PointNet

Zach Siegel, Apeksha Singh, Nat Snyder
Math 273

Abstract

Object classification and scene segmentation is a cornerstone of the computer vision community. While their are an abundance of algorithms and strategies for image based data, until recently, it was not known how to process point-cloud data efficiently and accurately in its raw form. In this paper we describe the novel PointNet architecture, a deep learning archiecture for

1 Introduction

Researchers in computer vision and machine learning have developed robust algorithms [15], [5], [18] for object classification and decision making [11], [8] through the use of convolutional learning techniques on 2D image inputs. Until the work of PointNet [13] in 2017, the same could not be said about 3-dimentional data involving point-clouds. Point clouds are inherently 'order-invariant' data types, meaning the permutation of any or all point-members within a cloud does not affect the information the cloud encompasses. Due to this unique characteristic, point cloud data does not fit the standard image-based convolutional network architecture built for object identification and scene segmentation.

While point clouds and images have inherently different data structures, the objectives for both data types typically surround tasks such as object classification, part segmentation, and scene segmentation. Traditionally, researchers would circumvent the challenges of point clouds by transforming clouds to another data representation. One such technique was to 'voxelize' the point-cloud onto a 3D grid representation [21],[22], so that 3D convolutional learning techniques could then be applied. This was not without drawbacks, as depending on the scale of the point cloud, enormous (and sparse) 3d voxel grids would emerge. This also led to the challenge of computing 3d convolution over potentially huge voxel-grids in a reasonable amount of time. Works such as [10],[3] allieviated some of the challenges of sparse 3D convolution, but large amounts of memory were still needed to store 3D voxel-grids.

Another technique described in [12] was to preprocess a point cloud into multiple image views, meaning the 3D point cloud would be transformed into a 2D image from several different perspectives. This technique worked well for simple object classification tasks when a single 3D object point-cloud was isolated from its surrounding environment, but became non-trivial to extend this technique to objectives such as scene or part segmentation. Furthermore, rendering multiple image views from the point-cloud upended any hopes for 'real-time' applicability. Hand-crafted features were also proposed [2], [1] in attempts to construct methods which could deal

with the various sizes and rotations that basic shapes could take when captured as clouds. These techniques used frequency domain transformations of the points captured in x-y-z space. However, they did not generalize well to scenarios with multiple objects and were not considered for scene or part segmentation. The deep learning community attempted to learn such spectral kernels [16] from data directly to classify manifolds, or meshes. Non-isometric shapes were a large shortcoming to the technique, as well as dealing with undesirable artifacts that were a biproduct of transforming the clouds to a mesh.

The motivation behind PointNet was to develop an architecture that could quickly process large point clouds in their raw form, for the tasks of object classification as well as part and scene segmentation. The novel contribution of this work was to both identify and understand why a deep learning architecture could deal with order invariant data directly. In this paper we showcase their network and how it is successfully applied to all the aforementioned objectives. We summarize the novel features of the PointNet architecture in section 2. Furthermore we directly compare the classification accuracy of PointNet against a VoxelNet 3D CNN with voxel-grid input, using the ModelNet40 point-cloud dataset as a train/test source for both network architectures.

2 PointNet Architecture

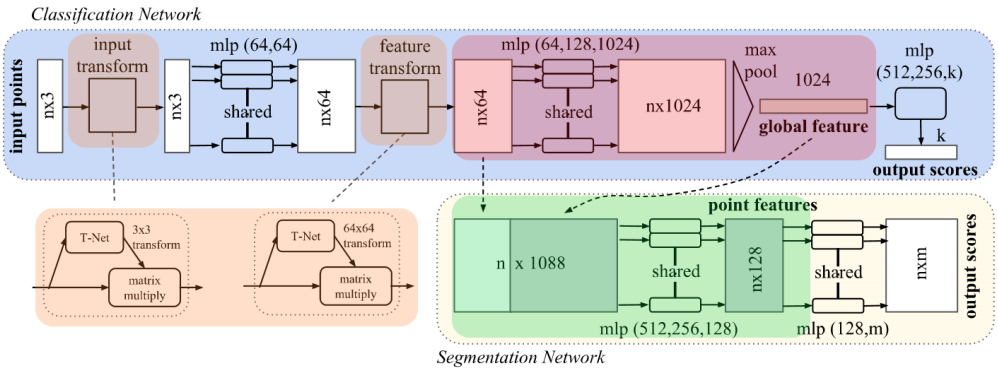

Figure 2. **PointNet Architecture.** The classification network takes n points as input, applies input and feature transformations, and then aggregates point features by max pooling. The output is classification scores for k classes. The segmentation network is an extension to the classification net. It concatenates global and local features and outputs per point scores. "mlp" stands for multi-layer perceptron, numbers in bracket are layer sizes. Batchnorm is used for all layers with ReLU. Dropout layers are used for the last mlp in classification net.

Figure 1: *Color coding was added to the figure of the network architecture from the paper to reflect which components of the architecture are inspired by what properties of the point set (red=unorderedness, green=local interactions, orange=invariance to transformations).*

The network architecture of the PointNet (Fig. 1) was created with the three main properties of point clouds in mind. The point set in $\mathbb{R}^n$, which is a subset of points from a Euclidean space, is firstly ***unordered***. This is unlike pixels in an image for example. The network therefore must be invariant to permutations of the order in which input data is fed to it. The authors note that there exist three strategies to create such invariance: 1) sort input into canonical order, 2) treat input as a sequence but augment with permutations, or 3) create a symmetric function to aggregate

information from each point in a way that is invariant to order, which is the strategy pursued by PointNet.

