Supplemental Material for DL2: Training and Querying Neural Networks with Logic

Here, we describe implementation details and hyperparameters for our experiments. In all experiments, we used $\xi = 1$, but we found the value not to have a large impact.

A. Details for Training Experiments

Supervised Learning For our experiments with supervised learning, we used a batch size of 128 and the Adam optimizer with learning rate 0.0001. All experiments were run for 200 epochs and the reported values in Table 3 are for the model with the highest constraint accuracy times prediction accuracy within the last 25 epochs. The average time for one epoch is given in Table 4. All other parameters are listed in Table 5. Additionally, for the CIFAR-10 experiments, we used data augmentation with random cropping and random horizontal flipping. Experiments with the Segment constraints were done by first embedding images in 40-dimensional space using PCA. In this lower dimensional space, it is sensible to consider linear interpolation between images which is not the case otherwise. This experiment was not performed for CIFAR-10 because we did not observe good prediction accuracy with baseline model using lower dimensional embedding. This is likely because the dimensionality of CIFAR-10's images is much higher than that of MNIST or FASHION.

We used ResNet-18 (He et al., 2016) for the experiments on CIFAR-10 and convolutional neural network (CNN) with 6 convolutional and 2 linear layers for MNIST and FASHION (trained with batchnorm after each convolutional layer). The layer dimensions of the CNN are (1, 32, 5x5) - (32, 32, 5x5) - (32, 64, 3x3) - (64, 64, 3x3) - (64, 128, 3x3) - (128, 128, 1x1) - 100 - 10 where (in, out, kernel-size) denotes a convolutional layer and a number n denotes a linear layer with n neurons.

We have implemented the Segment$^{\mathrm{G}}$ constraint as

$$\begin{aligned}\forall \boldsymbol{z}.\ ((\boldsymbol{z} = \lambda \cdot \boldsymbol{x} + (1-\lambda)\cdot \boldsymbol{x}')\\ \wedge\ (\|\boldsymbol{x} - \boldsymbol{x}'\|_2 < \epsilon)\\ \wedge\ (y \neq y'))\implies\\ H(p_\theta(\boldsymbol{z}), \lambda \cdot p_\theta(\boldsymbol{x}) + (1-\lambda)\cdot p_\theta(\boldsymbol{x}')) < \delta.\end{aligned}$$

Which in addition to the constraint in the paper, only considers training samples that are close together ($\|\boldsymbol{x} - \boldsymbol{x}'\|_2 < \epsilon$) and that have different labels ($y \neq y'$).

For the C-Similarity constraint we use the formulation as in the body of the paper, but we not only apply it to the car-truck-dog labels, but also deer-horse-ship, plane-ship-frog, dog-cat-truck and cat-dog-car.

Semi-supervised Learning For this experiment, we use the VGG-16 architecture (Simonyan & Zisserman, 2014) and optimize the loss using Adam with learning rate 0.001 and a batch size of 128. For each labeled batch, we also sampled one unlabeled batch and combined the losses before back-propagating. We trained for 1600 epochs, although stable results were observed after around 400 epochs. We used $\lambda = 0.6$ as the weighting factor for the DL2 loss.

Unsupervised Learning Our model is the multilayer perceptron with $N \cdot N$ input neurons, three hidden layers with 1000 neurons each and an output layer of N neurons. N is the number of vertices in the graph, in our case 15. The input is the adjacency matrix of the graph and the output is the distance for each node. The network uses ReLU activations and dropout of 0.3 after each hidden layer. The network is optimized using Adam with learning rate 0.0001.

B. Additional Details for Section 6.2

Here, we provide statistics on the networks and queries used in the experiments of Section 6.2. Table 6 summarizes the networks that we used, their architecture, and accuracy. Each row shows the dataset, the type of the network (classifier, generator, or discriminator), the network signature, and the architecture of the network. For example, the first row describes a classifier that takes as input images of size 28×28 pixels, each ranging between 0–1, and returns a probability distribution over ten classes.

Figure 4 shows the query templates, where the template parts are colored in blue. In the queries, we use the following names for the template parts. Networks are named `N`, $\texttt{N}_1$, $\texttt{N}_2$ for classifiers, `G` for generator, and `D` for discriminator. A variable denoting an input image (e.g., `deer` in Figure 1a) is `var`. Classes are `c`, $\texttt{c}_1$, $\texttt{c}_2$ or $\texttt{c}_v$ to refer to the (known) class of a variable `var`. Masks are de-

Table 4. Average time per epoch in seconds for the supervised training experiments in Table 3.

