Volume-preserving Neural Networks: A Solution to the Vanishing Gradient Problem

Gordon MacDonald  
School of Mathematical and Computational Sciences  
University of Prince Edward Island  
Charlottetown, PE C1A 4P3 Canada  
GMACDONALD@UPEI.CA

Andrew Godbout  
School of Mathematical and Computational Sciences  
University of Prince Edward Island  
Charlottetown, PE C1A 4P3 Canada  
AGODBOUT@UPEI.CA

Bryn Gillcash  
School of Mathematical and Computational Sciences  
University of Prince Edward Island  
Charlottetown, PE C1A 4P3 Canada  
BGILLCASH@UPEI.CA

Stephanie Cairns  
Department of Mathematics and Statistics  
McGill University  
Montreal, QC H3A 0E9 Canada  
STEPHANIE.CAIRNS@MAIL.MCGILL.CA

Abstract

We propose a novel approach to addressing the vanishing (or exploding) gradient problem in deep neural networks. We construct a new architecture for deep neural networks where all layers (except the output layer) of the network are a combination of rotation, permutation, diagonal, and activation sublayers which are all volume preserving. This control on the volume forces the gradient (on average) to maintain equilibrium and not explode or vanish. Volume-preserving neural networks train reliably, quickly and accurately and the learning rate is consistent across layers in deep volume-preserving neural networks. To demonstrate this we apply our volume-preserving neural network model to two standard datasets.

Keywords: volume-preserving, neural network, machine learning, deep learning, vanishing gradient problem

1. Introduction

Deep neural networks are characterized by the composition of a large number of functions (aka layers), each typically consisting of an affine transformation followed by a non-affine “activation function”. Each layer is determined by a number of parameters which are trained on data to approximate some function. The deepness refers to the number of such functions composed (or the number of layers). The number of layers required to be deep is not well-defined, but an overview of deep learning (Schmidhuber, 2015) states that any
network with more than three layers is deep, and any network with more than ten layers is very deep.

Deep neural networks have been successfully applied to a number of difficult machine learning problems, such as image recognition (Krizhevsky et al., 2012), speech recognition (Hinton et al., 2012), and natural language processing (Cho et al., 2014).

In deep neural networks trained via gradient descent methods with backpropagation, the problem of vanishing gradients makes it difficult to train the parameters of the network. The backpropagation equations, via the chain rule, multiply a large number of derivatives in deep networks. If too many of these derivatives are small, the gradients vanish, and little learning happens in early layers of the network. In the standard neural network model, there are two main contributors to small derivatives: activation functions which often squash vectors and as such have small derivatives on a large portion of their domain; and weight matrices which act compressively on large parts of their domain.

The vanishing gradient problem was first identified by Sepp Hochreiter in his Diploma thesis in 1991 (Hochreiter, 1991) (see also Bengio et al. (1994)), and there have been a number of approaches to addressing this problem, either by modifying how the activation functions or weight matrices act, or by adding additional features to compensate for the loss of gradients, or a combination of these. These techniques include using alternative activation functions (such as ReLU) (Nair and Hinton, 2010), alternative weight matrices (such as unitary matrices) (Arjovsky et al., 2016; Jing et al., 2017), multi-level hierarchies (Schmidhuber, 1992), long short-term memory (LSTM) units (Hochreiter and Schmidhuber, 1997) and gated recurrent units (GRU) (Cho et al., 2014) to name a few.

Despite this, the issue of vanishing/exploding gradients is still problematic in many deep neural networks. With recurrent neural networks, the need to find a solution to the vanishing/exploding gradient is particularly acute as such networks need to be deep to handle long-term dependencies.

Our approach is to tackle the two main sources of vanishing gradients: the activation functions and the weight matrices, by replacing them with mathematical variants which are volume preserving. Enforcing volume preservation ensures that gradients cannot universally vanish or explode. We replace the standard weight matrix a product of rotation, permutation, and diagonal matrices, all of which are volume preserving. We replace the usual entrywise-acting activation functions by coupled activation functions which act pairwise on entries of an input vector (rather than entrywise) and allows us to use a wider selection of activation functions, ones that can “squash” while still being volume preserving.

In Section 2, we delineate the model for a basic volume-preserving neural network (VPNN). In Section 3 we discuss some features of the VPNN model and in Section 4 give the backpropagation equations for the VPNN model.

In Section 5, we demonstrate the utility of the VPNN model by applying it to two disparate types of data using datasets often used to test neural networks: image data (the MNIST dataset of handwritten digits) and text data (the IMDB dataset of movie reviews).

We test the performance of three VPNN variants on these two datasets versus some standard models as well as some mixed models (taking layers from both). Keeping the width equal to the input dimension and considering neural networks of various depths, we compare the accuracy of the model and the amount of learning throughout the layers. Making no effort to tune or tweak the model to fit any specific dataset, VPNN produces
similar or superior accuracy and is superior in terms of amount of learning throughout the layers (i.e. gradients don’t vanish or explode), thus demonstrating the flexibility and wide applicability of the VPNN model.

The width of a network is the maximum number of inputs accepted to a layer of the network. If the width $w$ is roughly constant across a network $d$ layers deep, the number of parameters required in a standard fully-connected network is of order $w^2d$. In our VPNN, the number of parameters required is of order $w \log_2(w)d$, yet despite this significant reduction in the size of the parameter space, we achieve similar or improved results against most benchmarks.

Finally, in Section 6 we discuss some computational and implementation issues, some variations of VPNNs, and mention some future work in progress. The model is implemented in Python as an extension of the PyTorch library, and in this wrap-up section we show how to access the code.

Since being volume preserving is at the core of the model, we begin by reminding the reader of the definition.

**Definition 1** A function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is volume preserving if

$$\text{vol}(f^{-1}(S)) = \text{vol}(S) \quad \text{for all measurable sets } S \subset \mathbb{R}^n$$

(where $\text{vol}(\cdot)$ is the usual (Lebesgue) volume of a set).

