ReNeg (Regression with Negative Examples) For Behaviorally Cloning Autonomous Vehicles

Jacob Beck
Brown University RLab, Self-Driving Car (SDC) Lab
In collaboration with Zoe Papakipos
Advised by Professor Michael Littman

I. ABSTRACT

Since reinforcement learning involves making mistakes, it is only viable for autonomous vehicle control in a simulation. Therefore, machine learning for autonomous vehicle control in the real world focuses either on a subset of the problem (i.e. computer vision) or behavioral cloning (i.e. learning from a correct demonstration). We would like to propose a third alternative that will allow for better and faster learning of the entire problem: learning from demonstration that covers the full range of positive to negative behavior. We expect to find that the additional information provided by incorrect behavior (and by gradations in how good or bad a behavior is) will enable the autonomous agent to learn the correct control output (policy) more quickly.

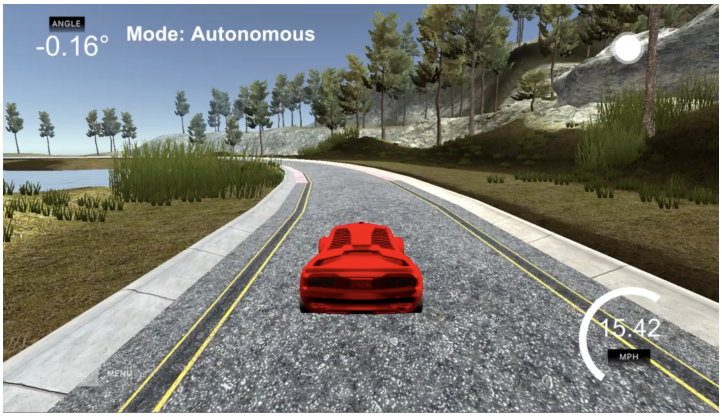

Fig. 1. Self-Driving Car Simulation

II. BACKGROUND

When end-to-end learning works, it has the potential to better optimize all aspects of the pipeline for the task at hand. If the option is to use separate pre-made modules for certain sub-tasks such as computer vision and not optimize them for the given task, or to connect these modules in a way that can be updated for the given task, the latter will always perform better, since the end-to-end model always has the ability to simply not update the sub-module if it will not help training loss. Of course having a black box be entirely responsible for controlling our cars is not a responsible goal, but it can in fact be the right thing to do, either provided that there are some sanity checks to make sure the system is not actively endangering the passengers, or in conditions where no other solution is possible and an optimized black-box is our only hope of preventing imminent danger.

A method for behavioral cloning with end to end learning was put forth by NVIDIA [1], however, to the best of our knowledge, no research so far has focused on using expert-labelled feedback in the context of autonomous vehicle control. (Nor even regression with negative examples in general.) Yet, we believe that this is a major oversight for self-driving cars: given that many research groups are investing exorbitant amounts of time into collecting driving data, if it is truly the case that our hypothesis is correct and that labelling feedback will help the car to learn, such gains can be gotten simply by having an expert labeller sit in the car alongside the driver. For no additionally real world time, advantages can be gained simply by having a "backseat driver". Additionally, for behavioral cloning, in order for the network to know what to do in all sorts of states, the car must get into those states. Often times, getting into those states, requires "bad" actions. Instead of simply throwing out this data, we can use it to augment our data-set.

III. METHODS

The task we will be evaluating our model on is is lane-following in a car simulator implemented in Unity. The correctness of an example will be labelled using feedback from a human. Our goal is to produce a policy network (here on abbreviated as PNet) to map states to actions. In our case, the states will be RGB images from the driver's point of view, and the PNet will have to output a steering angle that keeps the car as close to the center of the road as possible. The crux of the issue we are tackling is finding the correct loss function to take in positive and negative examples with gradations in both. Secondarily, it is important to choose the feedback in an appropriate way as to encourage the correct behavior in the context of lane following. And finally, I will discuss architecture implementation.

A. Loss Function

If θ is the angle in the demonstrated example, $\hat{\theta}$ is the angle predicted by the PNet, and f is the feedback, then the loss function we choose should have the following 3 properties:

1) Minimizing the loss should minimize the distance between θ and $\hat{\theta}$ for positive examples
2) Minimizing the loss should should maximize the distance between θ and $\hat{\theta}$ for negative examples
3) The degree to which minimizing the loss does 1) and 2) should be determined by the magnitude of the feedback, f.

These three properties together will ensure that the network avoids the worst negative examples as much as possible, while seeking the best examples. Given an input state s, the first loss function that comes to mind is what we term "scalar loss":

$$Loss_{scalar} = f * (\theta(s) - \hat{\theta}(s))^2$$

This loss function is notable for several reasons. The first reason that this loss is notable is that it looks a lot like the mean squared loss used for behavioral cloning: $Loss_{clone} = (\theta - \hat{\theta})^2$. In fact, we have set up two hyper-parameters that, when set appropriately, will recover this behavioral cloning loss. The first parameter that we introduced to our loss function was a Boolean setting called THRESHOLD, or THRESH for short. If THRESH is set to true, we simply replace f with $sign(f)$. This will eliminate gradations in positive and negative data and will provide a good metric for comparison. Additionally, we introduced the parameter α. We replace f with $max(\alpha * f, f)$. This has the effect of scaling down all of our negative feedback by α and is useful in isolating the effects of gradations in all data from gradations in just positive data. We apply α after we threshold so that with THRESH set to true and an α value of 0.0, we an can recover behavioral cloning.