The authors comment that sorting is often not possible as it essentially requires defining a geometry-preserving bijection between a high-dimension space and a $1d$ line. The sorting should be stable with respect to perturbations of the points, which would require spatial proximity in the high dimension space to be preserved by the bijection. Additionally for point sets, there are often thousands of input elements, so sampling enough permutations of ordered data for training a RNN may also be infeasible. A symmetric function is one that takes n vectors as input and ouptuts the same vector regardless of the input order of the n vectors. Empirically, the authors demonstrate that these first two strategies do not perform as well as their PointNet's reliance on a symmetric function. The idea behind the PointNet architecture is to approximate a function f defined on a point set by applying a symmetric function on transformed input elements:

$$f(x_1, ..., x_n) \approx g(h(x_1), ..., h(x_n)) \tag{1}$$

where the $x_i \in \mathbb{R}^d$ and $h : \mathbb{R}^d \to \mathbb{R}^K$, typically for some $K \gg d$. In the PointNet architecture, h is approximated by a multi-layer perceptron (MLP) network and g is approximated by a composition of a single variable function and a max pooling function (red highlight in Fig. 1). The elementwise maximum, i.e. *max pooling* along each component of a set of vectors, is a simple example of a symmetric function. The authors tried others like elementwise summation and elementwise averaging, but found empirically that max pooling worked best. By using different h, many f can be learned that each capture different properties of the point set. This global signature ($[f_1...f_K]$) is then used as input to train a final MLP for classification. The authors additionally prove that the neural network can approximate set functions (f from above) that are continuous with respect to the Hausdorff $d_H(.,.)$ distance (the greatest of all the distances from a point in one set to the closest point in the other set), formally:

$$\left| f(S) - \gamma\left(MAX_{x_i \in S}\{h(x_i)\} \right) \right| < \epsilon \tag{2}$$

Here $f : \mathcal{X} \to \mathbb{R}, \mathcal{X} = \{S : S \subseteq [0,1]^m and |S| = n\}$. Additionally x_i are the elements of S ordered arbitrarily, γ is a continuous function, and MAX is the elementwise-maximum of n vectors. The idea behind this proof is to consider the set S' such that $d_H(S, S') < \delta_\epsilon$ so that $|f(S) - f(S')| < \epsilon$ for any such S' given the continuity of f. The interval [0, 1] (taking a one dimensional case for simplicity in notation) is split in K subintervals where $K = \lceil 1/\delta_\epsilon \rceil$ and S' is defined by taking the points in S and mapping them to the left end of the subinterval in which they lie. Next the function $\mathbf{h}(x) : \mathbb{R} \to \mathbb{R}^K = [h_1(x); ...; h_K(x)]$ is defined where each h_j is a soft indicator of x occupying interval j (i.e $h_j = e^{-d(x, [\frac{j-1}{K}, \frac{j}{K}])}$ where d is the point to interval distance function). Now the occupancy of interval j by points in S can be found by taking the maximum over $\{h_j(x_1), ..., h_j(x_n)\}$, denoted by the function $v_j(x_1, ..., x_n)$. Let $\mathbf{v} = [v_1; ...; v_k]$, a symmetric function taking in n vectors from $\mathbb{R}$ and returning a value in $\{0,1\}^K$ which describes the occupancy in each

interval. A function τ can then be defined to map this description of the occupancy back to the unique points in S'. Letting $\gamma(\mathbf{v}) = f(\tau(\mathbf{v}))$ from the continuity of f, $|\gamma(\mathbf{v}(x_1, ..., x_n)) - f(S)| < \epsilon$ and an expression like Equation 2 is obtained by substituting back the elementwise maximum and the function h.

There are several practical consequences of this theoretical analysis. Since the neural network approximates a continuous function, one take away is that small perturbations in the points of the point cloud should result in similarly small changes in the output. Additionally from the proof it is easy to see that the output of the neural network depends on the value of the K dimensional $\mathbf{v}(x_1, ..., x_n)$ rather than the n dimensional $\{x_1, ..., x_n\}$ that comes from the original point cloud. In particular, a critical set of points of size less than or equal to K can be found (Letting $u = MAX_{x_i \in S}\{\mathbf{h}(x_i)\}$, they can be found by choosing a point, c_j, in $\mathcal{X}$ such that $h_j(c_j) = u_j$ for each element of u) and so long as these critical points are included in the input, the output of the network will not change. This means the network is robust to input data loss and can handle more sparse representations of the point cloud. Conversely adding additional points ($p \in \mathcal{X}$) to the input where $h_j(p) < u_j \forall j$ will also not change the output of the network. The authors call the collection of all points, p, satisfying this condition the upper bound shape. This practically means the network is robust to some input data noise as well. Examples of the critical points and upper bound shape learned by PointNet for some objects is given in Figure 2. As also evident by this analysis, the choice in the value of K, which the authors call the bottleneck size, does significantly impact the performance of the network. They found empirically that increasing K improved classification accuracy on the ModelNet40 dataset. Finally this theoretical analysis unpacks the main idea behind PointNet, that is is to learn a new function h, that performs better than the voxelization described.

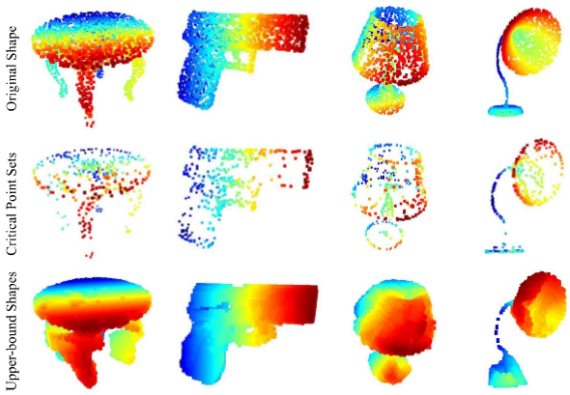

Figure 2: *Figure from the paper demonstrating the critical point set and upper bound shapes learned by PointNet for ModelNet40 objects.*

CAN ADD IN MORE ABOUT VOXELIZATION AND SYMMETRIC FUNCTIONS HERE!

