

Volume-preserving Neural Networks

Gordon MacDonald*, Andrew Godbout*, Bryn Gillcash* and Stephanie Cairns[§]

**School of Mathematical and Computational Sciences, University of Prince Edward Island, Charlottetown, Canada*
{gmacdonald@upei.ca, agodbout@upei.ca, bwgillcash@upei.ca}

[§]*Department of Mathematics and Statistics, McGill University, Montreal, 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. Our approach replaces the standard weight matrix of a neural network with a combination of diagonal, rotational and permutation matrices, all of which are volume-preserving. We introduce a coupled activation function allowing us to preserve volume even in the activation function portion of a neural network layer. This control on the volume forces the gradient (on average) to maintain equilibrium and not explode or vanish. To demonstrate our architecture we apply our volume-preserving neural network model to two standard datasets.

I. 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 [1] 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 [2], speech recognition [3], and natural language processing [4]. In deep neural networks trained via gradient descent methods with backpropagation, the problem of vanishing gradients [5], [6], [7] 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 first author acknowledges the support of NSERC Canada. The third and fourth author acknowledge the support of NSERC Canada through the NSERC USRA program.

There have been a number of approaches to address the vanishing (or exploding gradient) problem. These techniques include using alternative activation functions (such as ReLU) [8], clipping gradients [9], long short-term memory (LSTM) units [10], gated recurrent units (GRU) [4], multi-level hierarchies [11] and orthogonal constraints on parameter initializations [12].

There are a number of approaches that focus on adjustments to the standard weight matrices of the network such as the use of orthogonal or unitary constraints [13]–[16]. The approaches in [15] and [16] enforce the weight matrices within a recurrent neural network to be unitary and achieve this by parameterizing the weight matrix as a product of unitary building blocks (such as a combination of diagonal, permutation and reflection matrices). Although these approaches are applied to recurrent neural networks they share similarities with our approach and especially in their parameterization of the weight matrix to ensure it is unitary.

Our approach proposes adjustments to the the activation functions and the weight matrices, by replacing each of 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 with 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.

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 number of parameters as compared to a fully-connected network our results are similar in our testing.

Since being *volume preserving* is at the core of the model, we begin by reminding the reader of the definition. A function $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is volume-preserving if $\text{vol}(f^{-1}(S)) = \text{vol}(S)$ for all measurable sets $S \subset \mathbb{R}^n$ (where $\text{vol}(\cdot)$ is the usual (Lebesgue) volume of a set).

arXiv:1911.09576v3 [cs.LG] 26 Apr 2021

II. 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 $l = 1, 2, \dots, L - 1$) is of the form:

$$\mathbf{x} \rightarrow A(V^{(l)}\mathbf{x} + \mathbf{b}^{(l)}) \quad (1)$$

where $V^{(l)}$ is a volume-preserving linear transformation, $\mathbf{b}^{(l)}$ 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, $\mathbf{b}^{(l)}$ 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:

$$\mathbf{x} \rightarrow Z\mathbf{x}. \quad (2)$$

A. 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.

Let:

$$R_\theta = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (3)$$

(the matrix that rotates a vector in \mathbb{R}^2 by θ in the counter-clockwise direction). Then a rotation matrix R in a VPNN corresponds to a direct sum of such matrices:

$$R = \bigoplus_{i=1}^{n_{in}/2} R_{\theta_i} = \begin{bmatrix} R_{\theta_1} & 0 & 0 & \cdots & 0 \\ 0 & R_{\theta_2} & 0 & \cdots & 0 \\ 0 & 0 & R_{\theta_3} & & \\ \vdots & \vdots & & \ddots & \\ 0 & 0 & & & R_{\theta_{n_{in}/2}} \end{bmatrix}. \quad (4)$$

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.

A permutation matrix Q in a VPNN corresponds to a permutation q of $\{1, 2, 3, \dots, n_{in}\}$ (a bijection from $\{1, 2, 3, \dots, n_{in}\}$ to itself) which is chosen randomly before training begins. So the permutation matrix Q has $(q(i), i)$ entries (for $i = 1, 2, \dots, n_{in}$) 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.