### 2. The Basic VPNN Model

The basic $L$ layer VPNN will take $n_{in}$ inputs, process them through $L-1$ volume-preserving layers (the input layer and the hidden layers) and an output layer to produce $n_{out}$ outputs. Each volume-preserving layer (for $l = 1, 2, \ldots, L-1$) is of the form

$$x \rightarrow A(V^{(l)}x + b^{(l)})$$

where $V^{(l)}$ is a volume-preserving linear transformation, $b^{(l)}$ is a bias vector , and $A$ is a volume-preserving coupled activation function.

Being volume preserving necessarily implies being dimension preserving so in $L-1$ volume-preserving layers $V^{(l)}$ is an $n_{in} \times n_{in}$ matrix), $b^{(l)}$ is a vector in $\mathbb{R}^{n_{in}}$, and $A$ is a function from $\mathbb{R}^{n_{in}}$ to itself.

The $L$-th layer (the output layer) is necessarily not volume preserving as it must downsize to the size of the classifier space. In the basic VPNN we implement this by a fixed $n_{out} \times n_{in}$ matrix $Z$ so the output layer is just

$$x \rightarrow Zx.$$

### 2.1 Building the Volume-Preserving Linear Transformations of a VPNN

We build $V$, a volume-preserving linear transformation, as a product of rotation, permutation, and diagonal matrices. We first describe in detail these matrices and then describe how we fit them together in the VPNN architecture.
1. **Rotation Matrices** Let

\[ R_\theta = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \]

(the matrix that rotates a vector in \( \mathbb{R}^2 \) by \( \theta \) in the counterclockwise direction).

Then a rotation matrix \( R \) in a VPNN corresponds to a direct sum of such matrices:

\[ R = \bigoplus_{i=1}^{n_{in}/2} R_{\theta_i} = \begin{bmatrix} R_{\theta_1} & 0 & 0 & \cdots & 0 \\ 0 & R_{\theta_2} & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \cdots & R_{\theta_{n_{in}/2}} \end{bmatrix} \]

There are \( n_{in}/2 \) trainable parameters in a rotation matrix, each parameter is involved in four neuron connections and each input neuron connects to two output neurons.

2. **Permutation Matrices** A permutation matrix \( Q \) in a VPNN corresponds to a permutation \( q \) of \( \{1, 2, 3, \ldots, n_{in}\} \) (a bijection from \( \{1, 2, 3, \ldots, n_{in}\} \) to itself) which is chosen randomly before training begins. So the permutation matrix \( Q \) has \((q(i), i)\) entries (for \( i = 1, 2, \ldots, n_{in} \)) equal to one and all other entries are zero.

There are no trainable parameters in a permutation matrix, and each input neuron connects to one output neuron.

3. **Diagonal Matrices** A diagonal matrix \( D \) in a VPNN has diagonal entries which are positive and have product one. To stay away from possible “division by zero” problems, we implement this as

\[ D = \begin{bmatrix} f(t_1) \\ f(t_{n_{in}}) \\ f(t_2) \\ f(t_{n_{in}}) \\ \vdots \\ f(t_{n_{in}-1}) \\ f(t_{n_{in}}) \\ f(t_{n_{in}-2}) \\ \vdots \\ f(t_{n_{in}-1}) \end{bmatrix} \]

where \( f \) is a function from \( \mathbb{R} \) to \( \mathbb{R}^+ \) whose range lies in some compact interval (and all off-diagonal entries are zero). In our implementation we choose \( f(x) = \exp(\sin x) \).

There are \( n_{in} \) trainable parameters in each diagonal matrix, each parameter is involved in two neuron connections and each input neuron connects to one output neuron.

Then the volume-preserving linear transformation \( V \) is implemented as

\[ V = \left( \prod_{j=1}^{k/2} R_j Q_j \right) D \left( \prod_{j=k/2+1}^{k} R_j Q_j \right) \]

With each \( R_j Q_j \) connecting two input neurons to two “random” output neurons, using only \( \lceil \log_2(n_{in}) \rceil \) such \( R_j Q_j \) along with a diagonal matrix should achieve almost total neuronal interaction in each volume-preserving affine layer. However, testing showed there is
a slight improvement in accuracy when we add additional $R_jQ_j$ to gain some redundant neural connections. So, in the basic VPNN model we set $k$ (the number of rotations or permutations used in any layer) to be

$$k = 2\lceil\log_2(n_{in})\rceil.$$ 

This also ensures $k$ is even and so allows us to have the same number of rotations and permutations on each side of the diagonal (this is not strictly necessary). Not surprisingly, the more layers in the VPNN, the less pronounced is the effect of adding redundant rotations/permutations in any layer. In very deep networks, taking $k$ closer to $\lceil\log_2(n_{in})\rceil$ is probably optimal.

### 2.2 Building the Coupled Activation Functions of a VPNN

A coupled activation function corresponds to a non-affine function $C$ from $\mathbb{R}^2$ to $\mathbb{R}^2$ which is area preserving. Instead of the usual activation functions, which act entrywise on the entries of a vector, a coupled activation function $A$ acts on a vector $x$ (with an even number of entries) by grouping them in pairs and applying $C$ to them pairwise. So a coupled activation sublayer performs

$$x = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} \xrightarrow{A} \begin{bmatrix} C\left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}\right) \\ \vdots \\ \vdots \\ C\left(\begin{bmatrix} x_{n-1} \\ x_n \end{bmatrix}\right) \end{bmatrix}.$$ 

Such functions can be created in many ways. We will mention some other possibilities in Section 6, but for our basic VPNN model we use what we refer to as **coupled Chebyshev functions**.