The second reason our scalar loss is notable is that it closely resembles the loss that induces a stochastic policy gradient in standard continuous control reinforcement learning. In a standard RL policy network such as REINFORCE, the loss function would be $Loss = R * -log(Pr(\theta))$, which encourages the probability of the action taken by an amount proportional to an unbiased sample of future reward, R [2]. In continuous control, you would instead predict a mean $\hat{\theta}$ for a normal distribution and then sample you action θ from that normal distribution. Now if you replace $log(Pr(\theta))$ with the probability density function for a normal distribution, the loss you wind up with looks a lot like our scalar loss: $Loss = R * (\theta - \hat{\theta})^2$, (ignoring some constants based on the variance of the normal distribution that only affect learning rate).

Although this similarity provides inspiration, it is in fact not justified in this context by reinforcement learning. Since we are "off-policy", the neural network cannot not influence the probability of seeing an example again, and this leads can lead to problems. In RL, the network could try a bad action, and then would move away from it and not revisit it. Whereas, if we have a bad example in our training set for a given state, on every epoch of training, our neural net will encounter this example and take a step away from it, thus pushing our network as far away from it as possible. In fact, even if we have a positive example for that very state, if we have more negative examples than positive examples, if we are not careful, we may wind up ignoring our positive examples completely in an effort to get away from our negative examples.

In fact, this case highlights the trouble inherent in using negative examples. It is hard to know how and when to take into account the negative examples and by how much. For example, in Figure 2, if we perform a regression on positive and negative examples, with more negative examples in one state, we may wind up in a case where our loss is minimized by an output of positive or negative infinity, and our regression is "overwhelmed" by the negative examples. In an effort to avoid this, we considered making a loss function that drops off exponentially with the distance from the negative examples. This led us to our second "exponential" loss:

$$Loss_{exp} = |\theta(s) - \hat{\theta}(s)|^{2f}$$

Using this loss, negative examples will have infinite loss at distance 0, and then drop off exponentially with distance. We hope that this will create regressions more akin to the second image in Figure 2. In this image, adding more negative points will still nudge the regression away more and more, but one positive point not too closeby should be enough to prevent it from converging to positive or negative infinity. It should be noted, that the loss in a particular state still could only have negative examples, especially in a continuous state-space like ours where states are never truly repeated, however the reduction in loss caused by converging towards infinity would be so small that it should not happen simply due to the continuity with nearby states enforced by the structure of the network. In addition, one concern with this loss could be that for positive fractional differences, and negative non-fractional differences, the desired property 3) of loss functions no longer holds. That is, our positive loss will not grow with f it the difference being exponentiated is a fraction. And for negative exponents, the loss will only grow if the fractional denominator is a fraction that shrinks as it is raised to increasing powers of f. However, we hope that for negative examples, distances that are more than 1 unit away will not matter much (since, as discussed later, 1 unit is half the entire range of our output). And, for both positive and negative examples, I will later propose a solution to patch this loss function for future work.

Our final loss function is intended to produce regressions more like the final image in Figure 1. For this, we propose directly modelling the feedback with another neural network (the FNet) for use as a loss function. If this FNet is correctly able to learn to copy how we label data with feedback, it could

be used as a loss function for regression. Thus, in order to maximize feedback, our loss function would be as follows:

$$Loss_{FNet} = -FNet(s, \hat{\theta})$$

One thing to note, is that adding more negative points will not ”push” our regression further away from this point, but rather just make our FNet more confident of the negative feedback there. This may not be the desired effect for all applications. Moreover, the FNet cannot operate on purely positive points with no gradation. That is, behavioral cloning cannot be recovered from it. There may be workarounds that could do this, such as attempting to set a ”default” value for the FNet by feeding in random noise as an input and labelling it negatively, but this would likely have little affect, as the noise will never be similar to an input state. However, for our purposes, it will always function correctly so long as we correctly label our data. At this point, all we would have to do it correctly choose a feedback that, when maximized, produces correct behavior.

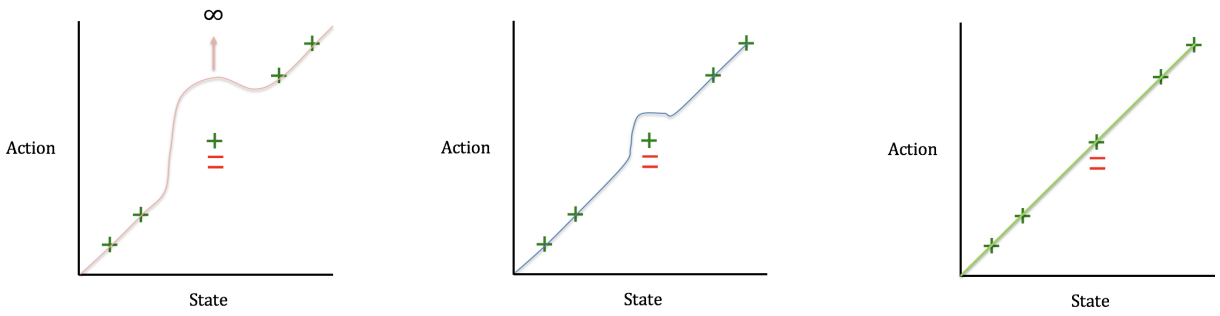

Fig. 2. Potential Outcomes of Regression

B. Feedback

In order to simply the feedback, we chose to force the feedback to be between -1 and 1. The first data we recorded was 20 minutes of optimal driving and we labelled all of this data with a feedback of 1.0. Choosing the rest of the data, and how to label it, was a bit more tricky.