	MNIST		FASHION		CIFAR-10	
	Baseline	DL2	Baseline	DL2	Baseline	DL2
Robustness^{T}	7.34	13.11	7.24	12.58	40.04	100.97
Robustness^{G}	14.23	866.19	13.9	533.41	40.78	418.62
Lipschitz^{T}	7.37	12.45	7.53	12.13	40.63	75.72
Lipschitz^{G}	12.2	67.79	7.58	401.4	40.02	287.64
C-similarity^{T}	-	-	-	-	40.13	83.85
C-similarity^{G}	-	-	-	-	39.87	423.82
Segment^{G}	15.16	101.55	13.5	30.62	-	-

Table 5. Hyperparameters used for supervised learning experiment

	MNIST, FASHION			CIFAR-10		
	λ	PGD Iterations	Params	λ	PGD Iterations	Params
Robustness^{T}	0.2	-	$\epsilon_1 = 7.8, \epsilon_2 = 2.9$	0.04	-	$\epsilon_1 = 13.8, \epsilon_2 = 0.9$
Robustness^{G}	0.2	50	$\epsilon = 0.3, \delta = 0.52$	0.1	7	$\epsilon = 0.3, \delta = 0.52$
Lipschitz^{T}	0.1	-	$L = 1.0$	0.1	-	$L = 1.0$
Lipschitz^{G}	0.2	50	$L_{\text{mnist}} = 0.1, L_{\text{fashion}} = 0.3, \epsilon = 0.3$	0.1	5	$L = 1.0, \epsilon = 0.3$
Classes^{T}	-	-	-	0.2	-	$\delta = 0.01$
Classes^{G}	-	-	-	0.2	10	$\delta = 1.0, \epsilon = 0.3$
Segment^{G}	0.01	5	$\epsilon = 100, \delta = 1.5$	-	-	-

noted by `mask` and their complementary mask by `nm`. Pixel constraints (e.g., `i in [0,255]`) are `pix_con`, and their range (e.g., `[0,255]`) is `range`. The shape of a variable (e.g., 28, 28 for MNIST) is `shape`, and a constant denoting a distance is `dist`.

The tested queries enable us to study various aspects of the DL2 system. The first query runs a neural network on a given input, which allows us to gauge the overheads of our system (without the optimizer). This includes parsing, query setup time and the call to PyTorch to run the neural network. Queries 2-4 look for inputs satisfying constraints without an **init** clause, while queries 5-11 look for adversarial examples. Queries 12-18 aim to find inputs classified differently by the two networks, whereas queries 14-15 leverage generators and discriminators for this task. Table 7 shows the average run-time and success rate of different queries.

C. Additional Query Experiments

Here, we provide further experiments to investigate the scalability and run-time behavior of DL2. For all experiments, we used the same hyperparameters as in Section 6.2.

Experiment I: Number of Variables We first study the run-time behavior of DL2 as a function of the number of variables. For this, we consider a simple toy query:

```
find i[c]
where 1000 < sum(i), sum(i) < 1001
return i
```

with different integer values for c. We execute this query for a wide range of c values – 10 times for each value – and we report the average run-time in Figure 5a. All runs completed successfully and returned a correct solution. We observe constant run-time behavior for up to 2^{13} variables, and a linear run-time in the number of variables afterwards.

Experiment II: Opposing Constrains We next study the impact of (almost) opposing constraints. For this, we consider another toy query:

```
find i[1]
where i[0] < −c ∨ c < i[0]
return i
```

for an integer c. This query requires optimizing two opposing terms until one of them is fulfilled. The larger c, the more opposed the two objectives are – in the extreme case, for $c \to \infty$, we obtain an unsatisfiable objective. All runs completed successfully and returned a correct solution. Figure 5b shows the average run-time over 10 runs for different values of c.

Experiment III: Scaling to Many Constraints Lastly, we study the scaling of DL2 as a function of the number of constraints. For this, we consider the following query which looks for an adversarial example:

Table 6. The datasets and networks used to evaluate DL2. The reported accuracy is top-1 accuracy and it was either computed by the TorchVision library ($^{\#}$), or by us (†).

