Appendices

A. GroupSort Activation

In this section, we provide visualizations to shed light on how GroupSort networks compute simple 1D functions, explain how GroupSort compares with other activations, analyze the effect of the grouping size on its expressivity and discuss its computational complexity of GroupSort.

A.1. Visualizing GroupSort Networks

In Figures 10 and 11, we visualize the hidden layer activations of GroupSort networks as the input to the network is varied. The networks are approximating the absolute value function and a curve resembling the letter ”W”, with a slope of 1 almost everywhere.

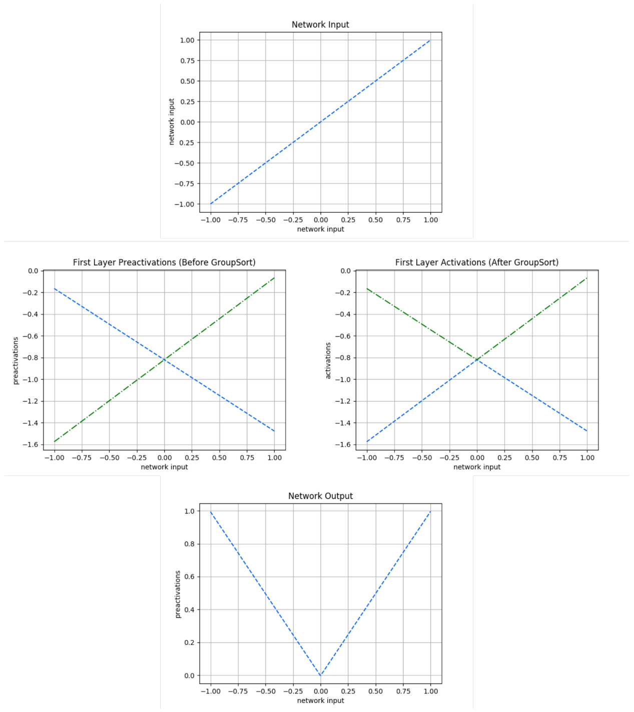


Figure 10: Visualization of the pre-activations and activations of a one hidden layer GroupSort network that is approximating the absolute value function. The network has two units in its hidden layer.

A.2. GroupSort and other activations

Here we show that GroupSort can recover ReLU, Leaky ReLU, concatenated ReLU, and maxout activation functions. We first show that MaxMin can recover ReLU and its variants. Note that,

$$\mathbf{MaxMin}(\begin{bmatrix} x \\ 0 \end{bmatrix}) = \begin{bmatrix} ReLU(x) \\ -ReLU(-x) \end{bmatrix} \tag{5}$$

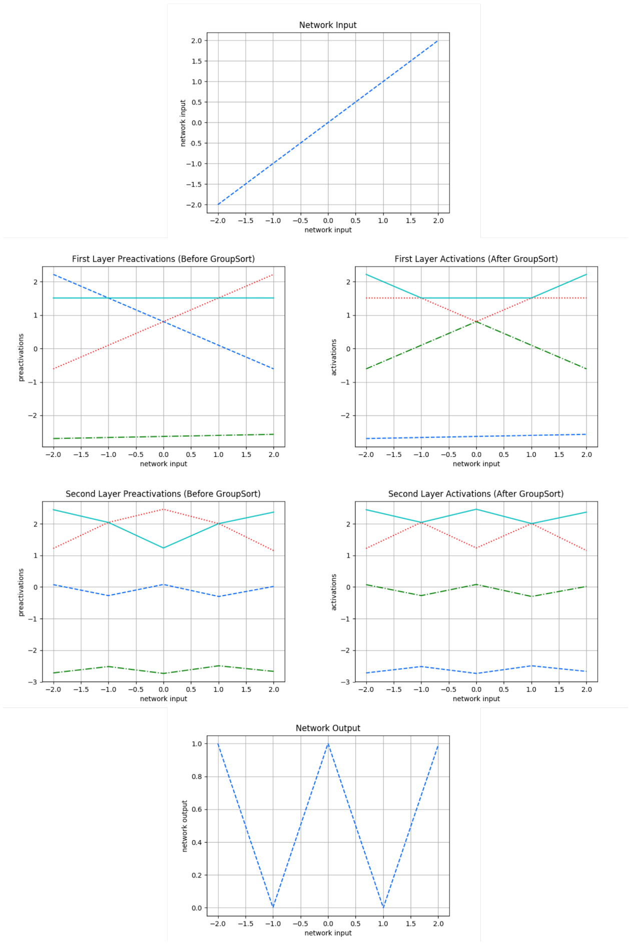


Figure 11: Visualization of the pre-activations and activations of a two hidden layer GroupSort network that is approximating a curve resembling the letter ”W” with a slope of 1 almost everywhere. The network has four units in its hidden layers.

By inserting 0 elements into the pre-activations and then applying another linear transformation after MaxMin we can output either ReLU or concatenated ReLU. Explicitly,

$$[1 \quad 0]\,\mathbf{MaxMin}(\begin{bmatrix} x \\ 0 \end{bmatrix}) = ReLU(x) \tag{6}$$

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}\mathbf{MaxMin}(\begin{bmatrix} x \\ 0 \end{bmatrix}) = \begin{bmatrix} ReLU(x) \\ ReLU(-x) \end{bmatrix} \tag{7}$$

If instead of adding 0 to the preactivations we added ax we could recover Leaky ReLU by using a linear transformation to select $\max(x, ax)$ (similarly to Equation 6).

To recover maxout with groups of size k, we perform GroupSort with groups of size k and use the next linear transformation to select the first element of each group after sorting.

A.3. Expressivity of GroupSort

We show that GroupSort activation with different grouping sizes have the same expressive power. We also show that neural networks built with GroupSort activation and absolute value activation have the same expressive power.

Expressivity of Different Grouping Sizes FullSort can implement MaxMin by "chunking" the biases in pairs. To be more precise, let $x_{max} = \sup_{\mathbf{x}\in\mathcal{X}} ||\mathbf{x}||_\infty$ where $\mathcal{X}$ represents the domain, and $\boldsymbol{b} = [b_1, b_2, ..., b_n]^T$ where $x_{max} < b_1 = b_2 \ll b_3 = b_4 \ll \cdots \ll b_{n-1} = b_n$ ($\ll$ denotes differing by at least x_{max}). We can write:

$$\mathbf{MaxMin}(\mathbf{x}) = \mathbf{FullSort}(\mathbf{Ix} + \boldsymbol{b}) - \boldsymbol{b},$$

where $\mathbf{I}$ denotes the identity matrix.