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

$$D = \begin{bmatrix} \frac{f(t_1)}{f(t_{n_{in}})} & & & & \\ & \frac{f(t_2)}{f(t_1)} & & & \\ & & \ddots & & \\ & & & \frac{f(t_{n_{in}-1})}{f(t_{n_{in}-2})} & \\ & & & & \frac{f(t_{n_{in}})}{f(t_{n_{in}-1})} \end{bmatrix} \quad (5)$$

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

$$V = \left(\prod_{j=1}^{k/2} R_j Q_j \right) D \left(\prod_{j=k/2+1}^k R_j Q_j \right) \quad (6)$$

with each $R_j Q_j$ connecting two input neurons to two ‘‘random’’ output neurons, using $\lceil \log_2(n_{in}) \rceil$ 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 $R_j Q_j$ 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:

$$k = 2 \lceil \log_2(n_{in}) \rceil. \quad (7)$$

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.

B. 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 \mathbf{x} (with an even number of entries) by grouping them in pairs and applying C to them pairwise. So a coupled activation sublayer performs:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} \xrightarrow{A} \begin{bmatrix} C \left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right) \\ \vdots \\ \vdots \\ C \left(\begin{bmatrix} x_{n-1} \\ x_n \end{bmatrix} \right) \end{bmatrix}. \quad (8)$$

Such functions can be created in many ways, 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:

$$C_M(r, \theta) = \left(\frac{r}{\sqrt{M}}, M\theta \right) \quad (9)$$

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

$$d(C_M(A)) = \frac{r}{\sqrt{M}} \frac{\partial C_M}{\partial r} \frac{\partial C_M}{\partial \theta} dr d\theta \quad (10)$$

$$= \frac{r}{\sqrt{M}} \frac{1}{\sqrt{M}} M dr d\theta = r dr d\theta. \quad (11)$$

Converting to Cartesian coordinates,

$$C_M \left(\begin{bmatrix} x \\ y \end{bmatrix} \right) = \begin{bmatrix} \frac{\sqrt{x^2 + y^2}}{\sqrt{M}} \cos \left(M \cos^{-1} \left(\frac{x}{\sqrt{x^2 + y^2}} \right) \right) \\ \frac{\sqrt{x^2 + y^2}}{\sqrt{M}} \text{sgn}(y) \sin \left(M \cos^{-1} \left(\frac{x}{\sqrt{x^2 + y^2}} \right) \right) \end{bmatrix}. \quad (12)$$

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:

$$C_M \left(\begin{bmatrix} x \\ y \end{bmatrix} \right) = \begin{bmatrix} \frac{\sqrt{x^2 + y^2}}{\sqrt{M}} T_M \left(\frac{x}{\sqrt{x^2 + y^2}} \right) \\ \frac{|y|}{\sqrt{M}} U_{M-1} \left(\frac{x}{\sqrt{x^2 + y^2}} \right) \end{bmatrix} \quad (13)$$

where T_n is Chebyshev polynomial of the first kind ($T_n(x) = \cos(n \cos^{-1}(x))$), and U_n is Chebyshev polynomial of the second kind ($U_n(x) = \frac{\cos(n \sin^{-1}(x))}{\sin(\cos^{-1}(x))}$). In the case $M = 2$ these have particularly nice form:

$$C_2 \left(\begin{bmatrix} x \\ y \end{bmatrix} \right) = \left[\frac{x^2 - y^2}{\sqrt{2}\sqrt{x^2 + y^2}}, \frac{\sqrt{2}x|y|}{\sqrt{x^2 + y^2}} \right]. \quad (14)$$

C. 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 trainable 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:

$$\mathbf{x} \rightarrow Z\mathbf{x}. \quad (15)$$

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.

III. 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 θ_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.

The number of trainable parameters in each of the first $L-1$ layers of a basic VPNN is $n_{in}([\log_2 n_{in}] + 2)$ where n_{in} is the number of entries in the input vector to the neural network ($n_{in}[\log_2 n_{in}]$ 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).

In [17], 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

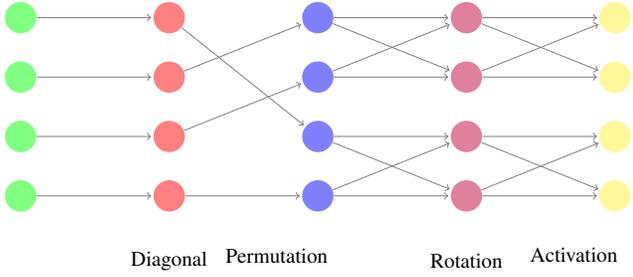


Fig. 1. A simplified VPNN layer

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.