These functions are most easily described in polar coordinates. Given a point $(x, y)$ in the plane, if $r$ is the distance from that point to $(0, 0)$ and $-\pi < \theta \leq \pi$ is the angle the ray from $(0, 0)$ to $(x, y)$ makes with the positive $x$ axis, then $r = \sqrt{x^2 + y^2}$ and $\theta = \text{sgn}(y) \cos^{-1}\left(\frac{x}{\sqrt{x^2 + y^2}}\right)$ are the polar coordinates of $(x, y)$. We introduce a contractive factor $M$ and define

$$C_M(r, \theta) = \left(\frac{r}{\sqrt{M}}, M\theta\right)$$

so the radius $r$ is contracted by $\sqrt{M}$ and the angle $\theta$ is increased by a factor of $M$. The area unit for polar coordinates is $dA = r \, dr \, d\theta$ so

$$d(C_M(A)) = \frac{r}{\sqrt{M}} \frac{\partial C_M}{\partial r} \frac{\partial C_M}{\partial \theta} dr \, d\theta = \frac{r}{\sqrt{M} \sqrt{M}} M \, dr \, d\theta = r \, dr \, d\theta.$$
Converting to Cartesian coordinates,
\[
C_M \left( \begin{bmatrix} x \\ y \end{bmatrix} \right) = \begin{bmatrix}
\frac{\sqrt{x^2+y^2}}{\sqrt{M}} \\
\frac{\sqrt{x^2+y^2}}{\sqrt{M}} \cos \left( M \cos^{-1} \left( \frac{x}{\sqrt{x^2+y^2}} \right) \right) \\
\frac{\sqrt{x^2+y^2}}{\sqrt{M}} \sin \left( M \cos^{-1} \left( \frac{x}{\sqrt{x^2+y^2}} \right) \right)
\end{bmatrix}
\]

This is the formula we will be using in our coupled activation function, typically with a value of \( M \) in the range \((1, 2]\). Just for interest we mention that, in the case where \( M \) is an even integer, these are related to the famous Chebyshev polynomials:
\[
C_M \left( \begin{bmatrix} x \\ y \end{bmatrix} \right) = \begin{bmatrix}
\frac{\sqrt{x^2+y^2}}{\sqrt{M}} T_M \left( \frac{x}{\sqrt{x^2+y^2}} \right) \\
\frac{|y|}{\sqrt{M}} U_{M-1} \left( \frac{x}{\sqrt{x^2+y^2}} \right)
\end{bmatrix}
\]

where \( T_n \) is Chebyshev polynomial of the first kind, and \( U_n \) is Chebyshev polynomial of the second kind:
\[
T_n(x) = \cos(n \cos^{-1}(x)) \quad \text{and} \quad U_n(x) = \frac{\cos(n \sin^{-1}(x))}{\sin(n \cos^{-1}(x))}
\]

In the case \( M = 2 \) these have particularly nice form:
\[
C_2 \left( \begin{bmatrix} x \\ y \end{bmatrix} \right) = \begin{bmatrix}
\frac{x^2-y^2}{\sqrt{2} \sqrt{x^2+y^2}} \\
\frac{\sqrt{2} x |y|}{\sqrt{2} \sqrt{x^2+y^2}}
\end{bmatrix}
\]

### 2.3 Building the Output Layer of a VPNN

Since volume-preserving layers cannot downsize (reduce dimension) we need some method to map down to the dimension of our classification space. We could use a fully-connected layer, but in the testing that follows we want to demonstrate that the learning is happening in the volume-preserving layers, so our output layer will have no parameters.

We implement this as simply as possible. We use no bias on this layer and fix a “random” matrix \( Z \) of size \( n_{out} \times n_{in} \) with \( ZZ^T = 1 \) and with most entries non-zero and of roughly equal magnitude. (This is chosen to preserve length and connect every output neuron in this layer to every input neuron with roughly the same weight). Then the output layer performs
\[
x \rightarrow Zx.
\]

So, roughly, we are just choosing a random initialization of a weight matrix \( Z \), but not allowing the weights to train in this final layer.

We generate the downsizer matrix \( Z \) by randomly choosing entries of an \( n_{out} \times n_{in} \) matrix \( A \) from the interval \([-1, 1]\), then applying the reduced Singular Value Decomposition to \( A \), we obtain so \( A = U \Sigma V^T \) where \( Z = U \) has the desired properties.

### 3. Discussion of the VPNN Model

The key feature of our neural network is that it is volume preserving in all layers except the output layer. Rotations, permutations, and translations are rigid maps on \( \mathbb{R}^n \) and so
leave volume unchanged. The determinant one condition ensures the diagonal layer is also volume preserving, and the coupled activation maps are also volume preserving. Because of the volume-preserving property, if vectors on the unit ball are sent through a layer, some will be shortened and some lengthened. When composing through multiple hidden layers, we would expect “on average” that a vector will be shortened at some layers and lengthened at others and generally not have its length vanish or explode, thus giving some management of the gradient.

Once being volume preserving was identified as a control mechanism for the gradient, we needed volume-preserving activation functions. Since activation functions are necessarily non-affine, they cannot be constructed as functions of one input variable only. So we had to allow coupled activation functions which take two (or more) inputs.

Next we needed finer control on the weights layer. Our construction is motivated by the Singular Value Decomposition, which states that any square matrix can be written as $UDV$ where $U$ and $V$ are orthonormal (i.e. real unitary) and $D$ is diagonal with non-negative diagonal entries. Any real unitary matrix (of determinant 1) can be written as a product of Givens rotations. Every Givens rotation is of the form $QRQ^{-1}$ for some permutation matrix and some choice of parameters $\theta_i$ (all but one chosen to be zero). Thus it is reasonable that we should be able to replace a general weight matrix $W$ by a volume-preserving matrix $V$ of the above form with little impact on ability to approximate.

