# David Pitts

December 9, 2020

Trying to predict the correct class from a data set of 6500~ images , containing cats , trucks and cars. The model we're using is a small neural network with dropout as a regulation tool.

**Starting point:** I ran the network on all the data and checked the performance , the results were 91% accuracy with the following confusion matrix :

$$\begin{bmatrix} 546 & 3 & 2 \\ 30 & 21 & 0 \\ 21 & 1 & 0 \end{bmatrix} \quad \begin{matrix} row\,0 & cars \\ row\,1 & trucks \\ row\,2 & cats \end{matrix}$$

Now using the confusion matrix and going through the data I'll try to find the main problems in the data and try to fix them in order to increase the accuracy

**The** main thing we can notice about the confusion matrix is that the classes are extremely unbalanced , almost 90% of our pictures are cars , so the accuracy being 91% is misleading , since most of our success is coming from the cars department, for example the accuracy ratio over cats is 0%

**Wait** if he have 0% Accuracy over cats , doesn't it mean that he knows how to reject each and every cat and therefor learned the features that define him? No , the confusion matrix signals a classic over fitting of the data toward cars.

**Trucks? why not 0% as well?** Well a car resembles a truck in many of her semantic features , wheels? check , almost rectangle in most cases? check. Also while a truck is much bigger , its hard to compare sizes from a single 2D image sometime , and it may be misleading , so when the network is over fitted toward cars , it make sense that it'll mistake it to a truck in a lot of cases , so the 42% accuracy archived on trucks is much better then a cat , but still not very precise.

**Data:** The first thing that can be seen about the data is that its extremely unbalanced which partly explain the over fitting toward cars , the other thing is that some mislabels exists between cats and cars. I'll start with tackling the balancing problem and move on to the mislabeling afterwards , my attempt at

improving it was done by a weighted sampler , to get a more balanced training data.

**First step toward improvement , using Weight Sampler results :** Using a weighted sampler means we tune the weights in back propagation to each class in the same frequency , which should result in a better results overall. The learning became longer , using 200 epochs and a batch size of 8 resulted in the following matrix :

$$\begin{bmatrix} 517 & 16 & 18 \\ 23 & 26 & 2 \\ 13 & 3 & 7 \end{bmatrix} \begin{matrix} row\,0 & cars \\ row\,1 & trucks \\ row\,2 & cats \end{matrix}$$

While the overall accuracy is worse (88%) , it is a more desired results because the accuracy for each class increased , which means the model over fitted less toward cars , and learned to identify features of more classes instead mode collapsing into a car..

$$Accuracy\,Cat\ :\ \frac{7}{24}$$

$$Accuracy\,Truck\ :\ \frac{26}{53}$$

$$Accuracy\,Car\ :\ \frac{517}{551}$$

So we can see that the accuracy over the cat class increase by 29.16%! and the truck accuracy by 7% , the car accuracy is reduced , but we're fine with that since the results from before weren't an accurate of the true distribution , since it over fitted the data , the mistakes are large and the cat accuracy's are still far from good , but an improvement can be seen and now after handling the mislabeled data problem I hope it'll increase.

**The mislabeled data problem:** Since some of the cats are identified as cars , we have a problem. the model is learning cat features as car features which greatly confuse him. My solution is , if a cat is tagged as a car , the model after enough training will surely predict it as a cat. So the way i filtered out most of the mislabeled cats , in both training and test was as follows :

- Train the model using the original data to distinguish between cats and trucks **only**. Using only the train data and validation data of cats and cars (We can use both here since we only try to find cats in all our data after all and not predict on new data), since the only place the cats are mislabeled are in cars , using the cars data could mislead the model , the truck and cat data is clean , and therefor safe to train and use , a few remarks about the training phase:

  - The training is not consistent , some favors accuracy toward trucks , and some toward cats , I ran the training phase a few times with different hyper parameters and choose the one which gave the highest accuracy at predicting cats and putting less emphasis on the truck accuracy , still the difference wasn't significant being 97% for cats and 92% for trucks over the test data.

- After we have a model which know how to recognize cats , and how to recognize something that resembles a car in enough ways to distinguish it properly ,I used it over the original data set and the test set. Every car that was classified as a cat with high enough confidence ( I regarded the output of the network at position 2 as the confidence level of it being a cat) was saved in an array of indexes. Afterwards i went over all of the manually and filtered out the cars since the vast majority of them were cats , found the cats indexes and remove them from all the mislabeled cats. I repeated the processes a few times , each time with more data to work with , which lead to better classification and results , and sometimes playing with the threshold for the confidence and most (if not all) of the mislabeled cats returned safely into their folder.

