Application of Neural Networks to Neutrino Interaction Identification

Haoyang Li,* Carlos Pareja,† Jay Sun,‡ and Sahil Bhalla§
University of California, San Diego

(Physics 139 Winter 2023 - Group 8)
(Dated: March 24, 2023)

In this project, we build and train 3 different neural networks in order to compare their respective performance in differentiating particle interaction images from LArTPC.

I. INTRODUCTION

Neutrinos are hard to be directly detected as they rarely interact with materials. However, the product particles of neutrino interactions can be detected and tracked via liquid argon time projection chambers (LArTPCs), which record the trace and energy deposit of the product particles. In [1], Aurisano et al. used a convolutional neural network (CNN) that takes in the side and top views of LArTPC's digitized data to classify the product particles in neutrino interactions. They named the architecture convolutional visual network (CVN).

In particular, we are interested in the CVN's inception module and the two-view design. The inception module allows extracting features with inception fields of different sizes at the same depth. The two-view design enables the CVN independently generate features for different views and then merges them later, which is alternative to the common solution that treats different views of the digitized detector data as different channels. We rebuilt the CVN model and also built two other networks (ResNet41, 2-View ResNet) along the way for comparison. The 2-View ResNet replaced the inception modules with residual connections while ResNet41 uses residual connections and treats different views as different input channels. Our goal is to compare the performances of the three models on classifying neutrino interactions' product particles. The project aims to evaluate the inception module and the two-view design as potential designs for future neutrino detection tasks.

II. DATASET

Our dataset includes two ROOT files of simulated digitized detector views of LArTPCs. The ROOT file of the training set can be found here, and the testing set can be found here. The training set consists of 50k events, and the test set has 40k events. Each event has 3 views (XY, YZ, ZX). The XY view of the first event in the training set is shown in Fig. 1. Fig. 2 shows the distribution of momentum in 3 orthogonal directions (x, y, and z), implying that the simulated particles have no preferred direction in our dataset. The simulated detector was a cube and all views have the same size (256×256). However, in [1], the two views of the CVN input was chosen according to the beam direction and the detector geometry, which implies that a machine learning model could benefit more in our dataset by providing all three views. For consistency, we followed [1] and picked the first two views. Each event has a PDG ID that indicates the type of partcicle, which transformed to a one-hot encoded target when training. The five types of particles in our dataset are electron, muon, photon, pion, and proton. Every class is balanced in the dataset, so a random guess would yield 20% accuracy. Our fundamental goal is to achieve an accuracy that can beat 20%.

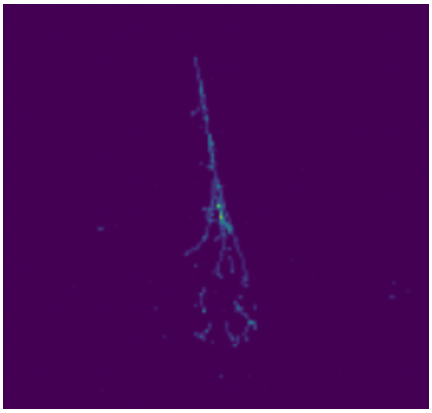

FIG. 1. An electron event in the XY view of the dataset.

We wrote notebooks (github-folder/notebooks/root2hdf5.ipynb) to convert the ROOT files into .h5 file, and then further convert to .npy. Around 5% of the events in our dataset has more than one track and were disgarded. Recall that each 'view' has a dimension of [256, 256]. Because the structure of CVN only takes in 2 images. For consistency, we feed the first 2 views as input data into all our networks (CVN, ResNet41, 2 view ResNet). Every image is normalized before we feed them into the models.

* hal113@ucsd.edu
† cpareja@ucsd.edu
‡ k8sun@ucsd.edu
§ s2bhalla@ucsd.edu

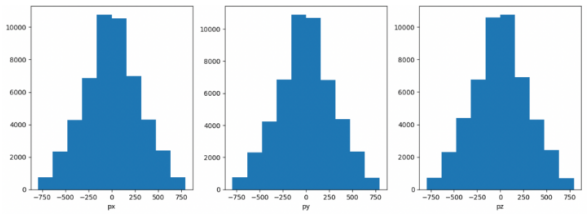

FIG. 2. The distribution of momentum in 3 basis directions.