In an earlier architecture we considered only allowing rotation matrices of the form $QRQ^{-1}$ where $Q$ is a permutation matrix (rather than the $RQ$ structure in the basic VPNN presented above). In that case, we randomly paired off standard axes and rotated in those standard planes. So why use $RQ$ rather than $QRQ^{-1}$? If we didn’t, we would have a built-in predisposition in our neural network for neurons to link to themselves across layers. The rotation matrices (as well as the coupled activation sublayers) connects two input neurons to two output neurons, and one of the two output neurons is always one of the input neurons. In practice the choice of $QR$ rather than $QRQ^{-1}$ seems to give slight but noticeable improvements in accuracy.

Some other properties of note for VPNNs include:

- In the literature, activation functions are sometimes referred to as *squashing functions*. Functions like sigmoid and hyperbolic tangent “squash” the real line into a bounded interval. Functions like ReLU squash all negative inputs to zero. This ability to squash seems to be a useful feature in activation functions but is also one source of the vanishing gradient problem. The coupled Chebyshev activation function is volume preserving, but by giving up a bit of one-to-oneness, also squashes the length of input vectors. The consistency of its squashing seems to give better performance than other coupled activation functions (such as shears).

- The number of trainable parameters in each of the first $L - 1$ layers of a basic VPNN is $n_{in}(\lceil\log_2 n_{in}\rceil + 2)$ where $n_{in}$ is the number of entries in the input vector to the neural network ($n_{in}\lceil\log_2 n_{in}\rceil$ from rotations, $n_{in}$ from diagonals, and $n_{in}$ from biases). Contrast this to $n_{in}^2 + n_{in}$ in a standard neural network (or even greater if there was upsizing).

- A VPNN has a (mostly) binary neuron linkage structure. Each single rotation matrix, and coupled activation function connects each input neuron to exactly two output
neurons. Each diagonal and permutation matrix connects each input neuron to exactly one output neuron. Figure 1 is a visualization of the neuron connectivity in a single layer of a simplified VPNN where there are four input neurons and $x \rightarrow A(RQDx+b)$.

![Figure 1: A simplified VPNN layer](image)

In Glorot et al. (2011), they illustrate the superior performance of ReLU compared to other activation functions in deep neural networks. One possible explanation for this superior performance, as mentioned in the paper, is the fact that ReLU introduces sparsity. Certain neuronal connections are pruned by virtue of having negative inputs into ReLU. In a VPNN, this sparsity is incorporated by a different mechanic, not by pruning but by building fewer neuronal connections as part of the architecture.

- We scale our input vectors so that their length is within the same order of magnitude of the output vectors (which should have length 1, if the network is learning correctly). In practice we preprocess our inputs by scaling entries so that they lie in some interval (say \([0, 1]\)) and then divide each entry by $\sqrt{n_{in}}$ where $n_{in}$ is the number of entries, so that the length of an input vector is reasonably close to 1. This is often done in neural network models but is particularly important for VPNNs since any stretching or compressing in the basic VPNN must be done in diagonal and activation layers, and we do not want to impose extra work on these layers to scale vectors up or down beyond what is needed for approximation.

- The VPNN must also act on vectors with an even number of entries, as the rotational layers and coupled activation layers require an even number of inputs. If we had an odd number of inputs the simplest solution would be to add one new input which was always zero.

- In a VPNN, when a neuronal connection (i.e. a parameter in the model) is strengthened in any rotational or diagonal sublayer, nearby neuronal connections will be weakened. This mimics the behaviour of biological neural networks (El-Boustani et al., 2018).

- This is a totally heuristic argument, but VPNNs may train more reliably due to the well-behaved nature of the surface of $U_n$ (the unitary $n \times n$ matrices), which is moved
along in gradient descent in rotational sublayers. This should make it less likely to get stuck in local minima.

4. Backpropagation Equations

In this section we give the backpropagation equations for an $L$ layer VPNN as described above. While the formulas are slightly different than that of a standard neural network, the approach to deriving these equations is exactly the same with multiple applications of the chain rule to compute gradients back through the layers.

Let $\theta_{p,i}^{(l)}$ denote the $i$th rotational parameter ($i = 1, 2, \ldots, n_{in}/2$) in the $p$th rotation matrix ($p = 1, 2, \ldots, k$) in the $l$th layer ($l = 1, 2, \ldots, L - 1$) and let $t_j^{(l)}$ denote the $j$th diagonal parameter in the diagonal matrix $D^{(l)}$ in the $l$th layer ($l = 1, 2, \ldots, L - 1$), and let $b_j^{(l)}$ denote the $j$th bias parameter in the bias vector $b^{(l)}$ in the $l$th layer ($l = 1, 2, \ldots, L - 1$).

For a given error function (or cost function) $E$, we need to compute:

for all bias sublayers: $\frac{\partial E}{\partial b_j^{(l)}}$ for $l = 1, 2, \ldots, L - 1$

for all rotational sublayers: $\frac{\partial E}{\partial \theta_{p,i}^{(l)}}$ for $l = 1, 2, \ldots, L - 1$

for all diagonal sublayers: $\frac{\partial E}{\partial t_j^{(l)}}$ for $l = 1, 2, \ldots, L - 1$.

For a single $x_{in} = a^{(0)}$ sent through the network generating output $y_{out} = a^{(L)}$, we use the following terminology for partially forward-computed terms:

$$V_{left}^{(l)} = \prod_{j=1}^{k/2} R_j^{(l)} Q_j^{(l)}, \quad V_{right}^{(l)} = \prod_{j=k/2+1}^{k} R_j^{(l)} Q_j^{(l)}, \quad V^{(l)} = V_{left}^{(l)}D^{(l)}V_{right}^{(l)}$$

$$z^{(l)} = V^{(l)}a^{(l-1)} + b^{(l)} \quad \text{for} \ l = 1, 2, \ldots, L - 1$$

$$a^{(l)} = A(z^{(l)}) \quad \text{for} \ l = 1, 2, \ldots, L - 1$$

$$a^{(L)} = Z a^{(L-1)}.$$

Define (for $l = 1, 2, \ldots, L - 1$)

$$\delta^{(l)} = \frac{\partial E}{\partial z^{(l)}} = \begin{bmatrix} \frac{\partial E}{\partial z_1^{(l)}} \\ \frac{\partial E}{\partial z_2^{(l)}} \\ \vdots \\ \frac{\partial E}{\partial z_{n_{in}}^{(l)}} \end{bmatrix}.$$
Then we have the following backpropagation equations to backpropagate completely through a layer. For any coupled activation function as described above:

For $l = 1, 2, 3, \ldots, L - 1$:

If $j$ is odd

$$\delta_j^{(l)} = \left(V^{(l)} T \delta^{(l+1)}\right)_j \left(\frac{\partial C_1}{\partial x} \bigg|_{x = z_j^{(l)} \atop y = z_{j+1}^{(l)}}\right) + \left(V^{(l)} T \delta^{(l+1)}\right)_{j+1} \left(\frac{\partial C_2}{\partial x} \bigg|_{x = z_j^{(l)} \atop y = z_{j+1}^{(l)}}\right)$$

If $j$ is even

$$\delta_j^{(l)} = \left(V^{(l)} T \delta^{(l+1)}\right)_{j-1} \left(\frac{\partial C_1}{\partial y} \bigg|_{x = z_j^{(l)} \atop y = z_{j+1}^{(l)}}\right) + \left(V^{(l)} T \delta^{(l+1)}\right)_j \left(\frac{\partial C_2}{\partial y} \bigg|_{x = z_j^{(l)} \atop y = z_{j+1}^{(l)}}\right)$$

(where $C_1(x, y)$ is the first component of the coupled activation function and $C_2(x, y)$ is the second component).

In the case of the coupled Chebyshev activation function, these partials simplify quite nicely in terms of previously computed quantities.

$$\frac{\partial C_1}{\partial x} = \frac{1}{x^2 + y^2} [x \ M y] C \begin{pmatrix} x \\ y \end{pmatrix} \quad \frac{\partial C_2}{\partial x} = \frac{1}{x^2 + y^2} [-M y \ x] C \begin{pmatrix} x \\ y \end{pmatrix}$$

$$\frac{\partial C_1}{\partial y} = \frac{1}{x^2 + y^2} [y \ -M x] C \begin{pmatrix} x \\ y \end{pmatrix} \quad \frac{\partial C_2}{\partial y} = \frac{1}{x^2 + y^2} [M x \ y] C \begin{pmatrix} x \\ y \end{pmatrix}.$$
(b) for \( p = k/2 + 1, \ldots k \)

\[
\frac{\partial E}{\partial \theta_p^{(l)}} = \delta^{(l)}_V(\theta_p^{(l)}) T\left( \prod_{j=k/2+1}^{p-1} R_j^{(l)} Q_j^{(l)} \right) Z_i \left( \prod_{j=p}^{k} R_j^{(l)} Q_j^{(l)} \right) a^{(l-1)}
\]

where \( Z_i \) is the matrix with a 1 in the \((2i-1, 2i)\) entry, a \(-1\) in the \((2i, 2i-1)\) entry and all other entries are zero. (Note this is equivalent to inserting into the formula for \( V^{(l)} \), before the location of rotation \( p \), a new matrix which has a \( 2 \times 2 \) rotation matrix \( R_{\pi/2} \) in the block corresponding to parameter \( \theta_{p,i} \) and zeroes elsewhere.)

### 4.1 Variant: Trainable parameters in Coupled Activation Sublayers

It is not much more costly to allow the parameters \( M \) in Coupled Activation Sublayers using coupled Chebyshev functions to be trainable. In fact we could implement the Coupled Activation Layer as

\[
x = \begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_{n-1} \\
x_n
\end{bmatrix} \xrightarrow{A} \begin{bmatrix}
C_{M_1}\left( \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right) \\
\vdots \\
C_{M_{n/2}}\left( \begin{bmatrix} x_{n-1} \\ x_n \end{bmatrix} \right)
\end{bmatrix}
\]

where \( M_1, M_2, \ldots, M_{n/2} \) are \( n/2 \) trainable parameters.

The derivatives with respect to \( M_i \) to be used in the modified backpropagation equations are:

\[
\frac{d}{dM_i} C_{M_i} \left( \begin{bmatrix} x \\ y \end{bmatrix} \right) = \begin{bmatrix}
-\frac{1}{2M_i} \\
-\frac{\theta(x,y)}{2M_i}
\end{bmatrix} C_{M_i} \left( \begin{bmatrix} x \\ y \end{bmatrix} \right)
\]

where

\[
\theta(x,y) = \text{sgn}(y) \cos^{-1} \left( \frac{x}{\sqrt{x^2 + y^2}} \right)
\]

### 5. Testing

To demonstrate the utility of VPNN, we compare its performance in terms of accuracy, training time, and size of gradients throughout the layers.

As mentioned in the introduction, we test on two standard datasets:

1. **Image data.** The MNIST Dataset (LeCun et al., 1999) consisting of images (28 \times 28 pixel greyscale) of 70,000 handwritten digits (60,000 for training and 10,000 for testing). The object is to determine the digit (from \( \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\} \)) from the image. So the input vector has \( n_{in} = 28^2 = 784 \) entries, and the output vector has \( n_{out} = 10 \) entries.

2. **Text data.** The IMDB Dataset (Maas et al., 2011) consisting of 25,000 movie reviews for training and the same number for testing. The object is to determine the sentiment
(positive or negative) from the text. We use preprocessed bag-of-words format provided with the database and remove stopwords (like: an, a, the, this, that, etc) found in the Natural Language Toolkit’s corpus, and then use the 4000 most frequently used remaining words in our bag-of-words. So the input vector has $n_{in} = 4000$ and the output vector has $n_{out} = 2$.

We consider 6 neural network models: three VPNN variants, one standard model for a control, and two mixed models using features of both:

1. **VPNN** The first $L - 1$ layers are volume preserving and made up of rotation, permutation, diagonal, and coupled Chebyshev activation sublayers as described in Section 2, with the number of rotations in each layer equal to $2^\lceil \log_2(n_{in}) \rceil$ and the Chebyshev parameter set to $M = 2$.