Expressivity of GroupSort and Absolute Value Networks Under the matrix 2-norm constraint, neural neural networks built with GroupSort activation and absolute value activation have the same expressive power. The two operations can be written in terms of each other, as can be seen below:

$$\begin{bmatrix} \mathbf{max}(x) \\ \mathbf{min}(y) \end{bmatrix} = \mathbf{M}\,\mathbf{abs}(\mathbf{M}\begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} B \\ 0 \end{bmatrix}) - \begin{bmatrix} \sqrt{2}B \\ 0 \end{bmatrix}$$

where

$$\mathbf{M} = \begin{bmatrix} \frac{1}{\sqrt{2}} & \frac{1}{\sqrt{2}} \\ \frac{1}{\sqrt{2}} & \frac{-1}{\sqrt{2}} \end{bmatrix}$$

$$\mathbf{abs}(x) = \begin{bmatrix} \frac{1}{\sqrt{2}} & -\frac{1}{\sqrt{2}} \end{bmatrix} \mathbf{MaxMin}(\begin{bmatrix} \frac{1}{\sqrt{2}} \\ \frac{-1}{\sqrt{2}} \end{bmatrix} x)$$

In Equation A.3, the value of B is chosen such that $2\mathbf{x} + \sqrt{2}B > 0$ for all $\mathbf{x}$ in the domain. Note that all the matrices in these constructions satisfy the matrix 2-norm constraint.

A.4. Computational Considerations

Let n be the total number of pre-activations and k be the size of the groups used in GroupSort. Then, a naive CPU implementation of GroupSort has a complexity of $\frac{n}{k}\mathcal{O}(k \log k)$. However, this operation can be parallelized on GPU. We use the built-in GPU accelerated sorting implementation in PyTorch (Paszke et al., 2017) in our experiments. We find that the additional computational cost added by the GroupSort activation is dwarfed by the other components of network training and inference.

Note that MaxMin (GroupSort with a group size of 2) can be implemented either by concatenating the results of a Maxout and Minout operations (in which case it is roughly twice as costly as a single MaxOut operation), or as in its own custom CUDA kernel (Nvidia, 2010), in which case it can be as efficient as the ReLU operation.

B. Implementing norm constraints

B.1. Comparing Björck and Parseval

In Cisse et al. (2017), the authors motivate an update to the weight matrices by considering the gradient of a regularization term, $\frac{\beta}{2}||W^TW - I||_F^2$. By subtracting this gradient from the weight matrices they push them closer to the Stiefel manifold. The final update is given by,

$$W \leftarrow W(I + \beta) - \beta W W^T W \tag{8}$$

Note that when $\beta = 0.5$ this update is exactly the first order ($p = 1$) update from Equation 2, with a single iteration. Compared to our approach, the key difference in Parseval networks is that the weight matrix update is applied *after* the primary gradient update. Instead, we utilize Equation 2 during forward pass to optimize directly on the Stiefel manifold. This is more expensive but guarantees that the weight matrices are close to orthonormal during training.

Choice of β We can relate the first order Björck algorithm to the Parseval update by setting $\beta = 0.5$. However, in practice Parseval networks are trained with very small choices of β, for example $\beta = 0.0003$. When β is small the algorithm still converges to an orthonormal matrix but much more slowly. Figure 13 shows the maximum and minimum singular values of matrices which have undergone 50 iterations of the first order Björck scheme for varying choices of $\beta < 0.5$. When β is much smaller than 0.5 the matrices may be far from orthonormal. We also show how the maximum and minimum singular values vary over the number of iterations when $\beta = 0.0003$ (a common choice for Parseval networks) in Figure 12. This has practical implications for Parseval training, particularly when using early stopping, as the weight matrices may be far from orthonormal if the gradients are relatively large compared to the update produced by the Björck algorithm. We observed this effect empirically in our MNIST classification experiments but found that Parseval networks were still able to achieve a meaningful regularization effect.

B.2. Comparing Björck and Spectral Normalization

Spectral Normalization (Miyato et al., 2018) enforces the largest singular value of each weight matrix to be less than 1 by estimating the largest singular value and left/right singular vectors using power iteration, and normalizing the weight matrix using these during each forward pass. While this constraint does allow *all* singular values of the weight matrix to be 1, we have found that this rarely happens in practice. Hence, enforcing the 1-Lipschitz constraint via spectral normalization doesn't guarantee gradient norm preservation.

We demonstrate the practical consequences of the inability of spectral normalization to preserve gradient norm on the task of approximating high dimensional cones. In order

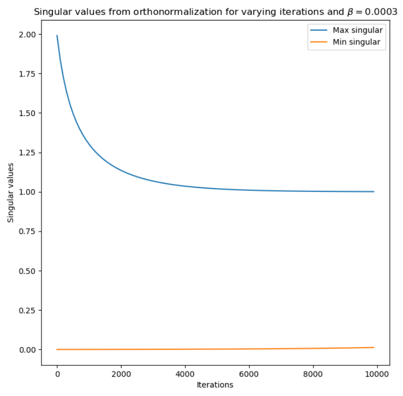


Figure 12: Convergence of the Björck algorithm for increasing iterations with $\beta = 0.0003$. The largest and smallest singular values are shown after each iteration.

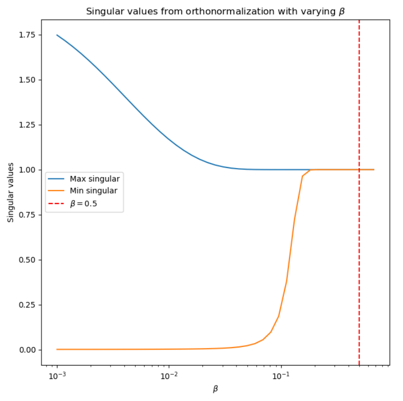


Figure 13: Convergence of the Björck algorithm for different choices of β. The largest and smallest singular values are shown after 50 iterations of the algorithm.

to quantify approximation performance, we carefully pick two n dimensional probability distributions such that 1) The Wasserstein Distance between them is exactly 1 and 2) the optimal dual surface consists of an $n-1$ dimensional cone with a gradient of 1 everywhere, embedded in n dimensions. We later train 1-Lipschitz constrained neural networks to optimize the dual Wasserstein objective in Equation 1 and check how well the choice of architecture is able to approximate the dual surface. Architectures that can obtain tighter estimates of Wasserstein distance are more expressive.