III. METHODS

FIG. 3. CVN Architecture

Convolutional Visual Network (CVN) is a convolutional neural network architecture developed by Aurisano et al. [1]. A schematic of CVN from [1] is shown in Fig. 3. In [1], the authors utilize the CVN architecture in 3 in order to accomplish neutrino interaction classification. This architecture is split between x-view data and y-view data where both of these networks are running in parallel and their outputs are then concatenated along the channels dimension. Both x-view and y-view data are of dimensions $[N, 1, 256, 256]$ where N is our batch size that we utilize. Throughout our research we utilized many different batch sizes to determine which batch size would have our network run faster but ultimately we ran the network with a batch size of 64. According to the authors from [1], the CVN architecture in Fig. 3 was inspired from the GoogLeNet architecture which is known for its success in many Computer Vision tasks such as image classification and object detection. GoogLeNet is most known for its use of the Inception Module architecture which will be further discussed later. One key difference between the work the authors in [1] conduct in comparison to traditional use of CNN's for image classification is that instead of passing a network two dimensions of images such as a $[N, 2, 256, 256]$, we instead split these two channels and pass the image data into separate but identical networks in order for these networks to extract as much information for each channel dimension. Traditionally, most image datasets consists of images that have dimensions of $[C, H, W]$ where C is the channels dimension consisting of three channels for the respective Red, Green, Blue (RGB) channels and H is the height of the image while W is the width of the image all in pixels.

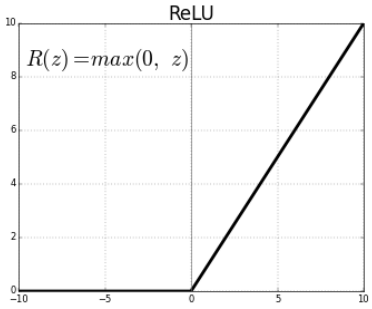

FIG. 4. ReLU activation function

Our implementation of the CVN architecture in Fig. 3 was developed in the PyTorch deep learning framework. In our code we implemented and utilized a total of four PyTorch modules. A PyTorch module is a class for all PyTorch Neural Network modules. PyTorch modules are also able to be nested with each other and this is something we heavily utilized to implement the architecture in Fig. 3. The first PyTorch module we implemented was for a simple convolutional block that we nested in our Inception Modules. In all of our convolutional layers we utilized the Rectified Linear Unit (ReLU) activation function in Fig. 4 in order to introduce nonlinearities to our network.

The second PyTorch module we implemented was the Inception Module architecture in Fig. 5. The Inception Module is utilized in deep neural networks in order to reduce the computational expense when training these networks on large datasets and it is also utilized for dimensionality reduction. In the order of left to right in Fig. 5, we first start with taking in the data from our previous layer and applying a 1x1 Convolution. The 1x1 convolution is special because this allows the network to learn more across the channel depth of the image. A 1x1 convolution is also beneficial due to the dimensionality

reduction it provides such as reducing the height, width, and channels of the image. Next in the Inception Module we have the 1x1 convolution followed by a 3x3 convolution as well as a 1x1 convolution followed by a 5x5 convolution. The purpose of the 3x3 and 5x5 convolutions here are to learn different spatial patters of the image at different kernel sizes. The ability to learn these representations at different kernel sizes of our convolutions is important because this allows the network to have a better performance in its classification task as it will be able to learn distinct features from our image data. Finally, in the Inception Module we have the 3x3 pooling layer followed a 1x1 convolution. This final branch of the Inception Module starts with a down sampling of the height and width of our image data with a 3x3 average pool. However, we add padding in this pooling layer in order to maintain the same height and width of the other branch outputs within our Inception module and finally we pass the output of 3x3 average pooling to our last 1x1 convolution. Next, we concatenate the outputs of all branches in our Inception Module along the channels dimension as pass in this data to the next step in the CVN architecture in Fig. 3.