IV. BACKPROPAGATION EQUATIONS

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

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

For a single $\mathbf{x}_{in} = \mathbf{a}^{(0)}$ sent through the network generating output $\mathbf{y}_{out} = \mathbf{a}^{(L)}$, we use the following terminology for

partially forward-computed terms:

$$V_{\text{left}}^{(l)} = \prod_{j=1}^{k/2} R_j^{(l)} Q_j^{(l)}, \quad (16)$$

$$V_{\text{right}}^{(l)} = \prod_{j=k/2+1}^k R_j^{(l)} Q_j^{(l)}, \quad (17)$$

$$V^{(l)} = V_{\text{left}}^{(l)} D^{(l)} V_{\text{right}}^{(l)} \quad (18)$$

$$\mathbf{z}^{(l)} = V^{(l)} \mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \quad \text{for } l = 1, 2, \dots, L-1 \quad (19)$$

$$\mathbf{a}^{(l)} = A(\mathbf{z}^{(l)}) \quad \text{for } l = 1, 2, \dots, L-1 \quad (20)$$

$$\mathbf{a}^{(L)} = Z\mathbf{a}^{(L-1)}. \quad (21)$$

$$(22)$$

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

$$\delta^{(l)} = \frac{\partial E}{\partial \mathbf{z}^{(l)}} = \begin{bmatrix} \frac{\partial E}{\partial z_1^{(l)}} \\ \frac{\partial E}{\partial z_2^{(l)}} \\ \vdots \\ \frac{\partial E}{\partial z_{n_{in}}^{(l)}} \end{bmatrix}. \quad (23)$$

Then we have the following backpropagation equations to backpropagate completely through a layer. For any coupled activation function as described above:

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

If j is odd:

$$\delta_j^{(l)} = \left(V^{(l)T} \delta^{(l+1)} \right)_j \left(\frac{\partial C_1}{\partial x} \Big|_{\substack{x = z_j^{(l)} \\ y = z_{j+1}^{(l)}}} \right) + \left(V^{(l)T} \delta^{(l+1)} \right)_{j+1} \left(\frac{\partial C_2}{\partial x} \Big|_{\substack{x = z_j^{(l)} \\ y = z_{j+1}^{(l)}}} \right) \quad (24)$$

If j is even:

$$\delta_j^{(l)} = \left(V^{(l)T} \delta^{(l+1)} \right)_{j-1} \left(\frac{\partial C_1}{\partial y} \Big|_{\substack{x = z_{j-1}^{(l)} \\ y = z_j^{(l)}}} \right) + \left(V^{(l)T} \delta^{(l+1)} \right)_j \left(\frac{\partial C_2}{\partial y} \Big|_{\substack{x = z_{j-1}^{(l)} \\ y = z_j^{(l)}}} \right) \quad (25)$$

(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.

$$\begin{aligned}\frac{\partial C_1}{\partial x} &= \frac{1}{x^2 + y^2} [x \quad My] C \left(\begin{bmatrix} x \\ y \end{bmatrix} \right) \\ \frac{\partial C_2}{\partial x} &= \frac{1}{x^2 + y^2} [-My \quad x] C \left(\begin{bmatrix} x \\ y \end{bmatrix} \right) \\ \frac{\partial C_1}{\partial y} &= \frac{1}{x^2 + y^2} [y \quad -Mx] C \left(\begin{bmatrix} x \\ y \end{bmatrix} \right) \\ \frac{\partial C_2}{\partial y} &= \frac{1}{x^2 + y^2} [Mx \quad y] C \left(\begin{bmatrix} x \\ y \end{bmatrix} \right).\end{aligned}$$