Figure 14 shows that neural networks trained with Björck orthonormalization not only are able to approximate high dimensional cones better than spectral normalization, but also converge much faster in terms of training iterations. The gap between these methods gets much more significant as the problem dimensionality increases. In this experiment, each network consisted of 3 hidden layers with 512

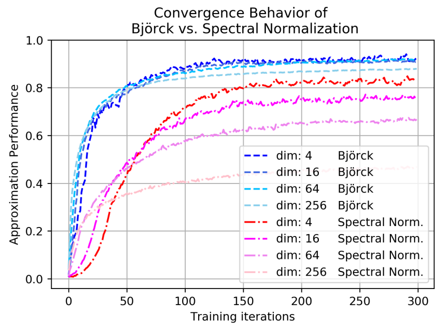


Figure 14: Comparing the performance of 1-Lipschitz neural nets using Björck orthonormalization vs. spectral normalization on the high dimensional cone fitting task (Section 7.1.1). Using Björck orthonormalization leads to fasted convergence and better approximation performance, measured by the estimated Wasserstein Distance.

hidden units per layer, and was trained with the Adam optimizer (Kingma & Ba, 2015) with its default hyperparameters. Tuned learning rates of 0.01 for Björck and 0.0033 for spectral normalization were used.

B.3. Sufficient Condition for Convergence of Björck Orthonormalization

The Björck orthonormalization can be shown to always converge as long as the condition $||\mathbf{W}^T\mathbf{W} - \mathbf{I}||_2 < 1$ is satisfied (Hasenclever et al., 2017). Since Björck orthonormalization is scale invariant, $(\mathbf{BJORCK}(\alpha\mathbf{W}) = \alpha\mathbf{BJORCK}(\mathbf{W}))$ (Björck & Bowie, 1971), the aforementioned sufficient condition can be implemented by simply scaling the weight matrix so that all of its singular values are less than or equal to 1 before orthonormalization.

A scaling factor can be computed efficiently by considering the following matrix norm inequalities:

$$\sigma_{max} \leq \sqrt{m * n}||\mathbf{W}||_{max} \tag{9}$$

$$\sigma_{max} \leq \sqrt{n}||\mathbf{W}||_1 \tag{10}$$

$$\sigma_{max} \leq \sqrt{m}||\mathbf{W}||_\infty \tag{11}$$

Above, σ_{max} corresponds to the largest singular value of the matrix and m and n stand for the number of rows and columns respectively. Note that computing the quantities on the right hand side of the inequalities involves at most summing over the rows or columns of the weight matrix, which is a cheap operation.

B.4. Computational Considerations Regarding Bjoŕck Orthonormalization

Björck orthonormalization is a costly operation even when implemented on a GPU, as it contains matrix-matrix products. In this section, we go over a few methods that can be used to accelerate Björck orthonormalization. Note that

GroupSort's additional cost is **only incurred during training**: once the network is trained, it is possible to use the orthonormalized parameters as the network weights and bypass the orthonormalization step.

Enforcing a soft Lipschitz constraint throughout training: We found in our experiments that one can run only a few iterations of Björck orthonormalization during training, then increase the number of iterations towards the end of training without hurting performance in classification tasks.

Performing spectral normalization before Björck orthonormalization: By normalizing the weight matrices by their spectral norm before Björck orthonormalization, one can not only guarantee convergence (as described in B.3), but also faster convergence. As opposed to guaranteeing convergence by normalizing the weights using estimates of other norms (as in equations 9, 10 and 11), normalizing by the spectral norm ensures the singular values of the matrix are closer to unity before Björck orthonormalization is run.

Implementing Björck using Matrix-Vector products: It is possible to rewrite the Björck algorithm in terms of Matrix-Vector products. This stems from the fact that we do not actually need to compute the entire orthonormal matrix $\tilde{A}$, but only a matrix-vector product $\tilde{A}v$, where v are the activations of the previous network layer. Recall the expression for one Björck update:

$$A_{k+1}v = \frac{3}{2}A_k v - \frac{1}{2}A_k A_k^T A_k v$$

Unfortunately, we cannot compute $A_{k+1}v$ from only $A_k v$ as we also need to compute $A_k A_k^T A_k v$ which requires A_k explicitly. However, we can rewrite the above using two operations: $u \mapsto A_k u$ and $u \mapsto A_k A_k^T u$. To see why this is useful, we write,

$$\begin{aligned} A_{k+1}A_{k+1}^T v &= (\frac{3}{2}A_k - \frac{1}{2}A_k A_k^T A_k) \\ &\quad (\frac{3}{2}A_k^T - \frac{1}{2}A_k^T A_k A_k^T)v \\ &= \frac{9}{4}A_k A_k^T v - \frac{3}{2}(A_k A_k^T)(A_k A_k^T)v \\ &\quad + \frac{1}{4}(A_k A_k^T)(A_k A_k^T)(A_k A_k^T)v \end{aligned}$$

Hence, we can write $u \mapsto A_{k+1}A_{k+1}u$ as a function of $u \mapsto A_k A_k u$. This allows us to recursively define the matrix-vector product of the kth iterate in terms of the previous iterates.

This method works very well for relatively few iterations (approximately less than 5) but scales poorly as the number of iterations increases. This is because the algorithm requires $O(3^k)$ matrix-vector products. Table 3 shows a runtime comparison for the original algorithm and the Matrix-Vector Product (MVP) for increasing iterations averaged over 10 runs. The weight matrices have a dimension of 1000×1000 and are normalized using the equation 11 to guarantee convergence.

Iterations	1	2	3	5	10
Full Björck	0.020	0.039	0.059	0.095	0.21
MVP Björck	0.0002	0.0005	0.0011	0.012	2.88
Speedup Factor	99.98x	78.59x	53.56x	7.77x	0.07x

Table 3: Runtime (seconds) for full Björck and Matrix-Vector Product (MVP) Björck for a 1000×1000 matrix, averaged over 10 runs.

C. Projecting Vectors on L_∞ Ball

The following algorithm uses sorting to project vectors on L_∞ balls (Condat, 2016).