One reason that we are not using reinforcement learning is that letting the car explore actions is dangerous and destructive. In this vein, we wanted to collect data only on ”safe” driving. However, the neural network needs data that will tell it ”bad” actions and also ”good” actions that will recover the car from bad states. In order to accomplish this, we made the goal staying as close to the middle of the road as possible, and labelled any actions that take the car away from the road as negative. Moreover, the speed at which the action takes you away from the center of the road indicates how bad the feedback should be. The opposite is true for positive examples: for positive examples, how fast you return to the center of the road should be how good the feedback is.

In order to explore these types of states and actions, we collected two more types of driving: ”swerving” and ”lane changing”. The first image in Figure 3 bellow is swerving. In swerving, the car was driving in a ”sine wave” pattern on either side of the road. I spent 10 minutes collecting this data on the right side of the road and 10 minutes collecting this data on the left. The second image is lane changing. For this, I drove to the right side of the road, straightened out, staid there, and then returned to the middle. I repeated this for 10 minutes, and then collected 10 minutes on the left-hand side as well.

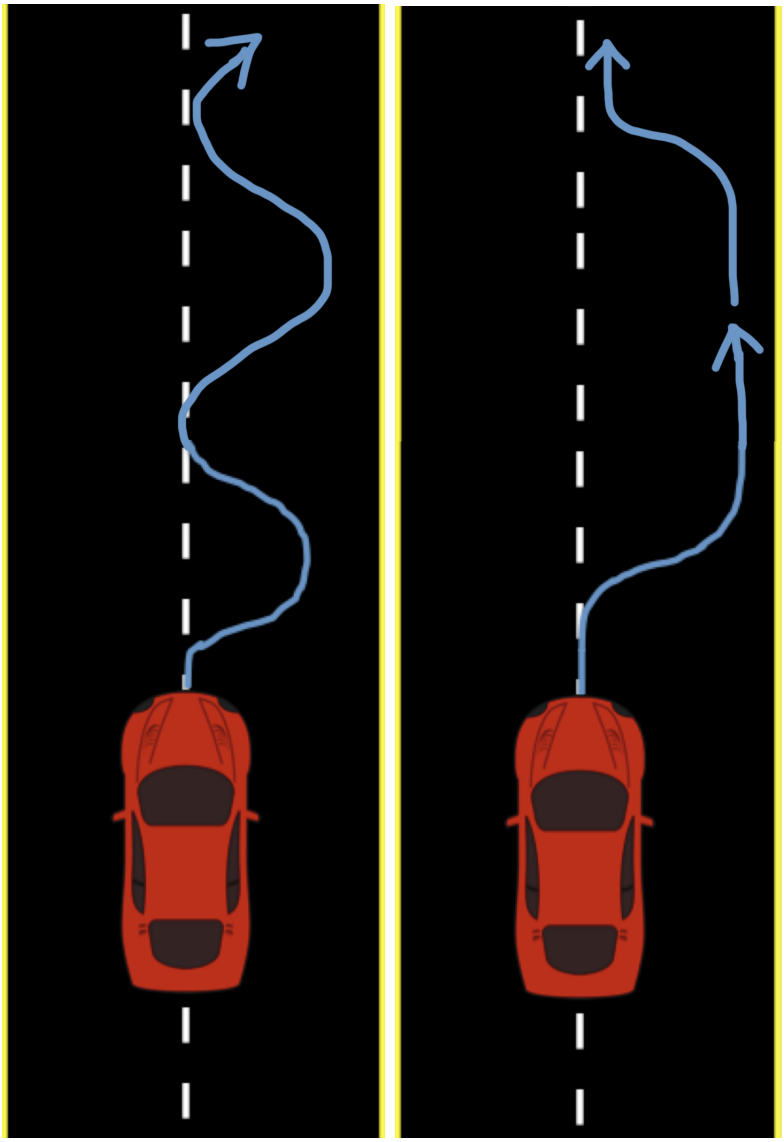

Fig. 3. Types of Driving: Swerving (1st), Lane change (2nd)

At first, we considered labelling all of this data by using a slider that returned values from -1 to 1. However, we realized that this is an issue for two reasons. First, it is very hard to tell when to label the data positively and when to label it negatively, and there are large discontinuity in how you would want the data labelled. For example, it may seem that when swerving away from the road, that this becomes a positive example again when you start heading back towards the center. However, this is not the case: although the state may become ”good” again when your are finally heading back towards the middle of the road, the action that actually made that happen (i.e. steering left), began earlier, since steering affects the car over time. Moreover, since heading off the road when you are already near the edge of the road is worse than doing so when you are near the middle of the road (in that it may cause you to crash sooner), when you suddenly start turning left (the point of inflection on the sine wave), you will need to jump to a positive feedback from a very negative feedback, which is a discontinuity that is hard to capture on a slider. And second, there may be small subtleties that are just difficult for humans to capture given their reaction time. For example, when you lane change to the right edge of the road, in order to straighten out, you start to turn left towards the center, which is

positive, but you then turn right again in order to stay straight. This action is hard to capture with a slider.