The next property of point sets is that there is ***interaction between points***; they are not isolated and their local structure matters. Considering local structure, in additional to global structure, is especially important in segmentation tasks. The global features previously described are

concatenated with each of the points' features (green highlight in Fig. 1). New point features are extracted from this modified input and then these point features are used classify each point (both steps are implemented with a MLP).

The final property of point sets is that they possess ***invariance under transformations***. Rotations or translations of all the points should not affect the results of classification or segmentation, since the point clouds represent physical objects. One solution is to align all input to a canonical space before extracting features. In PointNet an affine transformation (3 x 3) matrix is predicted by a mini-network called T-net and this transformation is then applied to the input data (orange highlight in Fig. 1). The T-net mimics the larger PointNet architecture. It is composed of a shared MLP on each point with layer output sizes 64, 128, and 1024, then max pooling across points, and finally two fully connected layers with output sizes 512 and 256. PointNet also predicts another affine transformation matrix to align the point features. It uses the same T-net except it now predicts a 64 X 64 matrix and adds the following regularization term to the loss function to constrain the matrix predicted, A, close to (real unitary), or orthogonal and to stabilize optimization

$$L_{reg} = ||I - AA^T||_F^2 \tag{3}$$

where $||.||_F$ denotes the Frobenius norm.

CAN ADD IN MORE ABOUT SPATIAL TRANSFORMER NETWORKS HERE

3 Optimization Methods in PointNet Training

Training of the PointNet network requires solving an optimization problem, in particular minimizing a loss function. Here the loss function is defined as the sum of a softmax training loss and the regularization term mentioned previously. The softmax function is a generalization of the logistic function to multiple dimensions. In the final layer of PointNet the softmax function generates the probability that an instance belongs to each class. The cross-entropy or log-loss, which is the objective to be minimized, is then calculated between the actual class labels and the predicted probabilities and summed over all instances in the training dataset. This loss function is a function of the weights in the network architecture. Like most deep learning models, PointNet uses a gradient-descent based algorithm to optimize its loss function and these gradients are computed with the backpropagation algorithm, which takes advantage of the structure of neural networks, calculating the gradient of the loss function with respect to each weight one layer at a time using the chain rule. In particular, PointNet uses the Adam (adaptive moment estimation) optimizer [9]. Adam is a variant of stochastic gradient descent, which are advantageous when gradient calculations can be computationally intensive and some can potentially avoid local minima.

Stochastic gradient descent can be applied to minimizing any objective function of the form $Q(w) = \frac{1}{n}\sum_{i=1}^{n} Q_i(w)$ where the parameter w are the weights in the case of training a neural network and the Q_i correspond to loss function for the ith instance in the training dataset. In typical stochastic gradient descent, the gradient of the loss function is approximated at each

iteration by computing the gradient based on a sample of the instances in the training data. The size of this sample is referred to as the batch size and the authors of PointNet use a batch size of 32 for training. At each iteration the samples are randomly shuffled and a new batch is selected for approximating the gradient. In vanilla stochastic gradient descent the step-size (α) or "learning rate" is fixed. This can be problematic because too large of a step size can cause instability and too small of a step size can lead to slow convergence. There have been several extensions and variants proposed based on stochastic gradient descent to address this. These include implicit stochastic gradient descent (ISGD) [20] where the stochastic gradient is essentially evaluated at the next iterate rather than the current one (which can be interpreted as a stochastic version of the proximal algorithms we saw in class). Additionally there are variants that consider a momentum term where the change in weights at each iteration is a linear combination of the stochastic gradient calculated at that iteration and the change in weights from the previous iteration (Δw), where the coefficient in front of Δw is typically an exponential decay factor between 0 and 1. Finally there are stochastic gradient methods where the step size is adaptive, such as AdaGrad, RMSProp, and Adam, in which we will focus here on the last one since it is utilized in training PointNet.

In addition to calculating the stochastic gradient at each each iteration, the Adam algorithm updates exponential moving average of the gradient (m) and the squared gradient (v) (i.e. the first and second moments) at each iteration (the decay rates (β_1, β_2) of these moments are hyperparameters of the algorithm for with the authors of Adam provide suggested good default values). The updated weights at each iteration are then given by $w_{t+1} = w_t - \alpha \frac{\hat{m}}{\sqrt{\hat{v}}+\epsilon}$ where ϵ is another hyperparameter for which the authors of Adam suggest a small scalar value. The authors of PointNet utilized these default values in their TensorFlow implementation ($\beta_1 = 0.9, \beta_2 = 0.99, \epsilon = 10^-8$) They utilize an initial learning rate, α, of 0.001 (the value suggested by the authors of Adam), but additionally divide the learning rate by 2 every 20 epochs. Since the moving averages are initialized at zero, $\hat{m}$ and $\hat{v}$ are the values of m and v with bias correction. Taking ϵ to zero, it can be shown that the effective step size at each iteration is bounded above by either $\alpha(1-\beta_1)/\sqrt{(1-\beta_2)}$ (depending on the values of β_1 and β_2) or α. The first case only emerges with extreme sparsity, i.e. the gradient with respect to a parameter is now nonzero but has been zero at all prior timesteps. In less sparse cases the bound will be smaller, demonstrating the adaptive behavior of Adam.