Algorithm 1 L_∞ Projection via. Sorting

```
  Input: $\boldsymbol{y} \in \mathbb{R}^N$, Output: $\boldsymbol{x} \in \mathbb{R}^N$ .
  Sort $\boldsymbol{y}$ into $\boldsymbol{u}$: $u_1 \geq \ldots \geq u_N$ .
  Set $K := \max_{1 \leq k \leq N}\{k | (\sum_{r=1}^{k} u_r - 1)/k < u_k\}$.
  Set $\tau := (\sum_k^K u_k - 1)/K$.
  for $n = 1, \ldots, N$ do
    Set $x_n := \max_{y_n - \tau, 0}$
  end for
```

D. Non-expressive norm-constrained networks are linear

Theorem 1. *Consider a neural net, $f : \mathbb{R}^n \to \mathbb{R}$, built with matrix 2-norm constrained weights ($||\mathbf{W}||_2 \leq 1$) and 1-Lipschitz, element-wise, monotonic activation functions. If $||\nabla f(\mathbf{x})||_2 = 1$ almost everywhere, then f is linear.*

Proof. We can express the input-output Jacobian of a neural network as:

$$\frac{\partial f}{\partial \mathbf{x}} = \frac{\partial f}{\partial \boldsymbol{h}_{L-1}} \frac{\partial \boldsymbol{h}_{L-1}}{\partial \boldsymbol{z}_{L-1}} \frac{\partial \boldsymbol{z}_{L-1}}{\partial \mathbf{x}} = \mathbf{W}_L \frac{\partial \phi(\boldsymbol{z}_{L-1})}{\partial \boldsymbol{z}_{L-1}} \frac{\partial \boldsymbol{z}_{L-1}}{\partial \mathbf{x}}$$

Note that $\mathbf{W}_L \in \mathbb{R}^{1 \times n_{L-1}}$. Moreover, using the sub-multiplicativity of matrix norms, we can write:

$$\begin{aligned} 1 = \left|\left|\frac{\partial f}{\partial \mathbf{x}}\right|\right|_2 &\leq \left|\left|\mathbf{W}_L \frac{\partial \phi(\boldsymbol{z}_{L-1})}{\partial \boldsymbol{z}_{L-1}}\right|\right|_2 \left|\left|\frac{\partial \boldsymbol{z}_{L-1}}{\partial \mathbf{x}}\right|\right|_2 \\ &\leq ||\mathbf{W}_L||_2 \left|\left|\frac{\partial \phi(\boldsymbol{z}_{L-1})}{\partial \boldsymbol{z}_{L-1}}\right|\right|_2 \left|\left|\frac{\partial \boldsymbol{z}_{L-1}}{\partial \mathbf{x}}\right|\right|_2 \leq 1 \end{aligned}$$

for x almost everywhere. The quantity is also upper bounded by 1 due to the 1-Lipschitz property. Therefore, all of the Jacobian norms in the above equation must be equal to 1. Notably,

$$\left|\left|\mathbf{W}_L \frac{\partial \phi(\boldsymbol{z}_{L-1})}{\partial \boldsymbol{z}_{L-1}}\right|\right|_2 = 1 \quad \text{and} \quad ||\mathbf{W}_L||_2 = 1$$

We then consider the following operation:

$$\begin{aligned}&||\mathbf{W}_L||_2^2 - \left|\left|\mathbf{W}_L \frac{\partial \phi(\boldsymbol{z}_{L-1})}{\partial \boldsymbol{z}_{L-1}}\right|\right|_2^2 \\ &= \sum_{i=1}^{n}(1 - \Big(\frac{\partial \phi(\boldsymbol{z}_{L-1})}{\partial \boldsymbol{z}_{L-1}}\Big)^2_{ii})(W_{L,i})^2 = 0\end{aligned} \tag{12}$$

We have $0 \leq \frac{\partial \phi}{\partial \boldsymbol{z}_L} \leq 1$ as ϕ is 1-Lipschitz and monotonically increasing. Therefore, we must have either $\frac{\partial \phi}{\partial \boldsymbol{z}_L}_{ii} = 1$ almost everywhere, or $\mathbf{W}_{L,i} = 0$. Thus we can write,

$$\begin{aligned}\boldsymbol{z}_L &= \sum_{i=1}^{m} W_{L,i}\phi(\boldsymbol{z}_{L-1})_i + b_L \\ &= \sum_{i:W_{L,i}\neq 0} W_{L,i}\phi(\boldsymbol{z}_{L-1})_i + b_L \\ &= \sum_{i:W_{L,i}\neq 0} W_{L,i}\boldsymbol{z}_{L-1,i} + b_L\end{aligned}$$

Then $\boldsymbol{z}_L$ can be written as a linear function of $\boldsymbol{z}_{L-1}$ almost everywhere and by Lipschitz continuity we must have that $\boldsymbol{z}_L$ *is* a linear function of $\boldsymbol{z}_{L-1}$. In particular, we can write $\boldsymbol{z}_L = \mathbf{W}_L\mathbf{W}_{L-1}\boldsymbol{h}_{L-2} + (\mathbf{W}_L\boldsymbol{b}_{L-1} + b_L)$, collapsing the last two layers into a single linear layer, with weight matrix $\mathbf{W}_L\mathbf{W}_{L-1} \in \mathbb{R}^{1\times n_{L-2}}$ and scalar bias $\mathbf{W}_L\boldsymbol{b}_{L-1} + b_L$.

From here we can apply the exact same argument as above to $\phi(\mathbf{z}_{L-2})$, reducing the next layer to be linear. By repeating this all the way to the first linear layer we collapse the network into a single linear function. □

Theorem 2. *Consider a network, $f : \mathbb{R}^n \to \mathbb{R}$, built with matrix 2-norm constrained weights and with $||\nabla f(\mathbf{x})||_2 = 1$ almost everywhere. Without changing the computed function, each weight matrix $\mathbf{W} \in R^{m\times k}$ can be replaced with a matrix $\tilde{\mathbf{W}}$ whose singular values all equal 1.*

Proof. Take a weight matrix $\mathbf{W}_i$, for $i < L$. By the argument in the proof of Theorem 1, this matrix must preserve the norm of gradients during backpropagation. That is,

$$1 = \left|\left|\frac{\partial f}{\partial \boldsymbol{z}_i}\mathbf{W}_i\right|\right|_2$$

Using the singular value decomposition, we write $\mathbf{W}_i = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^T$. We then define $\tilde{\mathbf{W}}_i = \mathbf{U}\tilde{\mathbf{\Sigma}}\mathbf{V}^T$ where $\tilde{\mathbf{\Sigma}}$ has ones along the diagonal. Furthermore, define $\mathbf{W}_i^{(t)} = t\mathbf{W}_i + (1-t)\tilde{\mathbf{W}}_i$. Replacing $\mathbf{W}_i$ with $\mathbf{W}_i^{(t)}$ in the network:

$$\begin{aligned}\frac{\partial f}{\partial t} &= \frac{\partial f}{\partial \boldsymbol{z}_i}\frac{\partial \boldsymbol{z}_i}{\partial t} = \frac{\partial f}{\partial \boldsymbol{z}_i}(\mathbf{W}_i - \tilde{\mathbf{W}}_i)\boldsymbol{h}_{i-1} \\ &= \frac{\partial f}{\partial \boldsymbol{z}_i}\mathbf{U}(\mathbf{\Sigma}_i - \tilde{\mathbf{\Sigma}}_i)\mathbf{V}^T\boldsymbol{h}_{i-1}\end{aligned}$$

As the norm of $\frac{\partial f}{\partial \boldsymbol{z}_i}$ is preserved by $\mathbf{W}_i$ we must have that $\boldsymbol{u} = (\frac{\partial f}{\partial \boldsymbol{z}_i}\mathbf{U})^T$ has non-zero entries only where the diagonal of $\mathbf{\Sigma}$ is 1. That is, $u_j = 0 \iff \mathbf{\Sigma}_{jj} < 1$. In particular, we have $\boldsymbol{u}^T\mathbf{\Sigma}_i = \boldsymbol{u}^T\tilde{\mathbf{\Sigma}}_i$ meaning $\frac{\partial f}{\partial t} = 0$. Thus, the output of the network is the same for all t, in particular for $t = 0$ and $t = 1$. Thus, we can replace $\mathbf{W}_i$ with $\tilde{\mathbf{W}}_i$ and the network output remains unchanged.