In order to circumvent all of these issues, we decided to collect feedback using the steering wheel. (Note, for the sake of consistency, we had one person do all of the driving and the other do all of the collecting.) Our first thought was to just turn the steering wheel to the correct angle. However, this is very difficult to do, especially on turns, when you cannot see the actual effects your steering is having on the car. (Also, if we could do this, we could just use this data for more behavioral cloning!) Therefore, what we decided to do instead was to collect differential steering that is proportionally correct. That is, we turned the wheel based on how much more we wanted the car to go in the direction we were steering. This number did not have to be the exact correct angle, it just had to be true that turning the wheel twice as much meant we wanted it to change twice as much.

In order to actually process this data into a +1 to -1 value, we used the following equation bellow:

FEEDBACK(c, θ)
1 **if** $|c| \geq \theta_{max}$
2 **return** $-\frac{1}{2}$
3 $c = c/\theta_{max}$
4 **if** sign(c) == sign(θ) or $|c| \leq \epsilon$
5 **return** $1 - |c|$
6 **else**
7 **return** $-|c|$

In line 3, we normalize all of our data by diving by a θ_{max} that we pick. (We will return to the case where $|c| \geq \theta_{max}$ later.) So now, by line 4, all of our c values fall between -1 and +1. In line 4, if we are turning the steering wheel in the same direction as the car (with some epsilon of error also being acceptable), then the feedback should be positive. (We set epsilon to $5/\theta_{max}$ so that it allows us to 5 degrees of tolerance.) Remembering that we are recording deltas in steering, a greater delta should result in a less positive signal. Therefore, we subtract c from 1. If we are steering in different directions (line 7), then the feedback should be negative. The greater the delta, the more negative it should be.

One problem with this algorithm occurs on turns. for example, if the car is rounding a left turn, and it is turning left, but not nearly enough, so that it will drive off the road in its current trajectory, we currently have no way to give this a negative value other than by turning the wheel to the right, which is unintuitive. (See Figure 4 bellow.) Therefore, we added lines 1 and 2 to the code. In these lines, if the magnitude of the correction is large enough, we automatically just map the output to -0.5. It is interesting to note that if the car is off to the right side of the road on a left turn, if the car is still going to make the turn, it will get positive feedback (just not as much as if it turned more left). This is in contrast to what happens on a straightaway, where staying off to the right and going straight results in a negative feedback. None of these seem to be an issue, but it is interesting to note the difference in reward on turns nonetheless. It should also be noted, however, that we never had to make use of the code in lines 1 or 2. Therefore, we simply set θ_{max} to an angle just larger than our largest magnitude. We chose it to be close to our largest magnitude because, if we didn't, all of our feedback would be very close to 0 or 1.

Fig. 4. Intial Feedback Issue with Turns

There are three other interesting thing to note with the feedback. One is that our positive or negative feedback may actually never reach 0 and -1 respectively, since we divided by a constant greater than the largest magnitude. However, our α parameter provides an easy way to scale the relative proportions. Second, slow actions back to the center of the road will be rewarded less than quick actions back to the center of the road. However, this is not true for straightening out at the center of the road. Straightening out at the center will require very little correction and so will have a very positive reward. This was an intentional feature to prevent oscillation of the car about the center of the road. Second, although the car does get more feedback in general for going back quickly, since we are sampling at a constant frame rate and driving at a constant speed of 15 mph, the slow actions will effectively be over-sampled. This does not seem to create any issues in practice, but it is something to be aware of.

C. Architecture

We chose to use only an hour of data so that it was not the case all of our loss functions produced a fully capable and indistinguishable self driving car. (In addition to the sake of time and efficiency.) Also, we sampled states at approximately a rate of 12 frames per second, but then down-sampled further to only keeping every 5th frame, for a result of about 2 frames per second. However, in order to not bias our model towards always turning left and also to get more data, we augmented all of our data by flipping it left-right, inverting the angle label, and leaving the feedback the same. After this augmentation, we had 17,918 training images and 3,162 validation images (a 85:15 split).

One method that helped us to bootstrap learning with limited data was transfer learning. We decided to start with a trained image recognition network. We chose Inception-V3 since, among the top winners of the ImageNet competition, it had few parameters (see Figure 5).

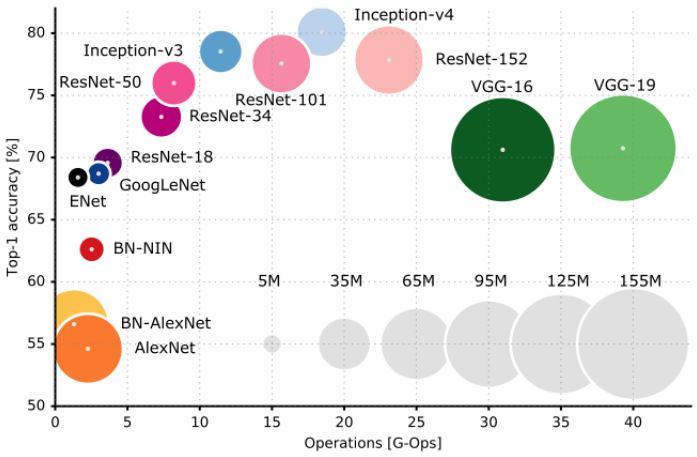

Fig. 5. Inception Parameters vs Others

In order to use Inception Net, we took all of the layers up to some point, and threw out all the layers after that, and then added our own fully connected layers, followed by a tanh activation to ensure that our output was -1 to 1. This -1 to 1 output works both for the PNet and FNet, since our simulator (a modified Udacity simulator coded in Unity) expects a steering value of -1 to 1 and our feedback is also -1 to 1. Note that for our FNet, we also appended 1 additional neuron onto the first dense layer, to be used for the input angle that needs feedback. And our loss for the FNet was mean squared error. We did not add our own convectional layers, since it was easier and better just to allow the gradient to flow through the whole network, changing the weights, and co-opt existing layers that needed to change. It turns out that the ”bottleneck” layer that we attached our new fully-connected layers to was one fairly close to the middle. (See the Figure 6.) We assume that this is the case because it can recognize edges and colors and maybe even some basic features of the road at this point, but has not yet moved on to having neurons that represent only high level features such as ”cats”.