  - The values used as confidence thresholds were : [14 , 10 , 6 ] below them most of the mistakes were cars that were really classified as cats since the model isn't accurate enough.

**Training with proper data:** After handling the mislabeling problem I can move on to training it with confidence in my data not tricking my own model. Using batch size of 16 and 200 epochs with a weight samplers resulted in much better accuracy for each class , 97.5% for cars , 86% for cats and 65% for trucks , the training time remained almost the same with no major differences from before.

**Augmentation of the data:** Augmentation of the data is a basic step to make the model generalize better and make him more robust to changes is the pictures , I augmented it with the following filters and attached an explanation why it improves the model:

**Random black squares :** mimics occlusion of objects and making him more robust in that department.

**Hue and saturation :** making the model robust to different lighting \ colors for the same object.

**Flip** - Seeing the same objects when he's mirrored on the x axis , which means we're making the model more robust to the direction the object is facing.

**Rotation** Seeing the same objects facing the same direction but rotating him a certain degree.

So the 5625 pictures we were provided with turned into 40,000~ Images after augmentation. The first thing i noticed is the training period becoming much longer and the loss did not converging to 0.001 as before , instead the loss converges to 0.1 at best with out any dependency on the amount of epochs , which served as a big red flag since the model couldn't over fit the data , which means that he there is a fundamental problem either with the data or the model , which later turned out to be a result of the low capacity of the model.

After further trying to tune the hyper parameters of the model with the augmentation above , he didn't perform as well as a model with out augmentation. The guess behind it is because the model isn't expressive and strong enough to extract meaningful features from such noisy images , therefor an attempt to increase the data set in a smaller capacity then before while doing dynamic augmentation for the needs of each class was brought up to tackle it. .

The dynamic augmentation are as follows :

**Cars Trucks** are both objects that are mostly shown from the side / front and they're rigid objects , they'll stay the same and the only difference is the angle the photo is taken , therefor I choose to augment them with flip and hue & saturation only , to take into account both sides the current rigid object could face , and different lighting , 4 pictures are presented for every car and truck.

**Cats** are not rigid , they could bend in many different ways , therefor a rotation is added on top of hue and saturation , so 8 pictures are presented for every cat , the angles for rotation being (0,90,180,270) , once for the filtered image and once for the original image.

Final data set now contains around 25k~ images.

**Final results basic model:** The results improve dramatically from the first augmentation done first of all, with a nice overall 90.022% accuracy over all 3 classes , partly because of the better augmentation and partly because a better training optimization. , with the following confusion matrix over the validation set :

$$cm = confusion\,matrix = \begin{bmatrix} 460 & 33 & 10 \\ 3 & 44 & 4 \\ 4 & 1 & 66 \end{bmatrix}, \begin{matrix} Cars & 91.5\% \\ Trucks & 86.3\% \\ Cats & 93\% \end{matrix}$$

$$\frac{cm[0,0] + cm[1,1] + cm[2,2]}{3} = TAN = Total\,Accuracy\,Noramzlied = 90.022\%$$

Optimizing the training process was done by trying different learning rates , batch sizes , momentous , but the most integral part of the improvement was understanding the model weakness , for example he learned how to recognize cats quite fast but had a hard time with cars and trucks since they're so similar

, being able to recognize trucks fairly good at the start but when he saw enough examples of cars he over fitted toward cars (97%˜) and gave low accuracy for trucks (65%˜) , i tackled this problem by (The values of the accuracies in the optimizing process are within a margin error of 1% since i forgot to record all of them , beside the final results which is 90.022% ,which i did record.)

1. first introducing to the model a high amount of truck pictures compare to cars with a standard lr of 0.001 and batch size=8.

    - I found that first introducing a normal amount of cars and then trying to increase the amount of trucks inputted to the network at some point didn't really update the weights in a way that improved the TAN , but it worked to some extent in the other way around.

2. when the accuracy was high enough with a loss over the data set that didn't indicate an over fit of the data , and most of the model budget was put into the trucks, (at around 84% normalized accuracy over the validation set and 0.1 loss over test set ) I stopped the training and saved the weights.