The following equations allow us to backpropagate through sublayers of a layer. In all (non-output) layers $l = 1, 2, \dots, L-1$ the bias parameters have partials:

$$\frac{\partial E}{\partial b_j^{(l)}} = \delta_j^{(l)}. \quad (26)$$

In the (non-output) layers $l = 1, 2, \dots, L-1$ the diagonal parameters have partials:

$$\frac{\partial E}{\partial t_j^{(l)}} = \delta^{(l)T} \left(V_{\text{left}}^{(l)} F_j^{(l)} V_{\text{right}}^{(l)} \right) a^{(l-1)}, \quad (27)$$

where $F_j^{(l)}$ is a diagonal matrix (of same size as $D^{(l)}$) whose j -th diagonal entry is $f'(t_j^{(l)})$ and whose $j+1$ -th diagonal entry (modulo n_{in}) is $-\frac{f'(t_{j+1}^{(l)})}{f'(t_j^{(l)})^2} f'(t_j^{(l)})$ and all other diagonal entries are zero. In the layers $l = 1, 2, \dots, L-1$, the rotational parameters have partials, for $p = 1, 2, \dots, k/2$ then:

$$\begin{aligned}\frac{\partial E}{\partial \theta_{p,i}^{(l)}} &= \\ \delta^{(l)T} \left(\prod_{j=1}^{p-1} R_j^{(l)} Q_j^{(l)} \right) Z_i \left(\prod_{j=p}^{k/2} R_j^{(l)} Q_j^{(l)} \right) D^{(l)} V_{\text{right}}^{(l)} a^{(l-1)}.\end{aligned} \quad (28)$$

For $p = k/2 + 1, \dots, k$ then:

$$\begin{aligned}\frac{\partial E}{\partial \theta_p^{(l)}} &= \\ \delta^{(l)T} V_{\text{left}}^{(l)} D^{(l)} \left(\prod_{j=k/2+1}^{p-1} R_j^{(l)} Q_j^{(l)} \right) Z_i \left(\prod_{j=p}^k R_j^{(l)} Q_j^{(l)} \right) a^{(l-1)}\end{aligned} \quad (29)$$

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 2×2 rotation matrix $R_{\pi/2}$ in the block corresponding to parameter $\theta_{p,i}$ and zeroes elsewhere.)

A. 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:

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ \vdots \\ x_{n-1} \\ x_n \end{bmatrix} \xrightarrow{A} \begin{bmatrix} C_{M_1} \left(\begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \right) \\ \vdots \\ \vdots \\ C_{M_{n/2}} \left(\begin{bmatrix} x_{n-1} \\ x_n \end{bmatrix} \right) \end{bmatrix} \quad (30)$$

where $M_1, M_2, \dots, M_{n/2}$ are $n/2$ trainable parameters.

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

$$\frac{d}{dM_i} C_{M_i} \left(\begin{bmatrix} x \\ y \end{bmatrix} \right) = \begin{bmatrix} -\frac{1}{2M_i} & -\theta_{(x,y)} \\ \theta_{(x,y)} & -\frac{1}{2M_i} \end{bmatrix} C_{M_i} \left(\begin{bmatrix} x \\ y \end{bmatrix} \right) \quad (31)$$

where:

$$\theta_{(x,y)} = \text{sgn}(y) \cos^{-1} \left(\frac{x}{\sqrt{x^2 + y^2}} \right). \quad (32)$$

V. TESTING

To demonstrate the utility of VPNN, we compare its performance in terms of accuracy, training time, and size of gradients throughout the layers on two standard classification datasets:

- 1) **Image data.** The *MNIST Dataset* [18] consisting of images (28×28 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* [19] 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 II, with the number of rotations in each layer equal to $2 \lceil \log_2(n_{in}) \rceil$ and the Chebyshev parameter set to $M = 2$.