A few hyper-parameters: our fully connected layers had sizes 100, 300, and 20, in order. (101 for our FNet.) Our batch size was 100 and we trained for 5 epochs. Unless otherwise specified for a given model in the results section, we had an α value of 1.0, we did not threshold, and we had a learning rate of 1e-6. As in the Inception model we were using, our input was bilinearly sampled to match the resolution 299x299. Likewise, we subtracted off an assumed mean of 256.0/2.0 and divided by an assumed standard deviation of 256.0/2.0.

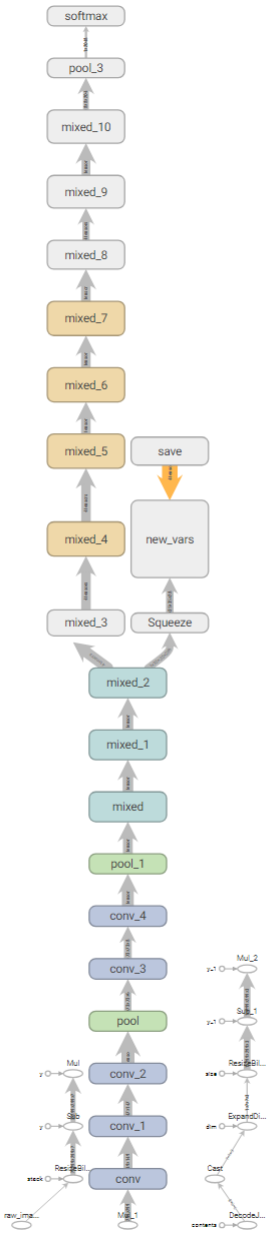

Fig. 6. Our Architecture

IV. RESULTS

A. *Training*

During training, I first tuned our FNet's hyper-parameters. I trained it such that our loss would converge to a low value but not take more than half an hour to train. It happened to be the case that these hyper-parameters worked well for the PNet as well, likely since the losses and architectures are so similar. During training, I kept track of two validation metrics: the loss for the model being trained, and the average absolute error on just the positive data multiplied by 50. The first I refer to as "loss" and the second I refer to as "cloning error" (since it is the 50 times the square root of the cloning error) or just "error". The reason I multiplied by 50 is that this is how Unity converts the -1 to 1

number to a steering angle, so the error is the average angle our model is off by on the positive data. (This is true with the maximum angle set to 50.) These errors are summarized as a tuple of (cloining error, loss), in our plots and reprts.

During training, these two metrics generally behaved very similarly, however, in the models for which I increased the learning rate, you can see that these eventually start to diverge. In this case, the error on the positive data started to increase, but the loss was still decreasing. For this reason, I tried varying the learning rate on several models, to see if the loss was more important than the ”cloning” error. It is clear that the behavioral cloning models ($\alpha = 0.0$ and THRESH) should in general do better on the ”cloning” error, since they are very closely related. Whereas for non-thresholded data, it was trained with examples weighted differently. And for the negative data, it was trained to also get ”away” from negative examples. We hope that even though the cloning error may increase, this means that it is because the model is choosing something BETTER than (yet further away from) the positive examples. We still use the cloning error, however, because it is a useful intuitive metric for training and comparison. In figure 7 bellow, we can see the loss and error diverging for a PNet trained with 10x the normal learning rate.

Fig. 7. Cloning Error vs Loss

Before we discuss how we compared the performance of our models, we should discuss the FNet. Instead of using the FNet as a loss for another neural net, we simply fed in a list of angles and returned the one that our FNet predicted would be best. Since fine granularity in angles is not necessary for this task, by just testing a series of discrete angles, we were better able to see if the FNet would work before committing to an expensive training process. In addition, I would like to mention that, after training, the results of the FNet did not look promising. In fact, what happened is that the FNet learned to predict different feedback for different images, but not for different angles. (See Figure 8 for a random sampling of feedback for given angles rows and images columns.) I suspect that this happened because the state is a much better predictor of feedback. It is likely enough to predict the feedback based on which way the car is facing, rather than which way the car steers. For example, if the car is heading off road,

than pretty much all of those will all have same feedback. The only image of a car driving off the road that will have a different feedback is when the car starts to turn back onto the road. However, this change in angle only lasts a brief amount of time, until the change from the steering actually has an effect and the car has an image state that is not pointing off the road. Only that state where you start to turn back has the same image but a different feedback. For this reason, the FNet's choice was almost never changed, and was almost always the largest or smallest angle possible (depending on stochasticity during training).

```
[[[0.27463108 0.8966497  0.8142399  ... 0.61680347 0.70496565 0.85056734]]
 [[0.27106214 0.8953991  0.8142741  ... 0.6152304  0.70133996 0.8501611 ]]
 [[0.26963258 0.8948949  0.81419414 ... 0.6144671  0.6998794  0.8499993 ]]
 ...
 [[0.25777608 0.8907649  0.8126278  ... 0.6096654  0.68775004 0.84868026]]
 [[0.25635573 0.89021856 0.81237054 ... 0.60916096 0.68617576 0.84817505]]
 [[0.2523774  0.8888414  0.8117536  ... 0.607898   0.68272495 0.8468019 ]]]
```

