuning parameter (hyper-parameter) we use in an optimization algorithm, like SGD, which determines the size of the step we take towards the minimum of the loss function (the opposite direction of the gradient), at each iteration of the training.Thus, tuning the LR (=Learning Rate) is important, as un-suitable learning rates may raise the following problems:

1. Low learning rates result in very small steps towards the minimum which may slow down the learning process significantly or even make the model to get “stuck” in a wrong local minimum.

2. High learning rates will decay the loss faster, as it takes bigger steps, but can learn a sub-optimal set of weights and get stuck at bad loss values or a wrong local minimum.

I have examined the affect of the LR on my model training, by running different LR values over 500 epocs with SGD optimizer, and analyse the convergence rate. The results are a follows:

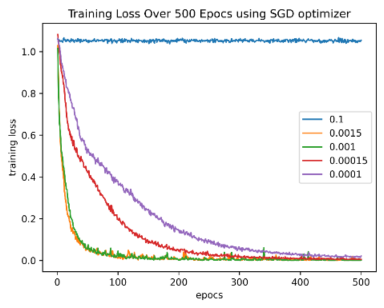


Figure 4: Learning Rate Convergence Rate over 500 Epocs

It is clear to see that the range of 0.001 LRs has the best convergence rate at aroung 40-50 epocs, which makes it the best LR candidate. When taking LR=0.1, we see it is too high, so it gets “stuck” at worse values of loss. When evaluated seperately it was found to predict only trucks on the data. The range of 0.0001 LRs are converging slowly at about 200-300 epocs. The lower the LR the more epochs needed to convergence, with the chance of getting stuck in a wrong minimum without getting out of it due to steps that are too small.

5 Model Evaluation

I have trained my final model with the tuned hyper-parametes as discussed above. I evaluated my final model on the re-labled evaluation and train set, and got the following results:

#epocs	**#batches**	**LR**	**optimizer**	**weight decay**
100	32	0.001	ADAM	0.01

$\rightarrow$

Data Set	**Weighted Accuracy**	**Car**	**Truck**	**Cat**
Evaluation	88%	92%	82%	90%
Train	98%	95%	99%	99%

Table 4: **Trained Model Success**

$$\begin{array}{c} \\ car \\ truck \\ cat \end{array} \begin{array}{c} \begin{array}{ccc} car & truck & cat \end{array} \\ \left(\begin{array}{ccc} 473 & 29 & 11 \\ 7 & 42 & 2 \\ 5 & 2 & 64 \end{array}\right) \end{array}$$

Table 5: **Confusion Matrix of the Model**

In this model, we can see that the classes success rate is quite balanced, and that we get significantly better predictions compared to the baseline model. That was done by fixing the imbalance using a weighted loss and correct tuning of the hyper-parameters to avoid wrong local minimums with an optimized learning rate. From the confusion matrix we can coclude that the model had mostly "confused" between 'car' and 'truck', and the diagonal shows a good success rate in total.

6 Adverserial Example

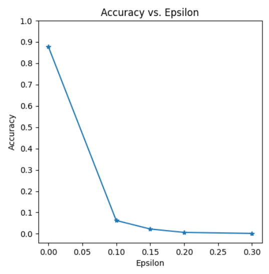
Adversarial example is an input to a machine learning model which has intentional feature perturbations that cause the model to make a false prediction. In my script, it was made by learning the "noise" in the model.

I have created a module that generates adversarial examples from a trained network, called Adversarial_Example.py . The module takes a trained network and input data set (the dev.pickle), and then adjusts inputs to maximize (not minimize!) the loss based on the backpropagated gradients of the trained model. The adversarial example is generated by perturbing the original inputs - adding a "noise" - going up the gradient with a constant rate epsilon. The module also extracts the noise added to each photo. The perturbed image calculation is as follows:

$$perturbed - image = image + \epsilon \cdot sign(\nabla_x J(\theta, x, y))$$

Where image is the original clean image from the data set (also x), and $\nabla_x J$ is the gradient of the loss on the image.

I have evaluated the epsilon rates that creates the greatest perturbation and lowest accuracy when the pertubated images is given to the model to predict (based on all the given data set):



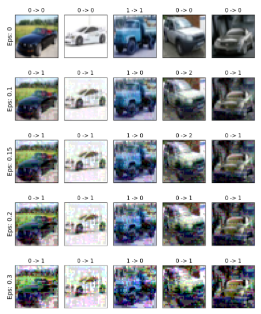


Figure 5: Epsilon anlysis: **Left** - accuracy of the perturbed images predictions using each epsilon. **Right** - 5 Images generated with each ϵ rate, showing its true label on the upper left of the image, and predicted on the right. $\epsilon = 0$ means no changes were made to the image.

We can easily see that the greater the epsilon- the lower the accuracy of the model on the perturbed images generated from the data set. Additionally, we can see that from $\epsilon = 0.1$ the model already makes false predictions (predicts 1 - 'truck', instead of 0 - 'car').

Following this step, I have generated a pertubed image and the noise that was added to the original image, which made the model predict 'truck' instead of 'car':

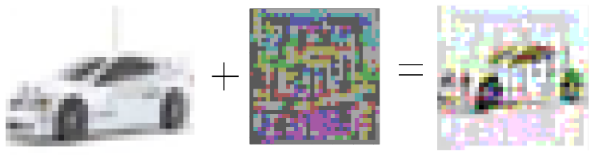


Figure 6: Original Image with the added noise, creates the perturbed image on the right. The image on the left is predictedby the model as 'car' (0), and the image on the right is predicted as 'truck' (1).