The authors of Adam provide an analysis of its convergence using the online learning framework [23]. In this framework, algorithm performance is measured by minimization of the regret defined as $R(T) = \sum_{t=1}^{T}[Q_t(w_t) - Q_t(w^*)]$ where w^* are the optimized weights. They provide a proof that Adam has a $O(\sqrt{T})$ regret bound when the gradients are bounded, $\frac{\beta_1^2}{\sqrt{\beta_2}} < 1$, and α is decaying at a rate of $t^{-1/2}$ for convex functions Q_t, f_t. The idea behind their proof comes from the convexity of f_t; since the f_t are convex, $f_t(w_t) - f_t(w^*) \leq g_t^T(w_t - w^*)$, where g_t is the gradient at iteration t. Therefore finding an upper bound on the regret is now a problem of finding an upper bound on the quantity involving the gradient from the previous expression. This relies on showing that $\sum_{t=1}^{T}\sqrt{\frac{g_{t,i}^2}{t}}$ where $g_{t,i}$ is the ith element of the gradient at iteration t and $\sum_{t=1}^{T}\frac{\hat{m}_{t,i}^2}{\sqrt{t\hat{v}_{t,i}^2}}$ are bounded in their proof. Some recent works (such as in [14]) have provided examples of where

Adam can fail because the average regret does not go to zero as $T \to \infty$, demonstrating the importance in hyperparameter choices and suggesting modifications to Adam, such as placing essentially a longer term memory on past gradients (AMSGrad).

Utilizing a gradient-based optimization method requires calculation of a gradient. This warrants some discussion since all the layers of PointNet include ReLU, the rectified linear activation function. ReLU returns its input if it is greater than zero and returns zero if not (i.e. $f(x) = max(x, 0)$). ReLU is thus a simple function that is not computationally intensive to deal with, which can contribute to faster model training times, and can give sparser solutions. Additionally traditional activation functions such as the sigmoid and tanh function can suffer from the vanishing gradient problem in gradient-descent based optimization methods. Since the sigmoid and tanh functions plateau over most values, the value of the gradient can often be near zero. This means the gradient of the front layers of neural network will be a multiplication of many small values through the backpropagation algorithm, and a vanishing gradient means the weights will not update at each iteration. Of note though, ReLU can suffer from the opposite problem of exploding gradients, which can make optimization unstable. In terms of calculating the gradient, ReLU is not differentiable at zero, however from our discussion of subdifferentials in class, gradient-based methods can simply pick a value between 0 and 1 at this point.

The PointNet network also includes a batch normalization in all layers except the last one. In batch normalization [6] a standard normalization (subtraction of mean and dividing by variance) is applied to the batch output of each layer before it is fed into the next layer. Empirically, batch normalization has led to faster model training with more robustness to hyperparameter choices, but it's worth trying to understand why these advantages occur. Viewing each layer of a neural network as a separate optimization problem, the inputs to this problem change at each step depending on the output from previous layers. Hence the optimization problem is not fixed, which the authors of the batch normalization paper term, internal covariate shift. The authors hypothesize that this constant shifting of the optimization problem hampers convergence, thus they propose batch normalization to alleviate this effect and contribute to faster convergence. More recent analysis [19] has revealed that in many settings, batch normalization actually helps to reduce the value of the gradient with respect to a layer's output and smooth the objective function. When optimizing a neural network using a gradient-based method, the internal covariate shift is related to the difference in the values of the gradient before and after updating the weights, so these two intuitions behind batch normalization's performance are related. Additionally, there are momentum versions of batch normalization, which is what is used by the authors in training PointNet. In these variations, an average of past batch mean and variances is maintained and used in computing the batch normalization, which is intended to make results more robust to batch size. For the training of PointNet, the authors actually implement a rate that starts at 0.5 and then gradually increases to 0.99.

Finally of note in the discussion of the optimization in training PointNet is the spatial transformer networks. These components do cause PointNet to be different than more standard feed-forward architectures since the input essentially enters the network twice, once for learning

the affine transformation, and then again when the learned affine transformation is applied to the input (a matrix multiplication). At first glance, it might appear that such a departure from standard architecture would cause problems for backpropagation, however this is not the case. The application of the affine transformation can be simply thought of as another layer within PointNet. The input to this layer is the affine transformation and the output to this layer is the affine transformation applied to the fixed input data. Also it is obvious that the output of this layer can be easily differentiated with respect to its input.

To conclude this section, we provide some demonstrations of the optimization methods and network components utilized in the training of PointNet by performing classification of the MNIST handwritten digit database. We implemented a simple network of three fully connected layers for this example in TensorFlow with softmax classification loss. The layers contain batch normalization and ReLU activation functions to match the PointNet implementation. First we wanted to explore the convergence behavior of the Adam optimizer. From the convergence analysis presented by the authors of Adam, we should be able to predict how changing the hyperparameters of the optimization algorithm affects convergence. In 3A we plot the training accuracy (inversely related to the loss function) versus the training epoch. We plot the results for Adam with the default parameter values mentioned previously, and also utilized by the authors PointNet. In addition we plot the results with an increased β_1, decreased β_2, and increased α. As predicted by the theoretical analysis, these modifications all negatively impact the convergence of the Adam optimizer. Next in 3B, we explored the effect of swapping Adam for a vanilla SGD with the same learning rate, but fixed. The results demonstrate the clear advantage of a adaptive step size, with Adam enjoying much faster convergence than Vanilla SGD (the authors of Adam have also demonstrated the superior performance of Adam compared to other adaptive methods on MNIST classification [9]). Additionally in 3B, keeping the default Adam optimizer, we explored the effect of eliminating batch normalization and switching the ReLU activation function for a sigmoid one. It should be explicitly stated here that the underlying optimization has changed with these modifications, so the values for training accuracy obtained are not directly comparable. However we can still see in this example that these modifications to the network architecture do not greatly affect the stability nor speed of convergence of the optimization process here. Of note though, the default Adam implementation does maintain the best accuracy on a separate testing dataset, suggesting it might have learned a sparser solution providing for more generalizability. The advantages of batch normalization and the ReLU activation function become more apparent in more complex problems like PointNet, where the example used here is more trivial. Finally in 3C, we utilized an implementation of the spatial transformer network with a CNN for classification of the MNIST dataset [7] as a starting point to compare results with and without the spatial transformer network. The spatial transformer creates a more complex optimization problem, which is reflected by the more oscillatory plot of loss over epochs, but ultimately the network with the spatial transformer performs slightly better on unseen data. Like mentioned previously, PointNet stabilizes the optimization by adding a regularization term and the addition of the spatial transformation network provides slighty improved classification results.