Fig. 8. FNet Feedback Predictions by Angle (row) and Image (col)

B. Experiments and Benchmarks

We tested our models by running them 8 times in the simulator and recording the time until the car crashed or all 4 tires left the road. In the case that the car drove off the road and came back (which happened very rarely), we discarded that run and started over. It should also be noted that a common failure time was around 0:06 and 1:58. Both of these indicate a failure where the car drove into dirt. There is a patch of dirt near the start that replaces the side of the road, and also a patch of dirt later on that comes right ahead of a sharp turn.

We first tested our PNet with the scalar loss, our PNet with the exponential loss, and our FNet. We plotted their mean times over 8 runs and their standard deviations. See Figure 9. (All of these were on the default settings, except for the exponential loss model, since we could not get many exponential models train correctly.)*

*All of these were on the default settings, except for the exponential loss model. The default exponential model did not converge during training. Instead, we chose to set α to 0.1, which allowed the model to converge and had a loss summarized by the tuple (error=4.978, loss=0.1544). Whereas, the default exponential model was off by an average of 29.03 degrees, with a loss tuple of (29.03, 32.25). Other variation tested included training for 11 epochs (11.24, 0.03817), THRESHOLD (32.316, 32.32), α=0.1 (diverged and not recorded), α=0.1 THRESHOLD (5.876, 0.1669).

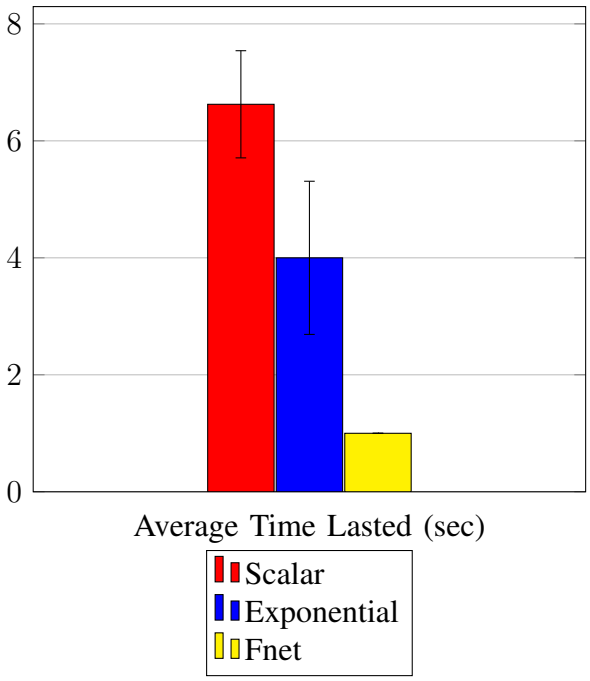

Fig. 9. Initial Comparison of Losses.

Given that the Scalar loss performed best (and was training correctly), we spent more time creating and bench-marking variations of this loss function but with different hyper-parameters. This can be seen in Figure 10. Figure 11 underneath will remind you of the default parameters if not specified.