Dataset	Type	Network	Architecture	Accuracy
MNIST	C	M_NN1: $[0,1]^{28\times28} \mapsto [0,1]^{10}$	small CNN, PyTorch Examples	$0.990^{\dagger}$
	C	M_NN2: $[0,1]^{28\times28} \mapsto [0,1]^{10}$	small FFNN, PyTorch Examples	$0.979^{\dagger}$
	G	M_G: $[-1,1]^{100} \mapsto [0,1]^{28\times28}$	DC-GAN	-
	D	M_D: $[0,1]^{28\times28} \mapsto [0,1]$	DC-GAN	-
	G	M_ACGAN_G: $[-1,1]^{100} \times \{0,\dots,9\} \mapsto [0,1]^{28\times28}$	AC-GAN	-
	D	M_ACGAN_D: $[0,1]^{28\times28} \mapsto [0,1] \times [0,1]^{10}$	AC-GAN	-
Fashion MNIST	C	FM_NN1 : $[0,1]^{28\times28} \mapsto [0,1]^{10}$	same as M_NN1	$0.876^{\dagger}$
	C	FM_NN2 : $[0,1]^{28\times28} \mapsto [0,1]^{10}$	same as M_NN2	$0.860^{\dagger}$
	G	FM_G: $[-1,1]^{100} \mapsto [0,1]^{28\times28}$	DC-GAN	-
	D	FM_D : $[0,1]^{28\times28} \mapsto [0,1]$	DC-GAN	-
CIFAR	C	C_NN1 : $[0,1]^{32\times32\times3} \mapsto [0,1]^{10}$	VGG-16	$0.936^{\dagger}$
	C	C_NN2 : $[0,1]^{32\times32\times3} \mapsto [0,1]^{10}$	Resnet-18	$0.950^{\dagger}$
	G	C_G : $[-1,1]^{100} \mapsto [0,1]^{32\times32\times3}$	DC-GAN	-
	D	C_D : $[0,1]^{32\times32\times3} \mapsto [0,1]$	DC-GAN	-
GTSRB	C	G_NN1 : $[0,1]^{32\times32\times3} \mapsto [0,1]^{10}$	VGG-16	$0.985^{\dagger}$
	C	G_NN2 : $[0,1]^{32\times32\times3} \mapsto [0,1]^{10}$	Resnet-18	$0.995^{\dagger}$
	G	G_G : $[-1,1]^{100} \mapsto [0,1]^{32\times32\times3}$	DC-GAN	-
	D	G_D : $[0,1]^{32\times32\times3} \mapsto [0,1]$	DC-GAN	-
ImageNet	C	I_V16 : $[0,1]^{224\times224\times3} \mapsto [0,1]^{1000}$	VGG-16 from PytTorch Vision	$0.716^{\#}$
	C	I_V19 : $[0,1]^{224\times224\times3} \mapsto [0,1]^{1000}$	VGG-19 from PytTorch Vision	$0.7248^{\#}$
	C	I_R50 : $[0,1]^{224\times224\times3} \mapsto [0,1]^{1000}$	ResNet-50 from PytTorch Vision	$0.764^{\#}$

```
find p[28, 28]
where class(M_NN1(clamp(p + M_nine,
                        0, 1))) = c
return clamp(p + M_nine, 0, 1)
```

The query looks for an adversarial perturbation p to a given image of a nine (`M_nine`) such that the resulting image gets classifies as class c. The query returns the found perturbation and the resulting image. The `clamp(I, a, b)` operation takes an input `I` and cuts its values such that they are between a and b.

Additionally, we impose constraints on the rows and columns of the image. For a row i, we want to enforce that the values of the perturbation vector are increasing from left to right:

```
p[i, 0] < p[i, 1],
p[i, 1] < p[i, 2],
p[i, 2] < p[i, 3], ...
```

For one row this yields 27 constraints. We further consider a similar constraint for a column j:

```
p[0, j] < p[1, j],
p[1, j] < p[2, j],
p[2, j] < p[3, j] ...
```

We apply these constraints on the first k rows and columns of the image independently and jointly. We vary the value of k, and for each we execute the query over all possible target classes $c \in \{0,\dots,8\}$. We report the average time in Table 8. Most queries could be solved. For **Row & Column constraints** where $k = 20$ and $k = 28$, only 6 out of 9 and 1 out of 9 could be solved receptively. These queries hit the 300 s timeout. Figure 6 shows a resulting image.