We can repeat this argument for all $i < L$ (for $i = 1$ we adopt the notation $\boldsymbol{h}_0 = \boldsymbol{x}$, the input to the network). For $i = L$ the result follows directly. □

E. Universal Approximation of 1-Lipschitz Functions

Here we present formal proofs related to finding neural network architectures which are able to approximate any 1-Lipschitz function. We begin with a proof of Lemma 1.

Lemma 1. *(Restricted Stone-Weierstrass Theorem) Suppose that (X, d_X) is a compact metric space with at least two points and L is a lattice in $C_L(X, \mathbb{R})$ with the property that for any two distinct elements $x, y \in X$ and any two real numbers a and b such that $|a - b| \leq d_X(x, y)$ there exists a function $f \in L$ such that $f(x) = a$ and $f(y) = b$. Then L is dense in $C_L(X, \mathbb{R})$.*

Proof. This proof follows a standard approach with small modifications. We aim to show that for any $g \in C_L(X, \mathbb{R})$ and $\epsilon > 0$ we can find $f \in L$ such that $||g - f||_\infty < \epsilon$ (i.e. the largest difference is ϵ).

Fix $x \in X$. Then for each $y \in X$, we have an $f_y \in L$ with $f_y(x) = g(x)$ and $f_y(y) = g(y)$. This follows from the separation property of L and, using the fact that g is 1-Lipschitz, $|g(x) - g(y)| \leq d_X(x, y)$.

Define $V_y = \{z \in X : f_y(z) < g(z) + \epsilon\}$. Then V_y is open and we have $x, y \in V_y$. Therefore, the collection of sets $\{V_y\}_{y\in X}$ is an open cover of X. By the compactness of X, there exists some finite subcover of X, say, $\{V_{y_1}, \dots, V_{y_n}\}$, with corresponding functions $f_{y_1}, \dots, f_{y_n}$.

Let $F_x = min(f_{y_1}, \dots, f_{y_n})$. Since L is a lattice we must have $F_x \in L$. And moreover, we have that $F_x(x) = g(x)$ and $F_x(z) < g(z) + \epsilon$, for all $z \in X$.

Now, define $U_x = \{z \in X : F_x(z) > g(z) - \epsilon\}$. Then U_x is an open set containing x. Therefore, the collection $\{U_x\}_{x\in X}$ is an open cover of X and admits a finite subcover, $\{U_{x_1}, \dots, U_{x_m}\}$, with corresponding functions $F_{x_1}, \dots, F_{x_m}$.

Let $G = max(F_{x_1}, \dots, F_{x_m}) \in L$. We have $G(z) > g(z) - \epsilon$, for all $z \in X$.

Combining both inequalities, we have that $g(z) - \epsilon < G(z) < g(z) + \epsilon$, for all $z \in X$. Or more succinctly, $||g - G||_\infty < \epsilon$. The result is proved by taking $f = G$. □

We now proceed to prove Theorem 3.

Theorem 3. *(Universal Approximation with Lipschitz Networks) Let $\mathcal{LN}_p$ denote the class of fully-connected networks whose first weight matrix satisfies $||\mathbf{W}_1||_{p,\infty} = 1$, all other weight matrices satisfy $||\mathbf{W}||_\infty = 1$, and GroupSort activations have a group size of 2. Let X be a closed and bounded subset of $\mathbb{R}^n$ endowed with the L_p metric. Then the closure of $\mathcal{LN}_p$ is dense in $C_L(X, \mathbb{R})$.*

Proof. The first property we require is separation of points. This follows trivially as given four points satisfying the required conditions we can find a linear map with the required $L_{p,\infty}$ matrix norm that fits them. It remains then to prove that we can construct a lattice under this constraint. We begin by considering two 1-Lipschitz neural networks, f and g. We wish to design an architecture which is guaranteed to be 1-Lipschitz and can represent $\max(f, g)$ and $\min(f, g)$.

The key insight is that we can split the network into two parallel *channels* each of which computes one of f and g. At the end of the network, we can then select one of these channels depending on whether we want the max or the min.

Each of the networks f and g is determined by a set of weights and biases, we will denote these $[\mathbf{W}_1^f, \mathbf{b}_1^f, \ldots, \mathbf{W}_n^f, b_n^f]$ and $[\mathbf{W}_1^g, \mathbf{b}_1^g, \ldots, \mathbf{W}_n^g, \mathbf{b}_n^g]$ for f and g respectively. For now, assume that these networks are of equal depth (we can lift this assumption later) however we make no assumptions on the width. We will now construct $h = max(f, g)$ in the form of a 1-Lipschitz neural network. We will design a network h which first concatenates the first layers of networks f and g and then computes f and g separately before combining them at the end.

We take the first weight matrix of h to be $\mathbf{W}_1^h = [\mathbf{W}_1^f \ \mathbf{W}_1^g]^T$, the weight matrices of f and g stacked vertically. This matrix necessarily satisfies $||\mathbf{W}_1^h||_{p,\infty} = 1$. Similarly, the bias will be those from the first layers of f and g stacked vertically. Then the first layer's pre-activations will be exactly the pre-activations of f and g stacked vertically.

For the following layers, we construct the biases in the same manner (vertical stacking). We construct the weights by constructing new block-diagonal weight matrices. That is, given $\mathbf{W}_i^f$ and $\mathbf{W}_i^g$, we take

$$W_i^h = \begin{bmatrix} W_i^f & 0 \\ 0 & W_i^g \end{bmatrix}$$

This matrix also has ∞-norm equal to 1. We repeat this for each of the layers in f and g and end up with a final layer which has two units, f and g. We can then take MaxMin of this final layer and take the inner product with $[1, 0]$ to recover the max or $[0, 1]$ for the min.