    - For example , the model at this point would reach 94%˜ accuracy for trucks and 60%˜ accuracy for cars in this way , and if i decided to prolong the training instead of stopping it , there would be a trade off between the accuracy of cars and trucks for about 30-50 epochs until the cars took most of the accuracy returning to around 96%-97% and trucks would go down to 65%˜ if i didn't stop the training , which indicate a trade off of accuracies.
    - When I saved the accuracy values for cars and truck at each epoch and graphed it , I noticed that the **trade off is not linear** , therefor a sweet spot which gives a sizable improvement to the TAN in their trade-off exists.

3. loaded the model again but this time inputted more cars then trucks and began the training to find the sweet spot , I found the sweet spot before the model over fitted toward cars to gain a few extra points in the trucks accuracy , reaching a TAN of 88%˜ , which was 76%˜ accuracy for trucks and 95%˜ for cars , which was an overall 11% increase for trucks that cost us a 2% decrease in cars , result in in a total of $\frac{11-2}{3} = 3\% \sim$ increase in the TAN.

4. Save the model , and the accuracy for trucks was a bit low (76%˜) , therefor i loaded the model , but this time input vastly more trucks then cars but tune the parameters with an extremely low learning rate = 0.00002(for some reason low momentous work better in this iteration being 0.2) , at this stage it managed to converge after checking a few different hyper parameters for lr and momentous into a TAN of 90.022% over the validation set.

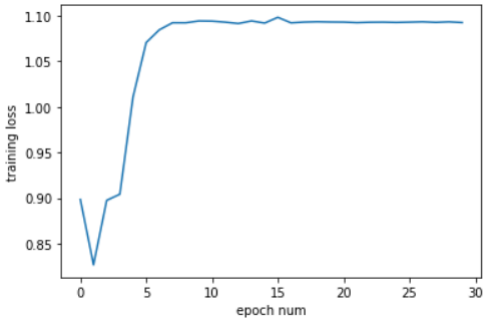
The idea behind reducing the learning rate dramatically (0.00002) at this point is because the high accuracy means the weights are located in a low loss area and jumping out of it will be a shame , so very low learning rate from here on to tune it toward the best minima at the low loss area is needed , otherwise it'll have a decent chance of degrading the accuracy in the second run , since it'll "jump out" of this area.

\

## Learning rate :

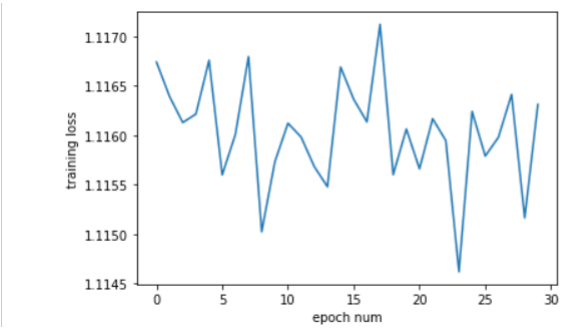
We can see in the following graphs that the learning rates doesn't converge if high or low:

**High learning rate** is bouncing between slopes and will probably never reach any minima , and if by some blind luck he will , he'll just jump out of it in this next iteration , the big steps produced by it make it extremely unstable which ruins his ability to converge. The attached graph describe the case where the training unstable and can't converge because of it.




**Low learning rate** The function of a neural network isn't convex , which means that many local minima could exists in it while we try to optimize it. A small learning rate means that while trying to converge to the global minima (or a minimal point that gives us a desirable results since expecting a neural network to be able to converge to the global minima is optimistic considering the complexity of the function.) almost every small "hole" in the graph (local minima) will be our final minimal for the model , the small learning rate won't allow the weights to skip over it and then it'll stay stuck there. Another option could occur if the network won't get stuck in some local minima where the training will processed as usual but the convergences will take too long for it to be practical.