	Query
1	**eval** N(var)
2	**find** i[shape] **where** **c**(N(i))=c
3	**find** i[shape] **where** **c**(N(i))=c, pix_con
4	**find** i[shape] **where** **c**(N(i))=c, N(i).p[c] > 0.8, pix_con
5	**find** i[shape] **where** **c**(N(i))=c **init** i=var
6	**find** i[shape] **where** **c**(N(i))=c, pix_con, $\|$i - var$\|_\infty$ < dist **init** i=var
7	**find** i[shape] **where** **c**(N(i))=c, pix_con **init** i=var
8	**find** i[shape] **where** **c**(N(i))=c, i[mask] **in** range, i[nm]=var[nm] **init** i=var
9	**find** i[shape] **where** **c**(N(i))=c, pix_con, N(i).p[c] > 0.8 **init** i=var

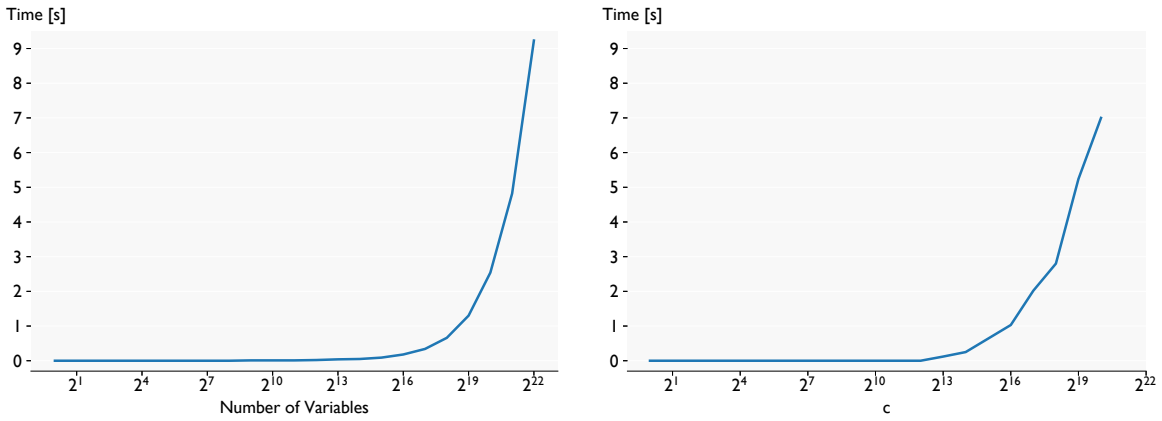
	Query
10	**find** i[shape] **where** **c**(N(i))=c, pix_con, N(i).p[c] > 0.8, N(i).p[c_v] < 0.1 **init** i=var
11	**find** i[shape] **where** **c**(N(i))=c, pix_con, N(i).p[c] > 0.8, N(i).p[c_v] < 0.1, $\|i-$var$\|_\infty$ < dist **init** i=var
12	**find** i[shape] **where** pix_con, **c**(N_2(i))=c_2, **c**(N_1(i))=c_1
13	**find** i[shape] **where** pix_con, **c**(N_2(i))=c, $\|$i - var$\|_\infty$ < dist, **c**(N_1(i))=c_v, **init** i=var
14	**find** i[shape] **where** **c**(N_1(i))=c_1, **c**(N_2(i))=c_2, N_1(i).p[c_1] > 0.5, N_2(i).p[c_1] < 0.1, N_2(i).p[c_2] > 0.5, N_1(i).p[c_2] < 0.1, pix_con,D(i) < 0.1

	Query
15	**find** i[100] **where** i **in** [-1,1], **c**(N_1(G(i)))=c_1, N_1(G(i)).p[c_1]> 0.3, **c**(N_2(G(i)))=c_2, N_2(G(i)).p[c_2]> 0.3
16	**find** i[shape] **where** **c**(N_1(i))=c_v, **c**(N_2(i))=c, i[mask] **in** range, i[nm]=var[nm] **init** i=var
17	**find** i[shape] **where** **c**(N_1(i))=c_1, **c**(N_2(i))=c_2, N_1(i).p[c_1] > 0.5, N_1(i).p[c_2] < 0.1, pix_con
18	**find** i[shape] **where** **c**(N_1(i))=c_1, **c**(N_2(i))=c_2, N_1(i).p[c_1] > 0.6, N_1(i).p[c_2] < 0.1, N_2(i).p[c_2] > 0.6, N_2(i).p[c_1] < 0.1, pix_con

Figure 4. The template queries used to evaluate DL2. N denotes a classifier neural network, var an example input from the corresponding dataset, c, c_1, c_2 are classes and $c_1 \neq c_2$, pix_con is a box constraint for pixels to be in the proper input range, D is a discriminator neural network and G is a generator neural network.