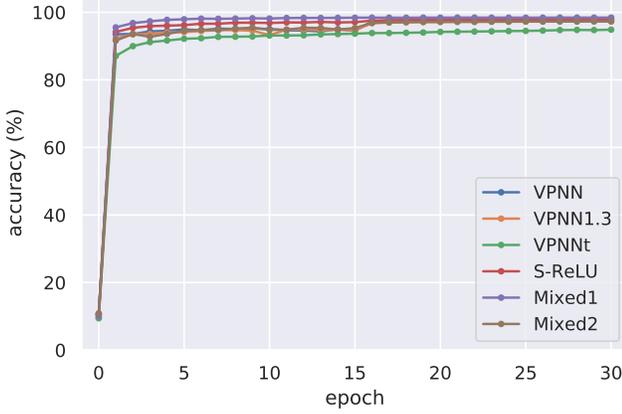


Fig. 2. Accuracy:MNIST

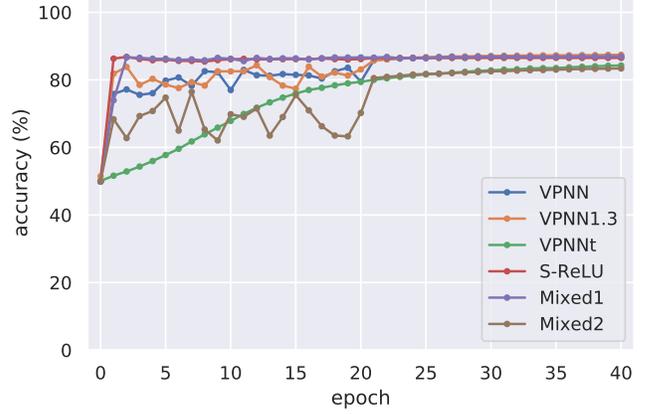


Fig. 4. Accuracy:IMDB

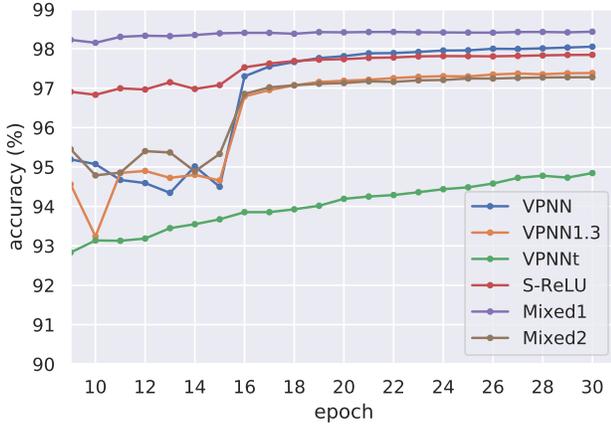


Fig. 3. Zoomed Accuracy:MNIST

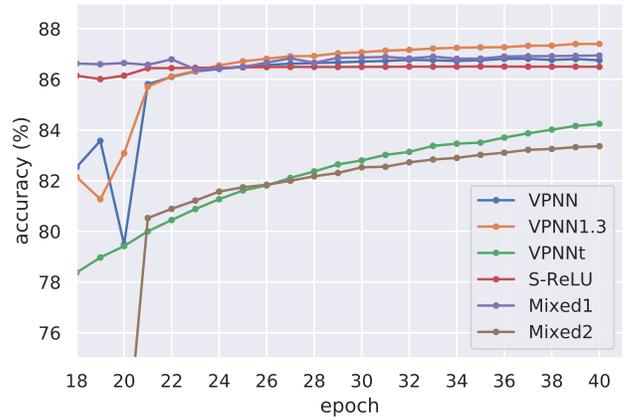


Fig. 5. Zoomed Accuracy:IMDB

- 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 II, 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 II, 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 IV-A.
- 4) **S-ReLU** The first $L - 1$ layers use a standard affine sublayer, $\mathbf{x} \rightarrow W\mathbf{x} + \mathbf{b}$ 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, $\mathbf{x} \rightarrow W\mathbf{x} + \mathbf{b}$ 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 II, 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

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 I
TRAINING TIME AND ACCURACY

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

$$E(\mathbf{y}, \hat{\mathbf{y}}) = - \sum_i \hat{y}_i \log(y_i) \quad (33)$$

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

A. 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 I.

Fig. 2 and Fig. 3 (for MNIST) and Fig. 4 and Fig. 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