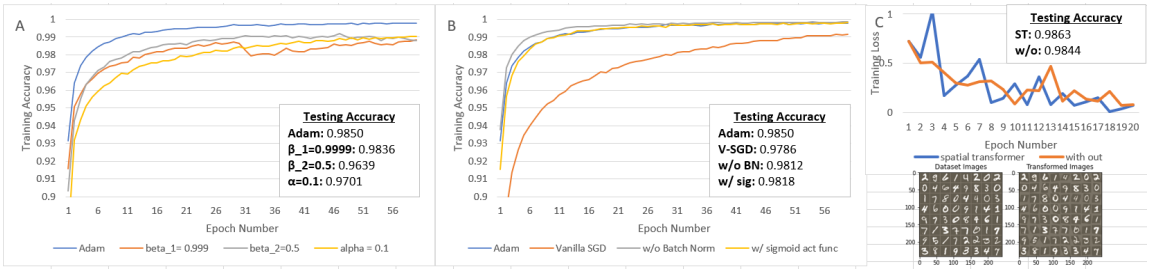

Figure 3

4 PointNet and Experimental Results

In this section we highlight our findings of testing the point-cloud architecture against voxelization. Particularly, we test the PointNet architecture against an architecture called 'Voxel-Net' which is a 3D convolutional neural network capable of performing classification over a 3D voxel-grid input. Here we compare PointNet to Voxel-Net using the ModelNet40 dataset, which consists of 40 different classes of point-cloud objects.

As the PointNet architecture has been discussed at length in the previous sections, we summarize the hyperparameters chosen to run the PointNet. An input size of 1024 points is subsampled from an object's point-cloud, standardizing the input dimention of the cloud to 1024×3. Standard stochastic gradient descent (SGD) with mini-batching of size 32 is chosen for use in conjunction with the Adam optimizer. A learning rate of .001 is initially set which is set to decay at a rate of .7 after every epoch. An epoch is declared as a single pass over all the training data divided by the size of a mini-batch. Before a point cloud is fed to the network for training, the cloud is rotated and jittered as well to augment the learning process. We see in figure 4 an accuracy of 80% is achieved after 3800 mini-batches (18 epochs). The authors of PointNet note the testing accuracy can achieve 89.2% on this ModelNet40. Figure 5 depicts the training loss. Note that after 3000 batches the loss is seen to become very flat, and it is unclear whether more hyperparameter tuning would aid in further convergence. Also note that swapping the adam optimizer for a momentum based optimizer was (empirically) not seen to have any large effects on training accuracy or model loss. It should lastly be noted that model training was ended after 30 minutes to prevent any overheating of our deep learning machinery, as temperatures reached near-critical system tolerances. More elegant methods of saving and reloading parameters could have been incorporated to overcome this problem but we leave this for future work.

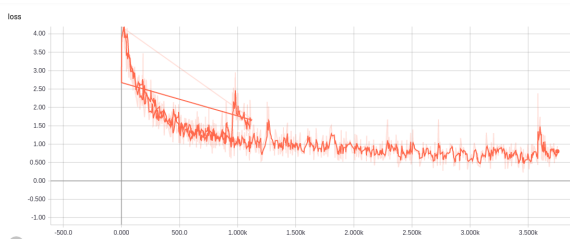

Figure 5: *PointNet: training loss vs number of training mini-batches.*

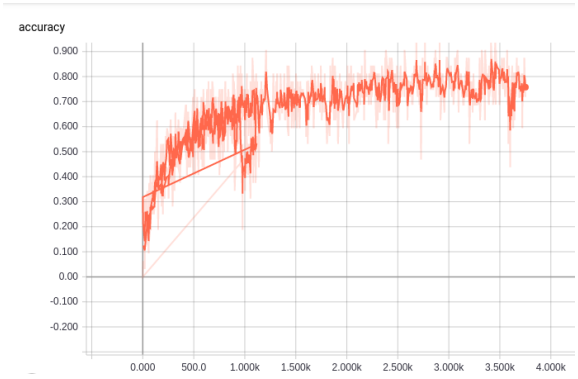

Figure 4: *PointNet: Test accuracy vs. number of training mini-batches. 80% accuracy was achieved after approximately 3800 minibatches (approximately 30 minutes of training on a NVIDIA GeForce 1060x GPU). Graphic from Tensorboard*

The performance of the PointNet architecture was compared to a 3D convolutional neural network which classified voxelized representations of the point-clouds. This type of network is typicalled dubbed a *VoxelNet*. First, all point clouds of the ModelNet40 dataset were transformed into 3D voxel-grids. (It should be noted the memory requirements to do so is quite large. 400MB of point-clouds were converted to greater than 5GB of voxel-grids). A coarse voxelgrid of $32 \times 32 \times 32$ voxels in the (x,y,z) was then chosen to uniformly discretize the point-cloud space for 3D convolution. Note this would take much more preprocessing if the ModelNet40 had not formatted the point-clouds appropriately to begin. Four layers of 3D convolution were used, which had kernal sizes of (64,5,2), (64,3,1), (128,2,2), (128,2,2), respectively. For example, layer one's (64,5,2) can be read as 64 5x2 kernels, which are then moved across the 3D voxel-grid with a stride of 2 (our chosen hyper-parameter). 3D max-pooling was used after the fourth layer to aggregate latent information and reduce dimentionality. Lastly, two fully connected layers are applied to transform the (flattened) latent information after layer 4. Finally a softmax is applied to classify the 40 different objects appropriately. Note that no batch normalization or ReLu activation was applied within the last fully connected layer, but both were applied in the first fully connected layer.