2. **VPNN1.3** The first $L - 1$ layers are volume preserving and made up of rotation, permutation, diagonal, and coupled Chebyshev activation sublayers as described in Section 2, with the number of rotations in each layer equal to $2^\lceil \log_2(n_{in}) \rceil$ and the Chebyshev parameter set to $M = 1.3$.

3. **VPNNt** The first $L - 1$ layers are volume preserving and made up of rotation, permutation, diagonal, and coupled Chebyshev activation sublayers as described in Section 2, with the number of rotations in each layer equal to $2^\lceil \log_2(n_{in}) \rceil$ but the Chebyshev parameters are trainable as described in Subsection 4.1.

4. **S-RelU** The first $L - 1$ layers use a standard affine sublayer, $x \rightarrow Wx + b$ followed by a ReLU activation function. (We considered also testing this model with a sigmoid activation function, however training was problematic due to vanishing gradients.)

5. **Mixed1** The first $L - 1$ layers use a standard affine sublayer, $x \rightarrow Wx + b$ but use coupled Chebyshev activation sublayers with $M = 1.3$.

6. **Mixed2** The first $L - 1$ layers are volume preserving and made up of rotation, permutation and diagonal sublayers as described in Section 2, with the number of rotations in each layer equal to $2^\lceil \log_2(n_{in}) \rceil$, but the activation function is ReLU.

Some specifics of the implementation of the testing are:

1. **Method**: We use Stochastic Gradient Descent with momentum set to 0.9 and with a batch size of 100 in all training.

2. **Layers**: For ease of comparison, all the models we consider have $L - 1$ layers of the same type which preserve dimension (so the number of neurons in each of the first $L$ layers is equal to $n_{in}$, the number of input neurons) followed by a fixed downsizer matrix $Z$ as in the basic VPNN model. For testing accuracy we take $L = 4$ and for testing learning throughout the layers we take $L = 10$.

3. **Learning Rate**: The surface of the error function seems to be smoother, and generally less steep for VPNN models than for standard models. This allows us to take larger step sizes (learning rates) at the start of training but causes slower convergence.
(for the same learning rate) later in training. When testing for accuracy \((L = 4)\), to accommodate for this and speed up training, we perform a variation of adaptive learning methods: we perform some preliminary runs (with a small number of batches) with larger learning rates to determine stability, and choose initial learning rate of 1/10 of the limit where training seems stable. So for the first half of the training, the learning rates are in the range of 0.1 to 1.0 and then as we have supposedly zeroed in on the minimum, the learning rate is set to 0.01 for all models. When testing for learning throughout the layers \((L = 10)\), we hold the learning rate at 0.01 for all models.

4. **Error Function:** We use the cross-entropy loss function

\[
E(y, \hat{y}) = -\sum_{i} \hat{y}_i \log(y_i)
\]

(where \(y\) is the predicted output for input \(x\) and \(\hat{y}\) is the actual output) for the error function.

5.1 Testing Accuracy and Training Times on a Four-Layer Neural Network

Using a four layer network and running 30 epochs for MNIST and 40 epochs for IMDB, we obtain the training times and accuracy rates as shown in Table 1.

| Model    | MNIST         | IMDB         |
|----------|---------------|--------------|
|          | Training Time | Accuracy     | Training Time | Accuracy     |
| VPNN     | 29 s/epoch    | 98.06 %      | 27 s/epoch    | 86.89 %      |
| VPNN1.3  | 29 s/epoch    | 97.21 %      | 27 s/epoch    | 87.46 %      |
| VPNNt    | 29 s/epoch    | 97.38 %      | 27 s/epoch    | 83.89 %      |
| S-ReLU   | 6 s/epoch     | 97.42 %      | 14 s/epoch    | 86.35 %      |
| Mixed1   | 7 s/epoch     | 98.40 %      | 15 s/epoch    | 87.16 %      |
| Mixed2   | 27 s/epoch    | 96.00 %      | 25 s/epoch    | 83.90 %      |

Table 1: Training Time and Accuracy

Figures 2 and 3 (for MNIST) and Figures 4 and 5 (for IMDB) show the progression of the accuracy throughout the training.
Some comments on accuracy:

- All the models perform comparably well, and very close to the state of the art for these classification tasks (approximately 99% for MNIST, and approximately 88% for IMDB using the best bag-of-words approach). The training times are also comparable, which may be a bit surprising considering all the trigonometric evaluations in the VPNN model.

- The swings in accuracy early in the training are due to the large learning rate. This could obviously be smoothed with a smaller learning rate (and thus more epochs).

- VPNN seems to be the superior volume-preserving neural net, with VPNNt being the least accurate. This is somewhat surprising as there is more “freedom” due to additional parameters in the VPNNt model. It may have something to do with the different types of parameters (rotational, bias, diagonal and Chebychev). This seems
to cause slow training of the Chebychev parameters in particular. The VPNNt accuracy is still trending upwards after most other models have levelled off. Running significantly more epochs does improve VPNNt performance.

- The best model overall for accuracy is Mixed1. This model incorporates some of the best features from both models: the significantly increased parameter space of the standard model, and the gradient control of the coupled Chebychev activation functions of VPNN models.

As Figures 6 and 7 show, all models also show similar performance in terms of minimizing the error function as well. Once again, VPNNt is lagging behind but still trending downwards and improves with additional training.

One factor that should be taken into consideration is the number of parameters in the various models. Fully-connected layer models (like S-ReLU) use $w(w + 1)$ parameters per layer of width $w$, versus $w(\lceil \log_2(w) \rceil + 2)$ parameters per layer VPNN2 and VPNN1.3 (or $w(\lceil \log_2(w) \rceil + 5/2)$ for VPNNt).