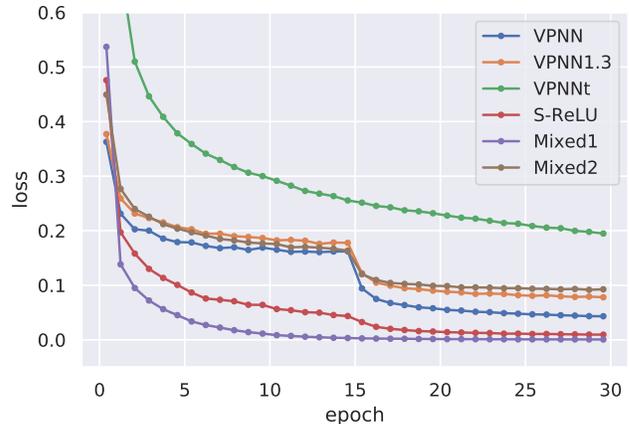


Fig. 6. Error Function: MNIST

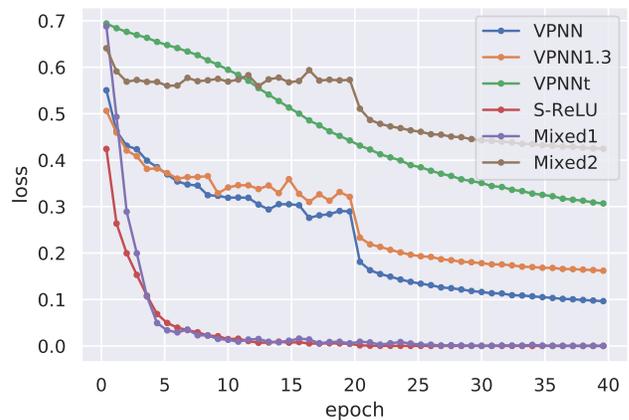


Fig. 7. Error Function: IMDB

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 II shows the number of parameters per layer for the different models:

Especially for datasets where each datapoint has a large number of entries, the number of parameters is dramatically

Model	MNIST	IMDB
VPNN	9.4×10^3	5.6×10^4
VPNN1.3	9.4×10^3	5.6×10^4
VPNNt	9.8×10^3	5.8×10^4
S-ReLU	6.2×10^5	1.6×10^7
Mixed1	6.2×10^5	1.6×10^7
Mixed2	9.4×10^3	5.6×10^4

TABLE II
PARAMETERS PER LAYER

lower for VPNNs than for standard neural networks.

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

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

$$y = \log_{10} \left(\frac{\|\delta^{(l)}\|}{\|\delta^{(L)}\|} \right) = \log_{10} (\|\delta^{(l)}\|) - \log_{10} (\|\delta^{(L)}\|) \quad (34)$$

for $l = 1, 2, 3, \dots, L$. (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

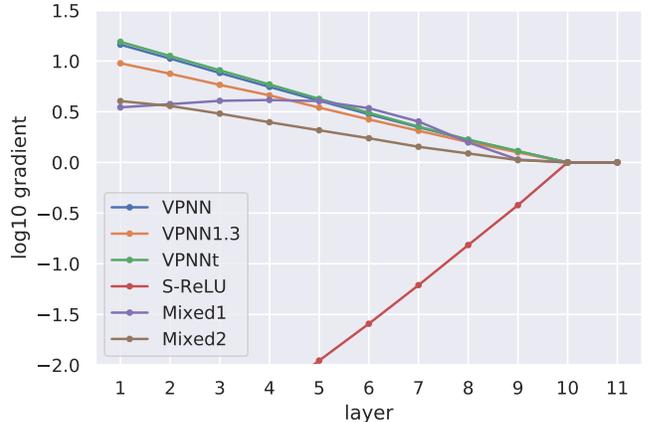


Fig. 8. Learning in the Layers: MNIST

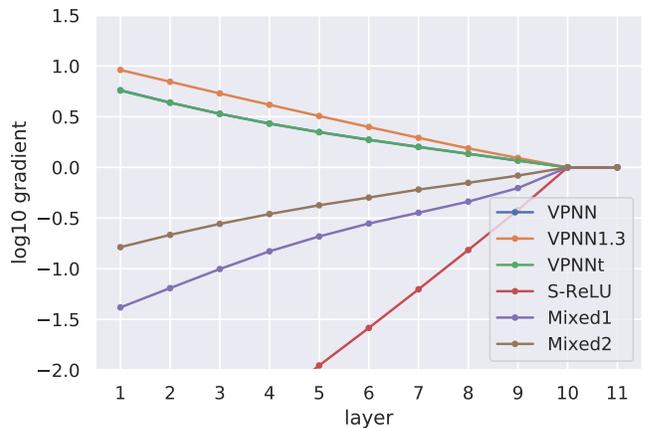


Fig. 9. Learning in the Layers: IMDB

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.