The third PyTorch module we implemented was the generic x-view and y-view model. This PyTorch module consisted of implementing both the left and right branch of the CVN architecture in Fig. 3. In this architecture we start with taking in our data with shape $[N, 1, 256, 256]$ then passing it along a 7x7 convolutional layer, then a max pooling layer. Next, we have a Local Response Normalization (LRN), the LRN is used because of our use of the ReLU activation function for each convolutional layer in our architecture. We can see in Fig. 4 that the ReLU activation function is unbounded therefore we need a method to normalize our values and to prevent this unbounded nature. The use of LRN comes from the field of Neurobiology because we want to encourage lateral inhibition which is the ability of excited neurons to limit the capacity and activity of surrounding neurons. In other words, this translates to our research because in our CVN we want to be able to boost the active neurons that are firing and have larger activations but we first need to normalize our data before we can determine which ReLU neurons have largest activations. After passing our data through more convolutional layers, pooling layers, and performing LRN we arrive at the Inception modules. In this PyTorch module we call three Inception Modules that we constructed as PyTorch modules for each view of the data.

Our final PyTorch module consisted of combining all of our previous PyTorch modules and having a single callable module that can implement the entire architecture in Fig. 3. This module combines the outputs of both the x-view and y-view networks along the channels dimension then passes in this data to another Inception Module, then we perform a max pool. In order to perform our classification task, we introduce two fully connected layers in order to expand our data. The number of neurons in the first fully connected layer is the result of the $C * H * W$ of the previous layer before passing it into the fully connected layer. Next we apply the ReLU activation function to this first fully connected layer and pass our outputs to our final fully connected layer which has output dimensions of five for our five classes in our classification task. Finally, we end the network by applying the softmax activation function to our final fully connected layer in order to attain the probabilities that each image class corresponds to the image being passed in.

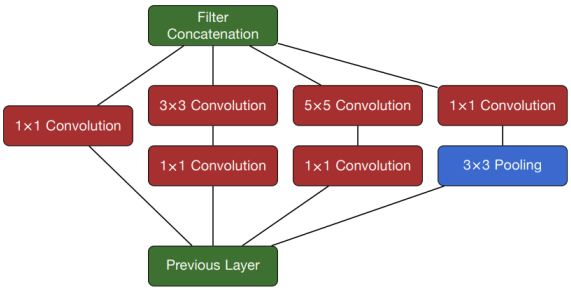

FIG. 5. Inception Module

ResNet50 (Fig. 6) is known for its iconic skip connections and its strong capability to classify image data. ResNet50 was one of the models we want to compare classification performance with CVN. However, they are quite different in the number of parameters. To make the 2 models' number of parameters comparable, we decided to exclude ResNet50's conv5 block. A detailed description of ResNet architecture can be found in [2]. The model becomes ResNet41 7, as there are 9 layers in the cfg3 block that we excluded. ResNet41 is the second model we ended up using, instead of ResNet50. When training ResNet41, we used droupout 0.2 to prevent overfitting.

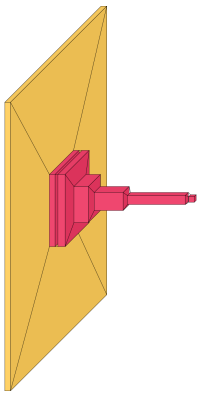

FIG. 6. Original ResNet-50 Network Visualization.

The third model we chose is a mix of CVN and

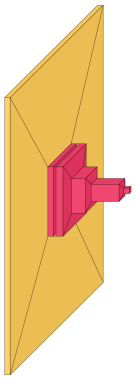

FIG. 7. ResNet-41 Network Visualization.

ResNet. We replaced all the CVN's inception modules with ResNet's skip connections. We name it '2 view ResNet.' The architecture backbone of the network is from CVN, and we swapped out the inception module and use the skip connection from ResNet.