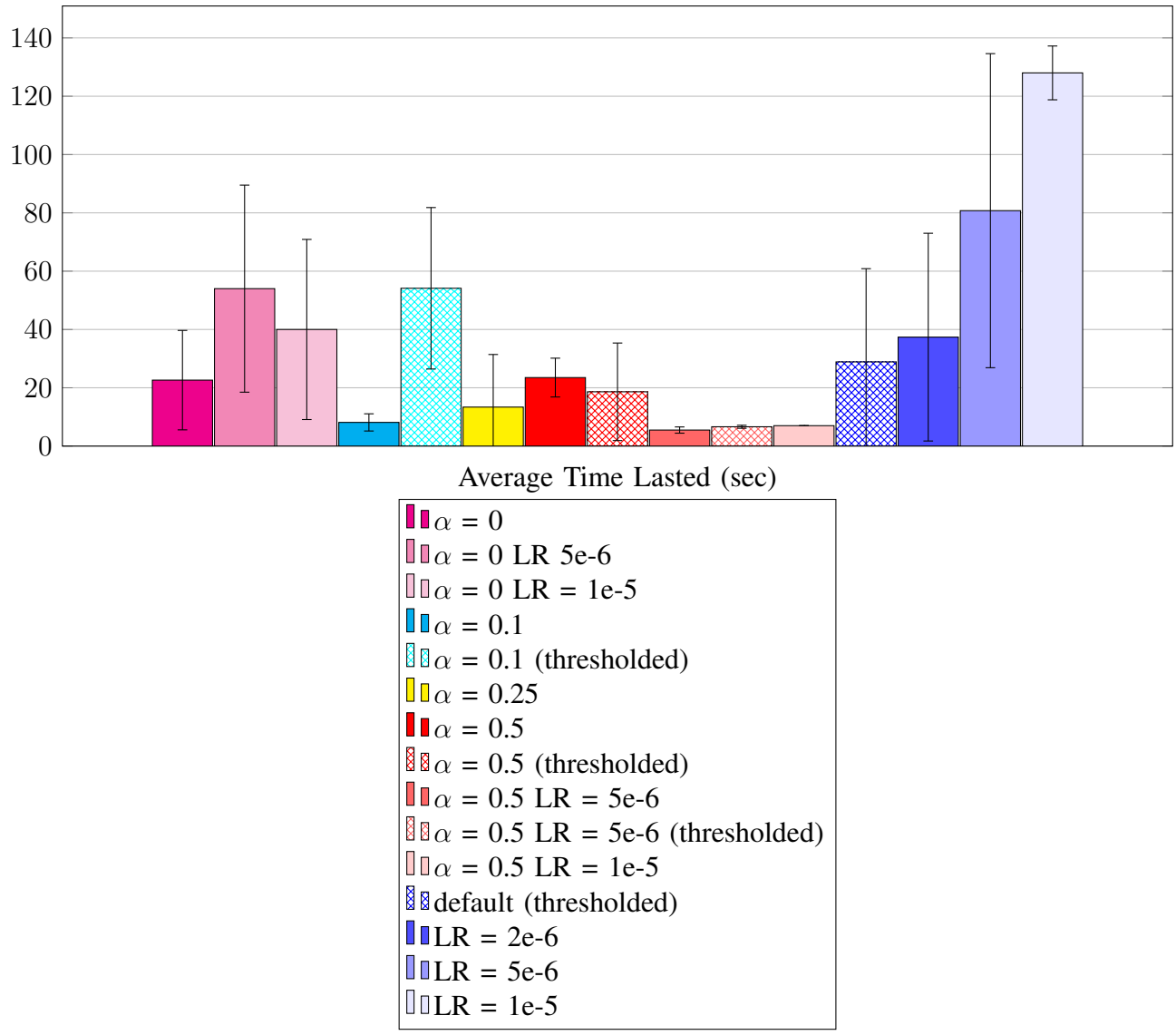

Fig. 10. Scalar Loss Performance by Hyper-Parameter

Name	Default Value	Description
LR	1e-6	Learning rate
Thresholded	False	If True, all feedbacks are thresholded to 1 or -1
α	1.0	How much to scale negative feedbacks down by, in [0, 1]

Fig. 11. The default hyperparameters we used for the scalar PNet.

From this plot we can learn several interesting things. The first interesting thing to note is that with thresholding the feedback to -1 or +1 (the blue crosshatch pattern), the performance came up to about 30 seconds (compared to the 4 seconds from our previous plot). Although at first this could be taken to indicate that having gradations in positive and negative data could be harmful, we can see that the same performance can be achieved by increasing the learning rate instead of thresholding, by looking to the three blue bars to the right. I believe the reason for this is simply that we tuned the learning rate to work well on thresholded data, and so, when we don't threshold or clone our data, the scalar on the loss drops significantly, forcing the network to take smaller steps, and effectively decreasing the learning rate.

The second interesting thing to note is the pink group. In the pink group, all of the α values are 0. In particular, the last two pink bars are identical to the last two blue bars, except for the fact that the alpha value is 0. This indicates that, not only are gradations in the data useful, but also that throwing away the negative data is harmful.

Overall, the best model from these experiments was the default settings but with a learning rate that was 10 times greater. In order to evaluate how this compares to behavioral cloning, we trained new versions of this network 2 more times. Each time, we ran it for 8 runs and calculated the performance as the mean over these 8 runs. We then calculated the mean performance over these 3 training sessions. We compared this to the mean over 3 training sessions of the average performance of a behavioral cloning network. Figure 12 shows the results.

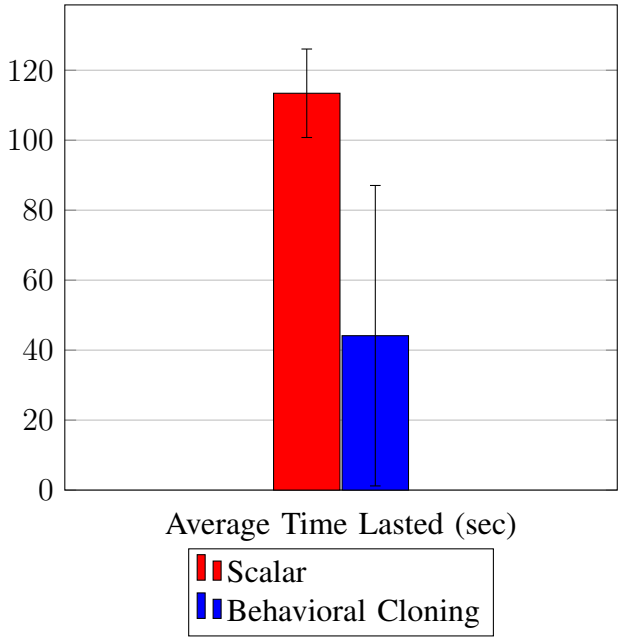

Fig. 12. Our policy net using feedback as a scalar in the loss function worked better than behavioral cloning!

As you can see, our best model trained by a Scalar loss performed over twice as well as the standard behavioral cloning benchmark, with significantly less variance.

V. CONCLUSION

We hypothesized that adding in negative examples would allow our models to have better performance. Moreover, we believed that one of our loss functions would accomplish this task. Given that our scalar model performed over twice as well as the behavioral cloning model, I believe that we have shown this to be true. Our method of regression with negative examples may have wider applications in areas outside of autonomous vehicle control, but at very least, our results indicate that for lane following, with no additional real world time, increased performance can be achieved by simply labelling the data as you collect it.