As seen in figure 6, a testing accuracy of 85% was achieved using this strategy. We observe a large increase in accuracy over the first 2000 mini-batches, which eventually levels out around the 16000th mini-batch. We should note that the voxelnet was trained for more batches than

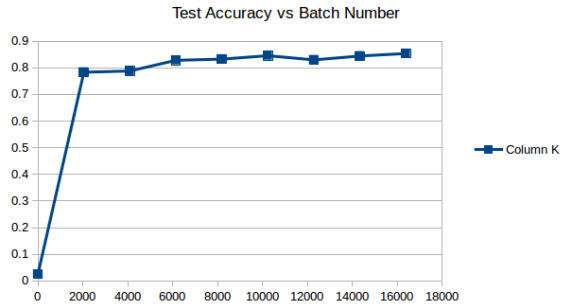

Figure 6: *VoxelNet: Test accuracy vs. mini-batch number*

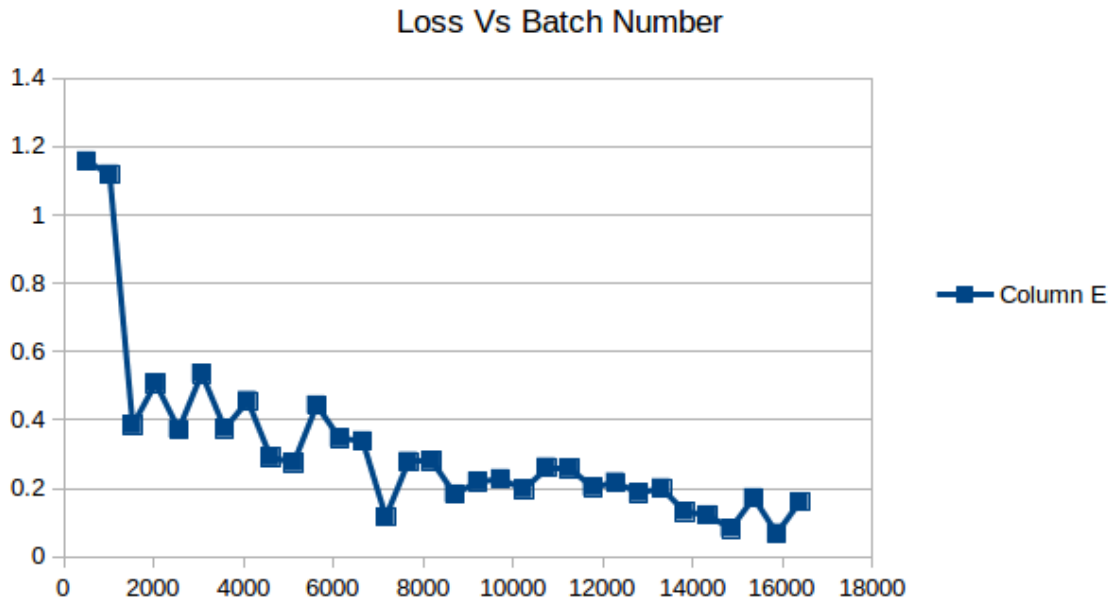

Figure 7: *VoxelNet: loss vs. mini-batch number*

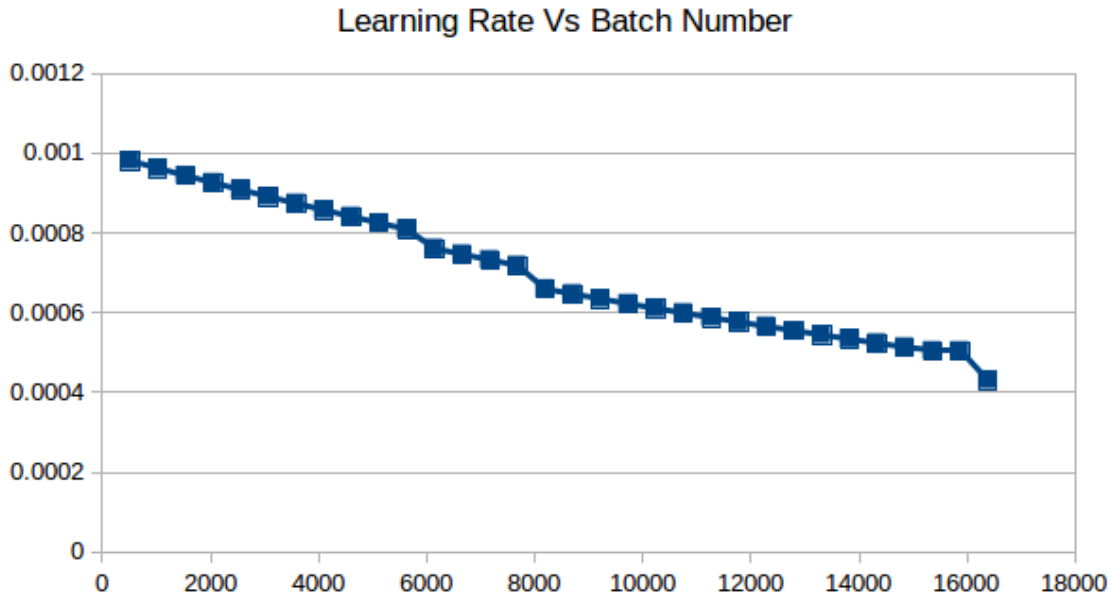

Figure 8: *VoxelNet: learning rate vs. mini-batch number*

pointnet only because computer overheating was (somewhat strangely) not observed in this case. The learning rate was set to decrease at a near linear rate (figure 8) after every 512 mini-batches, resulting in a reasonably decreasing loss (figure 7).