VI. CONCLUSIONS

We presented a new neural network architecture wherein all layers of the network are volume preserving, this includes a coupled activation layer, however on classification tasks we require a downsizing layer as the last layer. This last layer cannot therefore be volume preserving. We tested the network on two standard classification tasks and while the use of a fully volume-preserving network for classification tasks is unlikely to be the eventual application of this architecture we do showcase the ability of our architecture to both learn and preserve gradients. Demonstrating the preservation of gradients is our main goal in tackling these classification tasks as there are architectures such as convolutional neural networks that perform extremely well on these tasks and especially image classification tasks in general [20], [21].

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 and generative models. We also note the model may have applications where small memory footprints or smaller numbers of trainable parameters are required or to augment existing architectures.

A. Accessing the code

The code for the VPNN architecture is accessible in Github repository ([22]).

REFERENCES

- [1] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural networks*, vol. 61, pp. 85–117, 2015.
- [2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.
- [3] G. Hinton, L. Deng, D. Yu, G. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, B. Kingsbury *et al.*, "Deep neural networks for acoustic modeling in speech recognition," *IEEE Signal processing magazine*, vol. 29, 2012.
- [4] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.
- [5] S. Hochreiter, "Untersuchungen zu dynamischen neuronalen netzen," *Diploma, Technische Universität München*, vol. 91, no. 1, 1991.
- [6] Y. Bengio, P. Simard, P. Frasconi *et al.*, "Learning long-term dependencies with gradient descent is difficult," *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, 1994.
- [7] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics. JMLR Workshop and Conference Proceedings*, 2010, pp. 249–256.
- [8] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proceedings of the 27th international conference on machine learning (ICML-10)*, 2010, pp. 807–814.
- [9] R. Pascanu, T. Mikolov, and Y. Bengio, "On the difficulty of training recurrent neural networks," in *International conference on machine learning*. PMLR, 2013, pp. 1310–1318.

- [10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
- [11] J. Schmidhuber, "Learning complex, extended sequences using the principle of history compression," *Neural Computation*, vol. 4, no. 2, pp. 234–242, 1992.
- [12] A. M. Saxe, J. L. McClelland, and S. Ganguli, "Exact solutions to the nonlinear dynamics of learning in deep linear neural networks," *arXiv preprint arXiv:1312.6120*, 2013.
- [13] N. Bansal, X. Chen, and Z. Wang, "Can we gain more from orthogonality regularizations in training deep cnns?" *arXiv preprint arXiv:1810.09102*, 2018.
- [14] M. Henaff, A. Szlam, and Y. LeCun, "Recurrent orthogonal networks and long-memory tasks," in *International Conference on Machine Learning*. PMLR, 2016, pp. 2034–2042.
- [15] M. Arjovsky, A. Shah, and Y. Bengio, "Unitary evolution recurrent neural networks," in *International Conference on Machine Learning*. PMLR, 2016, pp. 1120–1128.
- [16] S. Wisdom, T. Powers, J. R. Hershey, J. L. Roux, and L. Atlas, "Full-capacity unitary recurrent neural networks," *arXiv preprint arXiv:1611.00035*, 2016.
- [17] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, 2011, pp. 315–323.
- [18] Y. LeCun *et al.*, "The MNIST dataset of handwritten digits (images)," 1999.
- [19] A. L. Maas, R. E. Daly, P. T. Pham, D. Huang, A. Y. Ng, and C. 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*. Association for Computational Linguistics, 2011, pp. 142–150.
- [20] A. Byerly, T. Kalganova, and I. Dear, "A branching and merging convolutional network with homogeneous filter capsules," *arXiv preprint arXiv:2001.09136*, 2020.
- [21] D. Hirata and N. Takahashi, "Ensemble learning in cnn augmented with fully connected subnetworks," *arXiv preprint arXiv:2003.08562*, 2020.
- [22] G. MacDonald, A. Godbout, B. Gillcash, and S. Cairns, "Python (Pytorch) code for VPNN," https://github.com/andrewgodbout/VPNN_pytorch, 2019.