Using a learning rate of 0.0000001 we get the following graph , which describe the case that the training process is too slow to converge to the optimal global minima in practical time:




## Adversarial example:

An adversarial example for the model means an image of "noise" that we add to a natural picture which tackles the model at it weak points and completely changes him. The way i did it was with the following loss function :

$$l\left(\vec{x}, y_{goal}, x_{target}\right) = \frac{1}{2}\left|\left|y_{goal} - \hat{y}\left(\vec{x}\right)\right|\right|_2^2 + \lambda\left|\left|\vec{x} - x_{target}\right|\right|_2^2$$

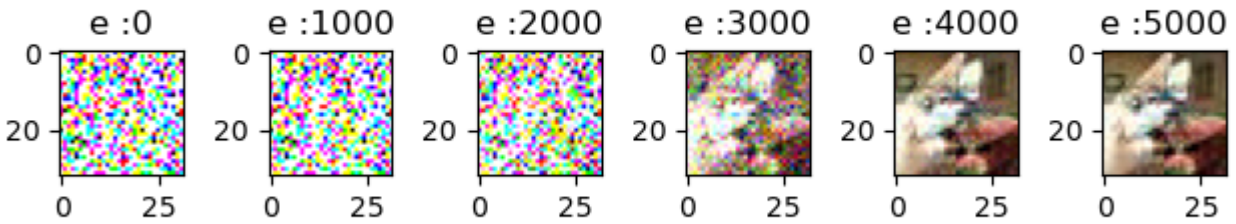
where:

- $\vec{x}$ is the original noise image.
- $x_{target}$ is the picture we want to the adversarial example to look like , therefor the dimension is 1024*3
- $\vec{y_{goal}}$ is the goal tag we want the network to output , which will be $[c, 0, 0]$ $c \in \mathbb{R}^+$
- $\hat{y}\left(\vec{x}\right)$ is the output of the noise image in the model
- $\lambda$ normalizing parameters that decides which term will have more influences.

This loss function works because it is a minimization of with 2 objectives in mind :

1. $\frac{1}{2}\left|\left|y_{goal} - \hat{y}\left(\vec{x}\right)\right|\right|_2^2$ tries to make the prediction of the network over the noise input to be as close as possible to the $y_{goal}$ , which is in my case $\begin{bmatrix} 20 & 0 & 0 \end{bmatrix}$ , so at every iteration the **image is updated with that loss** , which tries to change her intensity levels to get the following output vector from the model..
2. $\lambda\left|\left|\vec{x} - x_{target}\right|\right|_2^2$ , tries to make the noise image look at close as possible to our target image , with some normalization which gives priority over to who we should maximize first.

So our loss function tries to minimize both of those expression , on the one hand trying to make the prediction over the noise as close as possible to $y_{goal}$ and on the other try to make the noise picture look as close as possible to $x_{target}$. this processes converges to a minimal loss after around 5000 times of running the image through the network and updating her . The updating process from noise to final picture can be seen here :

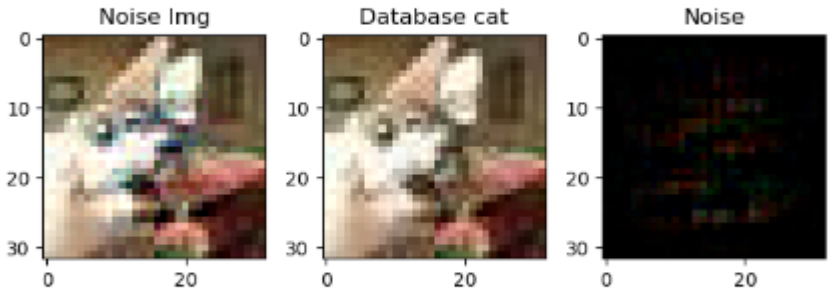



The idea behind it is running the same noise image through a constant network , time and time again and updating the noise image with the loss above and the gradient weights calculated through back propagation , using the following update formula after each iteration :

$$\vec{x} - = \alpha \cdot (derv.X + \lambda \cdot (\vec{x} - x_{target}))$$

- $\alpha$ - learning rate :
- $derv.x$ is the gradient of the noise vector from the loss functions of the network which is defined as above.

I defined my own loss function in PyTorch according to that and used the update rule at every iteration , running the noise pictures through the picture as $\hat{y}(x)$ suggests. I attempted to trick the model into thinking a cat is actually a car and it worked wonderfully , the value for a car outputted by the model is 15.63 and the value of the noise img being a cat is -10.45 , the images can be seen below , the noise makes the picture a bit more blue in the center , which make sense since **cats are never blue** , on the other hand there are enough blue cars in the data set , so adding a lot of blue colors to the center will trick the model into thinking its a car since the convolutions weights gives higher chance for blue pixels toward the car class prediction then the cat. :


Noise Img
Database cat
Noise
0
10
20
30
0
20