We note that the main advantage of the PointNet over other methods (such as voxelization) is that it does not require the initial data transformation the VoxelNet requires. This makes the PointNet architecture a serious candidate for real time application in the domain of object classification.

5 Conclusion

summarize critiques of pointnet future directions/more current work in field

(This is currently just notes) –Nat asks: How can RL be applied to MNIST? What is the reward function? Curious, why isnt this supervised learning?

–Zach answers: This *is* supervised learning, and it is only supervised by class labels. The RL I was contrasting the STN with would go as follows: for a given image of a handwritten digit, rotate it several times to create “rotated MNIST" samples, but preserve the groundtruth (vertical orientation). During training, for each sample, train part of the network to perform a rotation by scoring the output (loss function/reward function) based on how different it is from the “ground truth” orientation (e.g. using the most naive possible loss function: sum of squared pixel differences). Then that part of the network will learn to “orient” samples to the desired pose.

[I called this “reinforcement learning” because it requires more than class labels; it requires a

somewhat involved process of rotating images and deciding "how bad" an improperly-rotated output is. Maybe referring to that as RL is a stretch!]

Contrast this with the STN: the STN precedes some fully-connected MLP layers that classify images. But the STN is not a collection of fully-connected layers, and the "transformation" layer doesn't resemble a perceptron at all. The STN transformation layer is highly constrained to simply represent a linear transformation (in the case of PointNet, a 3x3 unitary matrix multiplication; in the STN paper, they explore affine and more complex transformations). So the output of the STN is trained to simply be a rotated version of the original sample. Since this is fed into a MLP that classifies, and error is backpropagated, this transformation learns to rotate images *in the best possible way to yield good classifications*. That is, it tries to make inputs of the same class look as similar as possible, hence it finds a canonical pose. My understanding is that this canonical pose need not be our desired canonical pose; that is, it might learn to orient 9's all upside down. Crucially, though, it learns to rotate all 9's to be in the same orientation, to the extent that this is possible given the limited class of transformations the STN is permitted to perform.

The transformation part of the STN (the only part used in PointNet) has two parts. First, there are MLP or even CNN layers that learn to map inputs to the desired rotation. That is, these layers learn "for input that is a 30°-rotated '9', output a -30° rotation matrix". Then, in the next part, the only output values that are passed to the subsequent classification layers is *the transformed input*. For this to be possible, the original input has to be passed into the transformation-generating layers, and then *the input is passed again into the network in applying the transformation to the input, which gets fed forward to subsequent layers*. As a result, the backpropagation update will include the input values in partial derivatives in two different places. That's fine, though, because the whole thing is differentiable. The error also ends up backpropagating both to the transformation-generating layers; to constrain the transformation further (e.g. to ensure a *unitary* matrix multiplication, so that you get a rigid rotation), a loss term measuring how "non-unitary" the transformation is as well, which also of course affects these transformation-generating layers.

- [7] notes that for training rigid-transformation-invariant classifiers (e.g. "rotated MNIST" handwritten digits), it is possible to use reinforcement learning: use separate software to transform (e.g. rotate) images with learned transformations, then compare to a ground-truth representation in a training set (e.g. a non-rotated depiction of each digit), then backpropagate the error, thereby training some layers to canonically display each digit (in a standard orientation). By contrast, they train a network that (internally) learns canonical representations and outputs canonically-oriented variants of the inputs (e.g. upright digits), strictly using backpropagation. Thus, their "spatial transformer network" ("ST" in [13]) can be included as a submodule to any network whose input is spatial data. Any network that is preceded by an STN in this way will learn from data "oriented" in a canonical representation; the STN itself will simultaneously learn how to perform this task via the error backpropagated through the rest of the network.

  The STN's function is intuitively clear on rotated image data, and the types of transformations it learns to perform can be restricted to those that correspond to rigid euclidean motion

(translations and rotations). However, its usefulness extends to other feature spaces and a slightly broader set of transformations. In PointNet [13], a ST module is included twice in the network architecture, both times applying both to direct spatial representations and other calculated features (max-pooled representations of regions, etc.).

In section 3.2 they discuss at length the family of transformations they consider that can be learned during backpropagation and applied as

$$T_\theta \mathbf{x_i} = \mathbf{A} \cdot \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}.$$

For example, an "attention" transformation $T_\theta x$ defined by

$$A_\theta = \begin{bmatrix} s & 0 & t_x \\ 0 & s & t_y \end{bmatrix},$$

which corresponds to "cropping, translation, and isotropic scaling", i.e. giving "attention". The broader class of transformations they consider in their model for image tasks utilizes all six entries of A_θ and allows for "cropping, translation, rotation, scale, and skew to be applied to the input feature map." They note that T_θ can be more general as long as it is differentiable with respect to the parameters.

This transformation is applied as a linear layer.

- In [4], they compare their scalable object detection-and-classification network to previous works that use separate networks to segment and to classify. Note that their novel network gives scores for how likely the output bounding boxes are to contain *an* object (in any of the classes), but does not actually classify.