VI. KNOWN ISSUES

Due to our simulator being down, we tested on a different computer than we trained on. Because of this, our graphics settings were left slightly different at test time as compared to training. That is, we ran in Unity editor instead of the build (with the specific graphics settings), and we left the resolution being sent to the PNet as 320x160 (instead of 640x320). However, the images all are re-sized to 299x299 when they are processed for the neural net, so we really only lost vertical resolution. We expect our results to be the same relative to each other and to be very similar to if we had run at the appropriate settings. Running it now, It seems that our models perform a little bit better, but very close to how they did when we benchmarked. Certainly our best model is still better than our benchmark. However, it may be useful to redo these benchmarks on the correct settings to show whether the neural networks that performed better did so due to greater tolerance of small resolution changes.

Additionally, our car tends to swerve on the road to various degrees. This may be due to a mislabelling of our serving data. Because we labeled our data after the fact, we have the ability to slow down the recording as we label. It should be collected at a slower speed.

VII. FUTURE RESEARCH

Future research should focus on 2 things: the loss function and enforcing continuity.

A. Loss Function

Immediate next steps should likely focus on alterations to the loss function. There are issues with both our scalar loss function and our exponential loss function. Our exponential loss function was created with the intention of making the magnitude of the loss for negative examples drop off exponentially with respect to the distance, since in our scalar loss, the negative examples actually have an exponentially increasing affect as the distance increases. Although the exponential loss accomplishes this solution, there is a dilemma: our exponential solution violates the desired the desired property 3) as mentioned earlier. That is, we want the magnitude of the loss to increase with the magnitude of the feedback, f. However, for positive examples, this only is the case when the absolute difference between θ and $\hat{\theta}$ is greater than 1, and for negative examples, this is only the case when the absolute difference is less than 1. Although this may not be a large concern for the negative examples, since the difference is at most 2, this could be an issue for our positive examples. This issue with the exponential loss function can actually be solved in two different ways. First, I can think of an elegant solution that will modify our scalar loss to have this exponential decay property. Second, I an think of a way to modify our scalar loss to have neither an exponential increase with distance nor an exponential increase. And third, I can think of an "ugly" patch for our exponential function, if we really must have the loss be exponential in f.

1. We can accomplish this exponential decay by modifying our scalar function in a very easy way: Move the sign of f into the exponent:

$$Loss_{scalar} = |f| * (\theta(s) - \hat{\theta}(s))^{2*sign(f)}$$

Using this loss function we have all three properties satisfied. That is, positive examples encourage moving towards them, negative examples encourage moving away from them, and the amount of this movement increases with the magnitude of f. Moreover, we also have the properly that, in negative examples, loss drops of exponentially with the distance from the negative example (because we are dividing by it).

2. If we want our scalar loss function to have neither an exponential decay nor an exponential increase with the distance from the negative points, we can simply use the following loss:

$$Loss_{scalar} = f * |\theta(s) - \hat{\theta}(s)|$$

This has the not-so-nice property that, in the positive example, it allows outliers much more easily than the traditional squared loss. However, it has the very nice property that, given a single state input, as long as you have more positive examples than negative examples, your loss will always be minimized in that state by a value between your positive examples. This is because, as soon as you get to your greatest or least positive example, every step away from your positive examples will cost you 1 loss, for each positive example you have, and you will only lose 1 loss for each negative example you have. (Note, if you are not thresholding, then this translates to more total $|f|$ for positive examples than negative examples.)

3. If we really want our feedback to be in the exponent, we can accomplish this with a more complex solution. We can split our examples into positive and negative examples and have a loss for each. The positive and negative loss functions could respectively be:

$$Loss_{exp,positive} = min(|\theta(s) - \hat{\theta}(s)|/tol, 1)^{2f}$$

$$Loss_{exp,negative} = (|\theta(s) - \hat{\theta}(s)|/2)^{2f}$$

For the positive examples, the magnitude of the loss increases with f in the case that the number being raised to the $2f$ is greater than 1. If $|\theta(s) - \hat{\theta}(s)|$ is greater than tol, we have both ensured this is the case and we are back to our original loss function, but scaled by a constant factor of tol. But what happens if the difference is less than tol? In that case, for any estimate $\hat{\theta}$ within tol of the intended $theta$, our parameters will have a derivative of 0 (no effect), due to the min. (If we want this to be in degrees, we can just set adjust this to tol/=50). (This will cause the neural network to forget about examples that it does well on until they become sufficiently problematic again. This may be helpful in that the network can focus on the area it needs to, or it could be harmful in that is prevents convergence.) For the negative examples, the magnitude of the loss will only increase with f if the number being raised to the $2f$ is less than 1, because we are dividing my it. To ensure this is the case, we can just divide by our range, which is 2.0. To combine these two functions, we can just pick the best convex combination of them.

B. Continuity

Because, in both our scalar and exponential loss, our loss function at a given state with just a negative example is minimized by moving away from the negative example, our regression in that state will tend toward positive or negative infinity. Certainly having a cost on negative examples that drops of exponentially will help, but it may not be enough. Moreover, we may not want to rely on the structure of neural networks to discourage this discontinuity. Therefore, research could be done on adding a regularization term to the loss that penalizes discontinuity. That is, we would add some small loss based on how dissimilar the answers for nearby states are. Of course, this implies a distance metric over states, but using consecutive frames may suffice.

REFERENCES

[1] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," *CoRR*, vol. abs/1604.07316, 2016.

[2] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine Learning*, vol. 8, pp. 229–256, May 1992.