Finally, we must address the case where the depth of f and g are different. In this case we notice that we are able to represent the identity function with MaxMin activations. To do so observe that after the pre-activations have been sorted we can multiply by the identity and the sorting activation afterwards will have no additional effect. Therefore, for the channel that has the smallest depth we can add in these additional identity layers to match the depths and resort to the above case.

We have shown that the set of neural networks is a lattice which separates points, and thus by Lemma 1 it must be dense in $C_L(X, \mathbb{R})$. □

Note that we could have also used the maxout activation (Goodfellow et al., 2013) to complete this proof. This makes sense, as the maxout activation is also norm-preserving in L_∞. However, this does not hold when using a 2-norm constraint on the weights. We now present several consequences of the theoretical results given above.

This result can be extended easily to vector-valued Lipschitz functions with respect to L_∞ distance by noticing that the space of such 1-Lipschitz functions is a lattice. We may apply the Stone-Weierstrass proof to each of the coordinate functions independently and use the same construction as in Theorem 3 modifying only the last layer which will now reorder the outputs of each function to do a pairwise comparison and then select the relevant components to produce the max or the min.

Observation. *Consider the set of networks, $\mathcal{LN}_\infty^m = \{f : \mathbb{R}^n \to \mathbb{R}^m, ||W||_\infty = 1\}$. Then $\mathcal{LN}_\infty^m$ is dense in 1-Lipschitz functions with respect to the L_∞ metric.*

Proof. Note that given two functions, $g, f : \mathbb{R}^n \to \mathbb{R}^m$ which are 1-Lipschitz with respect to the L_∞ metric, their element-wise max (or min) is also 1-Lipschitz with respect to the L_∞ metric. Consider the element-wise components of such an f, written $f = (f_1, \ldots, f_m)$. We can apply the Stone-Weierstrass theorem (Lemma 1) to each of the components independently, such that if the same conditions apply (trivially extended to $\mathbb{R}^m$) the Lattice is dense. Thus, as in the proof of Theorem 3, it suffices to find a network $h \in \mathcal{LN}_\infty^m$ which can represent the max or min of any other networks, $f, g \in \mathcal{LN}_\infty^m$.

In fact, we can use almost exactly the same construction as in the proof of Theorem 3. We follow the same initial steps by concatenating weight matrices and constructing block-diagonal matrices from the two networks. After doing this for all layers in the networks f and g, we will output $[f_1, \ldots, f_m, g_1, \ldots g_m]$. We can then permute these entries using a single linear layer to produce $[f_1, g_1, f_2, g_2, \ldots, f_m, g_m]$ finally we take MaxMin and use the final weight matrix to select either $\max(f, g)$ or $\min(f, g)$. □

F. Spectral Jacobian Regularization

Most existing work begins with the goal of constraining the spectral norm of the Jacobian and proceeds to achieve this by placing constraints on the weights of the network (Yoshida & Miyato, 2017). While not the main focus of our work, we propose a simple new technique which allows us to directly regularize the spectral norm of the Jacobian, $\sigma(J)$. This method differs from the ones described previously as the Lipschitz constant of the entire network is regularized using a single term, instead of at the layer level.

The intuition for this algorithm follows that of Yoshida & Miyato (2017), who apply power iteration to estimate the singular values of the weight matrices online. The authors also discuss computing the spectral radius of the Jacobian directly, and related quantities such as the Frobenius norm, but dismiss this as being too computationally expensive.

Power iteration can be used to compute the leading singular value of a matrix J with the following repeated steps,

$$\mathbf{v}_k = J^T \mathbf{u}_{k-1}/||J^T \mathbf{u}_{k-1}||_2, \mathbf{u}_k = J\mathbf{v}_k/||J\mathbf{v}_k||_2$$

Then we have $\sigma(J) \approx \mathbf{u}^T J \mathbf{v}$. There are two challenges that must be overcome to implement this in practice. First, the algorithm requires higher order derivatives which leads to increased computational overhead. However, the tradeoff is often reasonable in practice, see e.g. Drucker & Le Cun (1992). Second, the algorithm requires both Vector-Jacobian products and Jacobian-Vector products. The former can be computed with reverse-mode automatic differentiation but the latter requires the less common forward-mode. Fortunately, one can recover forward-mode from reverse mode by constructing Vector-Jacobian products and utilizing the transpose operator (Townsend, 2017). We can re-use the intermediate reverse-mode backpropagation within the algorithm which further reduces the computational overhead. The algorithm itself is presented as Algorithm 2.

Algorithm 2 Spectral Jacobian Regularization

```
Initialize $u$ randomly, choose hyperparameter $\lambda > 0$
for data batch (X,Y) do
    Compute logits $f_\theta(X)$
    Compute loss $\mathcal{L}(f_\theta(X), Y)$
    Compute $\mathbf{g} = \mathbf{u}^T \frac{\partial \mathbf{f}}{\partial \mathbf{x}}$, using reverse mode
    Set $\mathbf{v} = \mathbf{g}/||\mathbf{g}||_2$
    Compute $\mathbf{h} = (\mathbf{v}^T \frac{\partial \mathbf{g}}{\partial \mathbf{u}})^T = \frac{\partial \mathbf{f}}{\partial \mathbf{x}} \mathbf{v}$, using reverse mode
    Update $\mathbf{u} = \mathbf{h}/||\mathbf{h}||_2$
    Compute parameter update from $\frac{\partial}{\partial \theta} \left(\mathcal{L} + \lambda \mathbf{u}^T \mathbf{h}\right)$
end for
```

We present this algorithm primarily to be used for regularization but this could also be used to approximately control the Lipschitz constraint by rescaling the output of the entire network by the estimate of the Jacobian spectral norm similar to spectral normalization (Miyato et al., 2018).

	ReLU	MaxMin	GS(4)	FullSort	Maxout
Standard	1.61	1.47	1.62	3.53	1.40
Dropout	1.27	1.37	1.29	3.62	1.27
Björck	1.54	1.25	1.43	2.06	1.43
Spectral Norm	1.54	1.26	1.32	2.94	1.26
Spectral Jac	1.05	1.09	1.24	1.93	1.02
Parseval	1.43	1.40	1.44	3.36	1.35
L_∞	2.25	2.28	2.22	4.88	1.98

Table 4: **MNIST classification** Test error shown for different architectures and activations (GS stands for GroupSort.).

G. Additional Experiments

We present additional experimental results.

G.1. Classification

We compared a wide range of Lipschitz architectures and training schemes on some simple benchmark classification tasks. We demonstrate that we are able to learn Lipschitz neural networks which are expressive enough to perform classification without sacrificing performance.