  > A different line of work, closer to ours, is based on the idea that objects can be localized without having to know their class. Some of these approaches build on bottom-up classless segmentation [9]. The segments, obtained in this way, can be scored using top-down feedback [14, 2, 4]. Using the same motivation, Alexe et al. [1] use an inexpensive classifier to score object hypotheses for being an object or not and in this way reduce the number of location for the subsequent detection steps. These approaches can be thought of as Multi-layered models, with segmentation as first layer and a segment classification as a subsequent layer. Despite the fact that they encode proven perceptual principles, we will show that having deeper models which are fully learned can lead to superior results.

  This work requires reinforcement for each training sample: the output bounding boxes and confidence scores are compared to ground truth bounding boxes (each with confident 1); an assignment problem matches the predicted bounding boxes to the true bounding boxes they are closest to, and the error is backpropogated for both the bounding coordinates and the confidence scores of those boxes.

  Many of the confidence scores are trained to be zero, as the network has the capacity to

regress K bounding boxes (and confidence scores) while a typical image has $m << K$ actual objects.

The details of this method (requiring relatively expensive reinforcement on each training iteration) show that it is fairly specialized to the task of image segment detection (consider especially the details below regarding training set centroid cluster "prior matching" to encourage differentiation of bounding boxes). Still, the principles of segmentation can be generalized and indeed proved useful in the STM [7] and PointNet [13] works that followed.

> While the loss as defined above is in principle sufficient, three modifications make it possible to reach better accuracy significantly faster. The first such modification is to perform clustering of ground truth locations and find K such clusters/centroids that we can use as priors for each of the predicted locations. Thus, the learning algorithm is encouraged to learn a residual to a prior, for each of the predicted locations.
>
> A second modification pertains to using these priors in the matching process: instead of matching the N ground truth locations with the K predictions, we find the best match between the K priors and the ground truth. Once the matching is done, the target confidences are computed as before. Moreover, the location prediction loss is also unchanged: for any matched pair of (target, prediction) locations, the loss is defined by the difference between the groundtruth and the coordinates that correspond to the matched prior. We call the usage of priors for matching prior matching and hypothesize that it enforces diversification among the predictions.

References

[1] M. Aubry, U. Schlickewei, and D. Cremers. The wave kernel signature: A quantum mechanical approach to shape analysis. In *2011 IEEE International Conference on Computer Vision Workshops (ICCV Workshops)*, pages 1626–1633, 2011.

[2] Alexander M. Bronstein. Spectral descriptors for deformable shapes. *arXiv e-prints*, page arXiv:1110.5015, October 2011.

[3] Martin Engelcke, Dushyant Rao, Dominic Zeng Wang, Chi Hay Tong, and Ingmar Posner. Vote3Deep: Fast Object Detection in 3D Point Clouds Using Efficient Convolutional Neural Networks. *arXiv e-prints*, page arXiv:1609.06666, September 2016.

[4] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Scalable object detection using deep neural networks. *CoRR*, abs/1312.2249, 2013.

[5] Ross Girshick. Fast R-CNN. *arXiv e-prints*, page arXiv:1504.08083, April 2015.

[6] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift, 2015.

[7] Max Jaderberg, Karen Simonyan, Andrew Zisserman, and Koray Kavukcuoglu. Spatial transformer networks. *CoRR*, abs/1506.02025, 2015.

[8] Mattis Manfred Kämmerer. On Policy Gradients. *arXiv e-prints*, page arXiv:1911.04817, November 2019.

[9] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization, 2017.

[10] Yangyan Li, Soeren Pirk, Hao Su, Charles R. Qi, and Leonidas J. Guibas. FPNN: Field Probing Neural Networks for 3D Data. *arXiv e-prints*, page arXiv:1605.06240, May 2016.

[11] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv e-prints*, page arXiv:1312.5602, December 2013.

[12] Charles R. Qi, Hao Su, Matthias Niessner, Angela Dai, Mengyuan Yan, and Leonidas J. Guibas. Volumetric and Multi-View CNNs for Object Classification on 3D Data. *arXiv e-prints*, page arXiv:1604.03265, April 2016.

[13] Charles Ruizhongtai Qi, Hao Su, Kaichun Mo, and Leonidas J. Guibas. Pointnet: Deep learning on point sets for 3d classification and segmentation. *CoRR*, abs/1612.00593, 2016.

[14] Sashank J. Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond, 2019.

[15] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You Only Look Once: Unified, Real-Time Object Detection. *arXiv e-prints*, page arXiv:1506.02640, June 2015.

[16] Oren Rippel, Jasper Snoek, and Ryan P. Adams. Spectral Representations for Convolutional Neural Networks. *arXiv e-prints*, page arXiv:1506.03767, June 2015.

[17] Sebastian Ruder. An overview of gradient descent optimization algorithms. *CoRR*, abs/1609.04747, 2016.

[18] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *arXiv e-prints*, page arXiv:1409.0575, September 2014.

[19] Shibani Santurkar, Dimitris Tsipras, Andrew Ilyas, and Aleksander Madry. How does batch normalization help optimization?, 2019.

[20] Panos Toulis and Edoardo M. Airoldi. Asymptotic and finite-sample properties of estimators based on stochastic gradients, 2016.

[21] Zhirong Wu, Shuran Song, Aditya Khosla, Fisher Yu, Linguang Zhang, Xiaoou Tang, and Jianxiong Xiao. 3D ShapeNets: A Deep Representation for Volumetric Shapes. *arXiv e-prints*, page arXiv:1406.5670, June 2014.

[22] Yin Zhou and Oncel Tuzel. VoxelNet: End-to-End Learning for Point Cloud Based 3D Object Detection. *arXiv e-prints*, page arXiv:1711.06396, November 2017.

[23] Martin Zinkevich. Online convex programming and generalized infinitesimal gradient ascent. In *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, page 928–935. AAAI Press, 2003.