Table 7. Results for queries: (#✓) number of completed instances (out of 10), 🕐 is the average running time in seconds, and 🕐✓ the average running time of successful runs (in seconds).

	MNIST			FASHION			CIFAR-10			GTSRB			ImageNet		
Nr.	#✓	🕐	🕐✓	#✓	🕐	🕐✓	#✓	🕐	🕐✓	#✓	🕐	🕐✓	#✓	🕐	🕐✓
1	10	0.03	0.03	10	0.03	0.03	10	0.03	0.03	10	0.03	0.03	10	0.04	0.04
2	10	0.16	0.16	10	0.15	0.15	10	0.39	0.39	10	0.30	0.30	10	5.71	5.71
3	10	0.21	0.21	10	0.19	0.19	10	0.43	0.43	10	0.54	0.54	10	9.58	9.58
4	10	0.31	0.31	10	0.21	0.21	10	0.61	0.61	10	0.69	0.69	10	18.81	18.81
5	10	0.15	0.15	10	0.15	0.15	10	0.19	0.19	10	0.53	0.53	10	2.95	2.95
6	10	6.38	6.38	10	1.76	1.76	10	6.59	6.59	9	40.94	32.15	1	109.20	12.01
7	10	0.18	0.18	10	0.18	0.18	10	0.33	0.33	10	0.54	0.54	10	8.30	8.30
8	10	1.39	1.39	10	0.24	0.24	10	0.34	0.34	9	12.52	0.57	10	11.83	11.83
9	10	0.20	0.20	10	0.21	0.21	10	0.31	0.31	10	0.96	0.96	10	8.75	8.75
10	10	0.22	0.22	10	0.22	0.22	10	0.41	0.41	9	12.71	0.79	10	10.21	10.21
11	9	13.57	1.74	10	6.32	6.32	10	5.58	5.58	9	33.18	23.53	1	109.22	12.17
12	10	0.38	0.38	10	0.34	0.34	10	0.96	0.96	10	0.80	0.80	10	20.52	20.52
13	10	7.23	7.23	10	2.33	2.33	10	7.04	7.04	8	55.68	39.61	0	120.00	0.00
14	10	0.53	0.53	10	0.53	0.53	10	1.92	1.92	10	1.93	1.93	-	-	-
15	10	3.17	3.17	8	34.53	13.16	10	11.15	11.15	4	91.90	49.74	-	-	-
16	9	13.95	2.16	10	0.52	0.52	10	1.33	1.33	10	1.21	1.21	10	18.39	18.39
17	10	0.63	0.63	10	0.35	0.35	10	1.14	1.14	10	1.67	1.67	9	47.43	39.37
18	10	0.71	0.71	10	0.37	0.37	10	1.27	1.27	10	1.24	1.24	10	33.69	33.69



(a) Run-time for experiment 1. Runs up to 2^{13} variables take less than 0.01 s and don't show increase with number of variables.

(b) Run-time for experiment 2. All runs up to $c = 2^{12}$ take less than 0.01 s.

Figure 5. Experimental results for Experiments 1 and 2. Results are averaged over 10 runs with different random seed. All runs succeed.

Table 8. Run-times for additional constraints on adversarial perturbation. #✓is the number of successful runs out of 9 and 🕒✓is the average run-time over the successful runs in seconds. k row or column constraints corresponds to 27 individual constraints in DL2 each. Thus, the right most column adds 756 constraints for the first two settings and 1512 for the last.

Row constraints

k	0	1	3	5	10	20	28
🕒✓ [s]	0.02	1.24	3.25	6.36	24.82	67.25	101.49
#✓	9	9	9	9	9	9	9

Column constraints

k	0	1	3	5	10	20	28
🕒✓ [s]	0.02	1.00	3.22	5.83	21.44	71.87	270.83
#✓	9	9	9	9	9	9	9

Row & Column constraints

k	0	1	3	5	10	20	28
🕒✓ [s]	0.02	1.89	6.14	11.06	89.02	213.30	270.83
#✓	9	9	9	9	9	6	1

(a) The found perturbation p, scaled such that -0.3 corresponds to black and 0.3 to white.

(b) The resulting image.

Figure 6. Found results for the full 28 row & column constraints and target class 6.