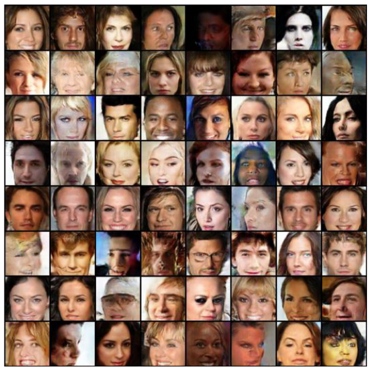
MNIST Classification We explored classification with a 3-layer fully connected network with 1024 hidden units in each layer. Each model was trained with the Adam optimizer (Kingma & Ba, 2015). The results are presented in Table. 4.

For all models the GroupSort activation is able to perform classification well - especially when the Lipschitz constraint is enforced. Surprisingly, we found that we could apply the GroupSort activation to sort the entire hidden layer and still achieve reasonable classification performance, even with dropout. In terms of classification performance, spectral Jacobian regularization was most effective (Appendix F).

While the Parseval networks are capable of learning a strict Lipschitz constraint this does not always hold in practice. A small beta value leads to slow convergence towards orthonormal weights. When early stopping is used, which is typically important for good validation accuracy, it is difficult to ensure that the resulting network is 1-Lipschitz.

Classification with little data While enforcing the Lipschitz constraint aggressively could hurt overall predictive performance, it decreases the generalization gap substantially. Motivated by the observations of Bruna & Mallat (2013) we investigated the performance of Lipschitz networks on small amounts of training data, where learning robust features to avoid overfitting is critical.

For these experiments we kept the same network architecture as before. We trained standard unregularized networks,


(a) Leaky-ReLU Critic

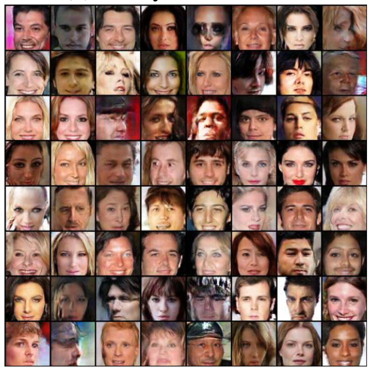

(b) MaxMin Critic

Figure 15: Generated images from WGAN-GP models trained on the CelebA dataset.

networks with dropout, networks regularized with weight decay, and 1-Lipschitz neural networks enforced with the Björck algorithm. We made use of a LeNet-5 architecture, with convolutions and max-pooling — the latter prevents norm preservation and thus may reduce the effectiveness of MaxMin substantially. We found that Dropout was the most effective regularizer in this case but confirmed that networks with Lipschitz constraints were able to significantly improve generalization. Full results are in Table 5.

Classification on CIFAR-10 We briefly explored classification on CIFAR-10 using Wide ResNets (Depth 28, Width 4) (Zagoruyko & Komodakis, 2016; He et al., 2016). We performed these experiments primarily to explore the effectiveness of the MaxMin activation in a more challenging setting. We used the optimal optimization hyperparameters for ReLU with SGD and performed a small search over regularization parameters for Parseval and Spec Jac regularization. We present results in Table 6. We found that MaxMin performed comparably to ReLU in this setting and hope to explore this further in future work.

G.2. Training WGAN-GP

We found that the MaxMin activation could also be used as a drop-in replacement for ReLU activations in WGAN architectures that utilize a gradient-norm penalty in the training objective. We took an existing implementation of WGAN-GP which used a fully convolutional critic network with 5 layers and LeakyReLU activations. The generator used a linear layer followed by 4 deconvolutional layers. We trained this model with the tuned hyperparameters for the LeakyReLU activation and then used the same settings to train a model with MaxMin acivations. We defer a more thorough study of this setting to future work but present here the output of the trained generators after 50 epochs of training on the CelebA dataset (Liu et al., 2015) in Figure 15.

G.3. Dynamical Isometry

In Figure 16 we plot the distribution of all singular values of ReLU and GroupSort 2-norm-constrained networks trained as MNIST classifiers, with a Lipschitz constant of 10. While the ReLU singular values are spread between 4-8 the GroupSort network concentrates the singular values in range 9-10. Dynamical isometry (Pennington et al., 2017) requires all Jacobian singular values to be concentrated around 1. Using 2-norm constraints and GroupSort activations we are able to achieve dynamical isometry throughout training.

H. Experiment Details

We present additional experimental details.

H.1. Designing Synthetic Distributions for Wasserstein Distance Estimation

Absolute value We pick $p_1(\mathrm{x}) = \delta_0(x)$ and $p_2(\mathrm{x}) = \frac{1}{2}\delta_{-1}(x) + \frac{1}{2}\delta_1(x)$, where $\delta_\alpha(x)$ stands for the Dirac delta function located at α. The optimal dual surface learned while computing the Wasserstein distance between p_1 and p_2 is the absolute value function. This also makes intuitive sense, as the function that assigns "as low values as possible" at $x = 0$ and assigns "as high values as possible" at $x = -1$ and $x = 1$ while satistying 1-Lipschitz condition must be the absolute value function.

The transport plan that minimizes the primal objective will

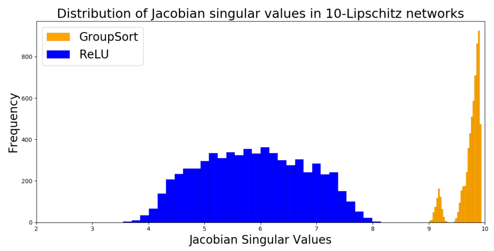


Figure 16: **Jacobian singular values distribution** We compare the Jacobian singular values of ReLU and GroupSort networks.

Data Size	Standard		Dropout		Weight Decay		Björck	
	ReLU	MaxMin	ReLU	MaxMin	ReLU	MaxMin	ReLU	MaxMin
300	12.40	12.14	7.30	10.64	11.06	10.81	8.12	7.81
500	8.57	9.13	5.54	6.15	7.33	7.50	5.96	6.98
1000	5.95	6.23	3.70	4.58	5.14	6.05	4.45	4.54
5000	2.54	2.51	1.84	2.15	2.31	2.55	2.23	2.31
10000	1.77	1.76	1.26	1.70	1.58	1.57	1.66	1.64

Table 5: **MNIST Classification with limited data** Test error for varying architectures and activations per training data size.