For our models $w = n_{in}$, and Table 2 shows the number of parameters per layer for the different models:

| Model     | MNIST  | IMDB   |
|-----------|--------|--------|
| VPNN      | $9.4 \times 10^3$ | $5.6 \times 10^4$ |
| VPNN1.3   | $9.4 \times 10^4$ | $5.6 \times 10^4$ |
| VPNNt     | $9.8 \times 10^4$ | $5.8 \times 10^4$ |
| S-ReLU    | $6.2 \times 10^3$ | $1.6 \times 10^4$ |
| Mixed1    | $6.2 \times 10^5$ | $1.6 \times 10^4$ |
| Mixed2    | $9.4 \times 10^3$ | $5.6 \times 10^4$ |

Table 2: Parameters per layer

Figure 6: Error Function: MNIST
Figure 7: Error Function: IMDB
Especially for datasets where each datapoint has a large number of entries, the number of parameters is dramatically lower for VPNNs than for standard neural networks.

5.2 Testing Learning Throughout the Layers on a Ten-Layer Neural Network

In the title of this paper we claim VPNNs are a solution to the vanishing gradient problem. This is the section where we perform the testing and collect the evidence to back up this claim.

We consider the amount of learning throughout the layers for the various models. This will show how well the VPNNs control the gradient in deep neural networks and allow for learning in all layers roughly equally. The magnitude of the vectors $\delta^{(l)}$ are a measure this, as they indicate how well the parameter updating has propagated back to the $l$-th layer. If we have vanishing gradients, we would expect $\|\delta^{(l)}\|$ to be small for early layers ($l$ close to 1) compared to later $\|\delta^{(l)}\|$ ($l$ close to $L$) as the training progresses. If we have exploding gradient we expect the reverse. If all are comparable in size, we have ideal backpropagation.

For testing learning throughout the layers we use deeper neural networks. We set $L = 10$ layers so there are 9 layers of volume-preserving or standard type, followed by a fixed matrix downsizer output layer. Since we aren’t testing the accuracy here, we run 3 epochs only and collect the norms of the vectors $\delta^{(l)}$ at this stage.

As it is the comparison of the order of magnitude (rather than the exact value) of the gradients across the layers which is relevant, we consider the $\log_{10}$ of the learning amount in each layer compared to $\log_{10}$ of the learning amount in the final layer for each of the models, so we are plotting

$$y = \log_{10} \left( \frac{\|\delta^{(l)}\|}{\|\delta^{(L)}\|} \right) = \log_{10} \left( \|\delta^{(l)}\| \right) - \log_{10} \left( \|\delta^{(L)}\| \right)$$

for $l = 1, 2, 3, \ldots, L$. (So, for a given $l$, 10 raised to the corresponding value of $y$ gives the percentage more (or less) of learning in that layer as compared to layer $L$.

Figures 8 (for MNIST) and 9 (for IMDB) display the data from these runs.
In Figures 8 and 9, a positive slope indicates vanishing gradients. More precisely, a slope of \( m \) on these graphs indicates that learning decreases (when \( m \) is positive) or increases (when \( m \) is negative) by a factor of \( 10^{-m} \) for each layer deeper we go into the neural network. For S-ReLU in both Figure 8 and Figure 9, the slope is approximately 0.4 so for every layer retreat into the network, the gradients (and hence the amount of learning) decrease by (approximately) a factor of \( 10^{-0.40} = 0.40 \). So in layer 1, there is roughly \((0.40)^9 \approx 2.6 \times 10^{-4}\) as much learning as in layer 10. Almost all the learning is in the late layers. Contrast this with VPNN models, where learning is comparable across the layers, and in fact there is slightly more learning in early layers than in late layers. (The learning throughout the layers for VPNN seems to be missing in Figure 9, in fact it is basically identical to that for VPNNt, and is obscured by that line.) The mixed models show learning throughout the layers superior to S-ReLU but inferior to the VPNN models.

The VPNN variants show clearly superior learning throughout the layers, with no vanishing gradient as compared to standard neural networks.

While we were not testing accuracy in the ten-layer network, we will mention for interest’s sake that the ten-layer VPNNs do train relatively accurately if training is allowed to continue. Since the corresponding four-layer neural networks train close to the standard in both the MNIST and IMDB cases, and we are adding additional complexity, it is perhaps not surprising that accuracy does drop by 5% to 10% from the four-layer to ten-layer VPNN in these cases. While VPNN was the most accurate in the four-layer network, VPNN1.3 is the most accurate in the ten-layer case. This is perhaps due to the fact that with additional layers, less “squashing” must be done in any particular layer.

6. Wrap-up

There are a number of other possible variants of the VPNN model besides the one mentioned above (trainable Chebyshev parameters). One could:

1. Use a different coupled activation function. Any area preserving function from \( \mathbb{R}^2 \) to \( \mathbb{R}^2 \) should work. One possibility is to use shears. Given two functions \( f \) and \( g \) from \( \mathbb{R} \) to \( \mathbb{R} \), we can construct two shears \( \Sigma_{\text{vert}} \) and \( \Sigma_{\text{hor}} \) from \( \mathbb{R}^2 \) to \( \mathbb{R}^2 \) by shearing vertically by \( f \) and/or horizontally by \( g \)

\[
\Sigma_{\text{vert}} (x, y) = (x, f(x) + y) \quad \text{and} \quad \Sigma_{\text{hor}} (x, y) = (x + g(y), y).
\]

It can be easily verified that such maps are area preserving and so could be used to construct coupled activation functions. We experimented with such coupled activation functions, mainly when \( f \) and/or \( g \) were sigmoids or a Henon function. While there was some success, the lack of the consistent “squashing” seemed be the main reason they didn’t achieve the accuracy of the coupled Chebyshev function.

One advantage of using shears is that in that case we were able to prove a Universality Theorem (in the sigmoid case) by approximating any standard neural network as part of a larger VPNN.