For all 3 networks, we use Adam as the optimizer, cross entropy as loss function, initial learning rate of 0.0005. And we train each of the nework for 150 epochs. 2-view ResNet has 3.12 million parameters. The number of parameters of NesNet41 is around 3.78 million. CVN has around 6.29 million parameters.

IV. RESULTS

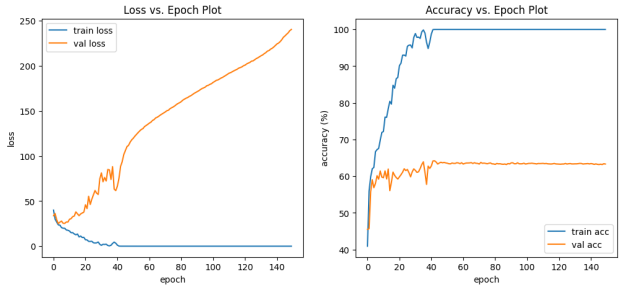

FIG. 8. CVN Accuracy and Loss Curves

Fig.8 shows overfitting of network and this could also be due to only utilizing 10% of the dataset for training and another 10% for validation. We can see from the start that our training data is achieving a lower loss than our validation data which means that our training data could be memorizing our training data and it's not performing well on unseen data.

In our accuracy curves of our CVN network in Fig. 8 we can also see that that we achieve a much higher accuracy with our training data while our validation data peaks at an accuracy of around 60%.

The results of ResNet41 Fig.9 and 2 view ResNet Fig.10 both shows some amount of overfitting like CVN.

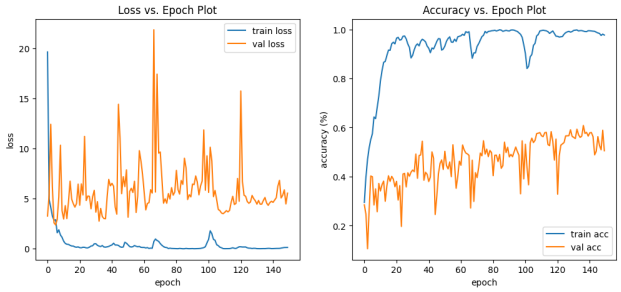

FIG. 9. ResNet41 Accuracy and Loss Curves

And we believe the analysis we did in CVN can also be applied here.

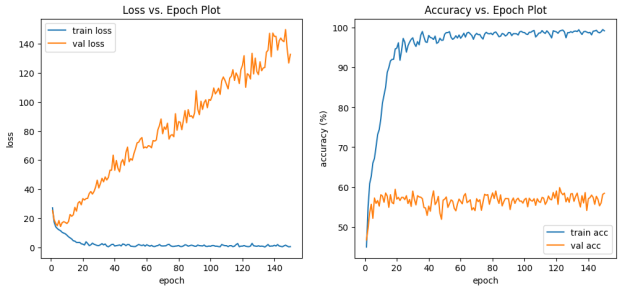

FIG. 10. 2-view-ResNet Accuracy and Loss Curves

V. CONCLUSION

Due to the limited CPU speed on DSMLP, we are not able to load the whole dataset onto DSMLP. We chose to only use 10% of the dataset as training data, and another 10% for testing and validation purposes. These two 10% subsets are mutually exclusive. Once we switch to a different platform which could allow us to train a even larger dataset, we expect to see less overfitting in all 3 graphs. Based on the accuracy plots of the 3 networks, it is **still unclear** which network would be the best classifier, because all 3 of them exhibits some degree of overfitting. But based on the performance of our 3 models after 150 epochs of training, CVN has maximum validation accuracy of 64.2%, ResNet41 and 2-view ResNet achieved 60.8% and 59.8%, respectively.

All of our code for this project can be found in our github page.

[1] A. Aurisano, A. Radovic, D. Rocco, A. Himmel, M. Messier, E. Niner, G. Pawloski, F. Psihas, A. Sousa, and P. Vahle, A convolutional neural network neutrino event classifier, Journal of Instrumentation **11** (09), P09001.

[2] K. He, X. Zhang, S. Ren, and J. Sun, Deep residual learning for image recognition, in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2016) pp. 770–778.