	Standard		Parseval		Spec Jac Regularization	
	ReLU	MaxMin	ReLU	MaxMin	ReLU	MaxMin
CIFAR-10	95.29	94.57	95.45	94.83	95.44	94.62

Table 6: **CIFAR-10 Classification** Test accuracy for Wide ResNets (Depth 28, Width 4) with varying activations and training schemes.

simply be to map the center Dirac delta equally to the ones near it. This leads to a Wasserstein distance of 1.

The networks we trained had 3 hidden layers each with 128 hidden units. We report the results obtained with the Aggregated Momentum optimizer (AggMo) (Lucas et al., 2018) with its default parameters, as it lead to faster convergence in our experiments compared to Adam optimizer (Kingma & Ba, 2015). We note that the choice of optimizer had minimal impact on the final Wasserstein Distance estimates.

Multiple 2D Circular Cones We describe the probability distributions p_1 and p_2 implicitly by describing how we sample from them. p_1 is sampled from by selecting one of the three points ($(-2, 0)$, $(0, 0)$ and $(2, 0)$) uniformly. p_2 is sampled from by first uniformly selecting one of the three points aforementioned, then uniformly sampling a point on the circle surrounding it, with radius 1. Wasserstein dual problem aims to find a Lipschitz function which assigns ”as high as possible” values to the three points, and ”as low as possible” values to the circles with radius 1 surrounding the three points. Hence, the optimal dual function must consist of three cones centered around $(-2, 0)$, $(0, 0)$ and $(2, 0)$. The behavior of the function outside this support doesn't have an impact on the solution.

The optimal transport plan must map the probability mass to the nearby circles surrounding them uniformly. This leads to an Wasserstein distance of 1.0.

The networks we trained had 3 hidden layers with 312 hidden units. We used the Aggretated Momentum optimizer (AggMo) (Lucas et al., 2018) with its default parameters.

***n* Dimensional Circular Cones** This is a simple extension of the absolute value case described above.

We pick p_1 as the Dirac delta function located at the origin, and sample from p_2 by uniformly selecting a point from high dimensional spherical shell with radius 1, centered at the origin. Following similar arguments developed for absolute value, it can be shown that the optimal dual function is a single high dimensional circular cone and the Wasserstein distance is also equal to unity.

H.2. Wasserstein Distance Estimation

The GAN variants we trained on MNIST and CIFAR10 datasets used the WGAN formulation first introduced in Arjovsky et al. (2017). The architectures of the generator and critic networks were the same as the ones used in(Chen et al., 2016). For the subsequent task of Wasserstein distance estimation, the weights of the generator networks were frozen after the initial GAN training has converged.

H.3. Wasserstein GAN with 1-Lipschitz Layers

We borrowed the discriminator and generator networks from Chen et al. (2016), but switched the ReLU activations with MaxMin and replaced the convolutional and fully connected layers with their Björck counterparts. We didn't use batch normalization, as this would violate the Lipschitz constraint.

H.4. Classification

For MNIST classification, we searched the hyperparameters as follows. For Björck, L_∞ constrained, and Spectral Norm architectures we tried networks with a guaranteed Lipschitz constant of 0.1, 1, 10 or 100. For Parseval networks we tried β values in the range 0.001, 0.01, 0.1, 0.5. For SpecJac regularization we scaled the penalty by 0.01, 0.05, or 0.1.

In order to scale the Lipschitz constant of the network, we introduce constant scaling layers in the network such that the product of the constant scale parameters is equal to the Lipschitz constant. As the activation functions are homogeneous, e.g. $\text{ReLU}(a\mathbf{x}) = a\text{ReLU}(\mathbf{x})$, this is equivalent to scaling the output of the network as described in Section 4.

H.5. Robustness and Interpretability

For the adversarial robustness experiments we trained fully-connected MNIST classifiers with 3 hidden layers each with 1024 units. We used the L_∞ projection algorithm referenced in Section 4.2. We applied the projection to each row in the weight matrices after each gradient update.

Our implementation of the FGS attack is standard but we found that the loss proposed by Carlini & Wagner (2016) (in particular, f_6 which the authors found most effective) was necessary to generate attacks for the Margin-0.3 MaxMin network (and produced stronger adversarial examples for the other networks). PGD also had difficulty generating adversarial examples for the Margin-0.3 MaxMin network. It was necessary to run PGD for 200 iterations and to use a scaled down version of the random initialization typically used: instead of randomly perturbing $\boldsymbol{x}$ in the ϵ ball we perturbed it by at most $\epsilon/10$ before running the usual scheme. Table 7 summarizes our results.

For the intepretable gradients in Figure 7 we used the same architecture, but switched to 2-norm constraints. We chose a random image from classes 1-4 and computed the input-output gradient with respect to the loss function. We found that similar results were achieved with ∞-norm projections (and hinge loss) but the uniform gradient scale made the 2-norm-constrained input-output gradients easier to visualize.

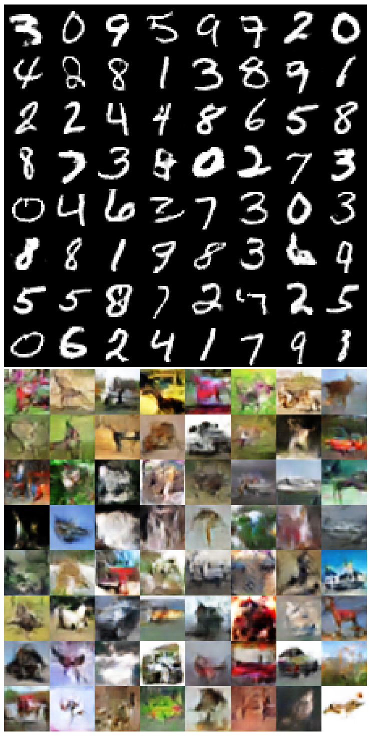


Figure 17: Samples from WGANs trained on MNIST and CIFAR10 whose critics use gradient norm preserving units.

Model	Clean	FGS		PGD	
	Err. $\backslash\epsilon$	0.1	0.3	0.1	0.3
Standard ReLU	1.6	98.3	100.0	100.0	100.0
Standard MaxMin	1.5	98.2	100.0	100.0	100.0
Margin-0.1 ReLU	6.2	88.3	100.0	89.7	100.0
Margin-0.1 MaxMin	1.9	36.3	99.2	44.4	99.8
Margin-0.3 ReLU	16.9	70.1	100.0	70.3	100.0
Margin-0.3 MaxMin	5.3	20.5	62.2	24.4	**77.7**
PGD 0.1	**1.02**	8.6	74.4	17.9	100.0
PGD 0.15	1.36	**8.1**	**52.9**	**15.1**	99.7

Table 7: **Adversarial robustness** The classification error for varying L_∞ distance of adversarial attacks. A perturbation size of 0.1 and 0.3 was used.