2. Introduce a tripled activation function by choosing a volume-preserving function from \( \mathbb{R}^3 \) to \( \mathbb{R}^3 \) and grouping the input vector into triplets.
3. Choose an alternate parametrization for diagonal sublayers. We experimented here and didn’t see much difference. It is possible that the diagonal layers are not that essential. Initially, we thought they would be needed for stretching and shrinking, and their inclusion definitely has an effect on accuracy of the VPNN, but not a major effect. Why not? Perhaps because the coupled Chebyshev activation function performs a similar stretching and shrinking, only non-linearly and instead of in standard directions, in normal and tangential directions.

4. Introduce a fold in the coupled Chebyshev activation functions. If $M$ is not an even integer, $C_M$ is not continuous on $\{(0, y) : y \leq 0\}$. By incorporating a fold $F(x, y) = (x, |y|)$ (a reflection about the $x$ axis) after $C_M$ (so $|y|$ is replaced by $y$, and $\text{sgn}(y)$ removed in the formulas in Subsection 2.2), we can make this activation function continuous and still volume preserving. Perhaps surprisingly, in our experiments, the fold’s presence or absence seems to make little difference to the trainability, accuracy, and reliability of the neural network.

5. Combine with other established neural network layer components: pooling, convolution, etc, as specific to the dataset. In particular the input and output layers, which were non-existent or rudimentary in basic VPNN, could be customized to the dataset under consideration.

6. Replace the product of rotations and permutations by a general unitary. A method of Tagare (Tagare, 2011) to inexpensively move along the gradient in the tangent space of the unitary surface was used by (Wisdom et al., 2016) in a neural network context. One advantage of our model is that we have finer control over the parameter space. In the full unitary models the number of parameters per layer is still on the order of $n^2_m$.

In the implementation of the VPNN there are technical details we glossed over. A few of these are:

1. Some of the above backpropagation formulas look daunting and may seem to be overly computationally intensive, but most of the values needed for the backward pass were already computed on the forward pass and are saved and reused.

2. Depending on the hardware, a call to a cosine or sine function (of which there are many while training a VPNN) usually costs about 20 times as much as a multiplication. Each parameter update typically calls two sines or cosines. There is obviously an additional cost to training a VPNN compared to a standard neural network, but that cost scales linearly with the size of the network and is mitigated by the greatly reduced size of the parameter space.

3. In the coupled Chebyshev activation layer, we have to concern ourselves with possible division by zero. We address this by doing check of any input $(x, y)$ to a Chebyshev function. If $|x| + |y| < 10^{-7}$, we set $(x, y)$ to $(10^{-7}, 0)$.

4. Our code is implemented in Python to run with PyTorch version 1.1. We ran our tests on a consumer grade desktop computer with a 3gb GPU running the Ubuntu 16.04 operating system.
6.1 Future Work

The basic VPNN model here was stripped down to its essentials for the purposes of demonstrating the efficacy of the model. In practice it should be another tool in the machine learning toolbox, used in conjunction with other approaches and techniques to achieve best results. That is one of our main goals for future work. Now that we have the basic model, we plan to consider variations to tackle different applications. Of particular interest for future work will be developing VPNNs into feasible models to handle problems with sequential data with long-term dependencies.

We would also like to apply our model to extremely large datasets (in terms of the number of entries in the inputs to the neural network). In such a scenario, the almost linear scaling of the size of the parameter space (versus quadratic scaling for the standard models) for similar performance should be advantageous.

6.2 Accessing the code

The code for the VPNN architecture is accessible in Github repository (MacDonald et al. (2019)).

Acknowledgments

The first author acknowledges the support of NSERC Canada. The third and fourth author acknowledge the support of NSERC Canada through the NSERC USRA program.

References

Martin Arjovsky, Amar Shah, and Yoshua Bengio. Unitary evolution recurrent neural networks. volume 48, pages 1120–1128, 2016.

Yoshua Bengio, Patrice Simard, Paolo Frasconi, et al. Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2):157–166, 1994.

Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Sami El-Boustani, Jacque PK Ip, Vincent Breton-Provencher, Graham W Knott, Hiroyuki Okuno, Haruhiko Bito, and Mriganka Sur. Locally coordinated synaptic plasticity of visual cortex neurons in vivo. *Science*, 360(6395):1349–1354, 2018.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323, 2011.

Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel-rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Brian Kingsbury, et al. Deep neural
networks for acoustic modeling in speech recognition. *IEEE Signal processing magazine*, 29, 2012.

Sepp Hochreiter. Untersuchungen zu dynamischen neuronalen netzen. *Diploma, Technische Universität München*, 91(1), 1991.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

Li Jing, Yichen Shen, Tena Dubcek, John Peurifoy, Scott Skirlo, Yann LeCun, Max Tegmark, and Marin Soljačić. Tunable efficient unitary neural networks (EUNN) and their application to RNNs. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1733–1741. JMLR. org, 2017.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

Yan LeCun et al. The MNIST dataset of handwritten digits (images). 1999.

Andrew L Maas, Raymond E Daly, Peter T Pham, Dan Huang, Andrew Y Ng, and Christopher Potts. Learning word vectors for sentiment analysis. In *Proceedings of the 49th annual meeting of the association for computational linguistics: Human language technologies-volume 1*, pages 142–150. Association for Computational Linguistics, 2011.

Gordon MacDonald, Andrew Godbout, Bryn Gillcash, and Stephanie Cairns. Python (Pytorch) code for VPNN. https://github.com/andrewgodbout/VPNN_pytorch, 2019.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

Jürgen Schmidhuber. Learning complex, extended sequences using the principle of history compression. *Neural Computation*, 4(2):234–242, 1992.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

Hemant D Tagare. Notes on optimization on Stiefel manifolds. Technical report, Technical report, Yale University, 2011.

Scott Wisdom, Thomas Powers, John Hershey, Jonathan Le Roux, and Les Atlas. Full-capacity unitary recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 4880–4888, 2016.