Neuronal Components Classification of Calcium Imaging Data

Shangying Jiang[1], Yang Sun[1], Yidi Zhang[1], Yiran Xu[1], Zhaopeng Liu[1]
Andrea Giovannucc[2], Eftychios A Pnevmatikakis[2]

[1] CENTER FOR DATA SCIENCE, NEW YORK UNIVERSITY, NEW YORK, NY, USA, 10011
[2] SIMONS FOUNDATION, NEW YORK, NY, USA, 10010

Abstract

Convolutional neural network (CNN) has been proved to be very effective in image classification domain by many recent publications (4). In this paper, we introduce our work to explore CNN-based approaches to classify neuronal components within images. Previous work by Andrea and Eftychios (our mentors) has achieved an accuracy of 0.950 in binary classification. This showed us an impressive result in classification of neurons and non-neurons. Based on these work, we constantly improved the classifier by studying regular CNN model, VGG-like architectures (1) and residual learning networks (ResNet) (2) to categorize four types of components. In training these models, we revised the original model introduced in (1; 2), which best suits our dateset. Our baseline model, VGG-like models and ResNet produced optimal accuracy of 0.801, 0.903 and 0.873, respectively.

1 Introduction

High-resolution monitoring of brain activity in behaving animals has recently been made possible by calcium imaging techniques. Analyzing such data entails the identification of the neuronal types and activities of components within the images (3). Our major task would be to build models to classify components extracted from calcium images into four classes: neurons, doubtful neurons, processes and noises.

AlexNet (4) is considered as a breakthrough in the image classification domain along with the improvement of computing hardware. Soon after, VGG-like models increased depth of AlexNet applying 3×3 convolutional filters for image recognition. Later, ResNet further improved deep neural network under residual learning framework while adding more layers and reducing parameters to gain validation accuracy. These attempts inspired us to explore and optimize our models to categorize the four types of neuronal objects.

The existing classification algorithm to identify neurons and noise artifacts in the extracted images has shown a satisfactory accuracy of 0.95, and our baseline model also gave a decent accuracy of 0.80 for the four-type components' classification. In this project, we first manually labeled components from two classes (neurons and non-neurons) into four classes (neurons,doubtful neurons, processes and noises); then trained various models to optimize the classifier.

2 Data

We obtained 9028 50×50 pixels gray-scale images from calcium imaging videos that monitored large neuronal objects populations. Each image contains one component that our classifier needs

to identify. We generated a label and a 50×50 numpy array [1] that contains pixel values for each component. Our models take a 50×50 array as input and compute the probability of components in each class. To train good-performing models, two important tasks needs to be performed carefully with respect to our dataset: labeling and cleaning.

2.1 Labeling

The original model is a binary classifier which can only classify $\{ClearNeuron, Nonneuron\}$. Our first job is to relabel all components into three classes $\{ClearNeuron, Process, Noise\}$ for building three-class classifiers. We could tell components' types by visually detecting the contours of components in each class [2]. It is worth noting that this effort requires the development of a substantial manually annotation skill.

2.1.1 Three-class labeling

As shown in Figure 1, clear neurons (left) usually appear as doughnuts with clear outline, and possibly have dendrites connected to them; processes (middle) are collection of possible dendrites which only appear as one of three forms: tiny bright dot, line segment or treelike structure. Noise (right) is a class for components that do not have any distinguishable structure. Figure 1 shows typical example of components of each class. Unfortunately, after going through the entire dataset, we found that most components were much less visually recognizable as those appeared in Figure 1.

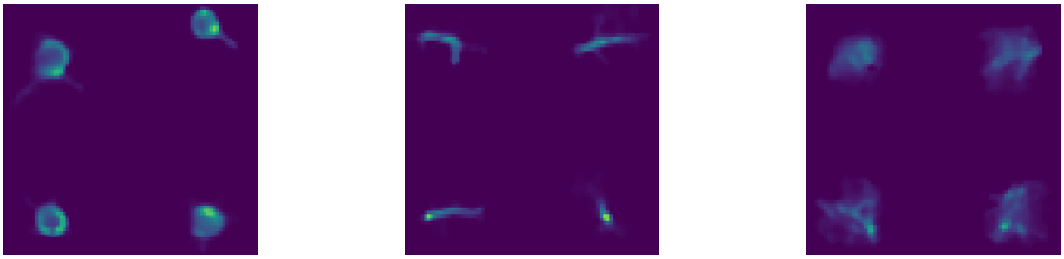

Figure 1: Example for components (clear neurons, processes, and noises from left to right)

To make labeling more efficient, we took advantage of original binary classifier that achieved 95% accuracy. First, we included all the components with positive/negative labels into the classifier (positive indicates clear neuron and negative indicates non-neuron); then we picked out processes and noises from positive components so that the positive class contained purely clear neurons. Finally, we examined the negative class and picked out neurons and processes out of the negative class so that it contained all the purely noise.

To reduce bias of human annotation, we did not split the data and label components individually. Instead, we went through all components together and assigned labels by consensus [3]. However, our first attempt of labeling was ineffective because of inconsistency: some components that share the same spatial features were assigned into different classes. As we encountered new components with unseen spatial features, our opinions of a certain class often differed. We also found that labeling in the entire team was time-consuming.

2.1.2 Four-class labeling

To tackle inconsistency of labeling, we decided to put all the dubious components into in a new class as $DoubtfulNeuron$. This class contains all the components that have blurry contours, yet some recognizable structure. Figure 2 shows examples of doubtful neurons. The top left one in Figure 2

[1] Numpy is a library for Python Language.

[2] We thank our mentors for their domain knowledge guidance on neuronal objects' classification by their appearances.

[3] When we disagreed on one label of a certain component, we voted and went with the majority.

has bright dot but its size is too big for a process. The other three do not have clear outline though they are quite round. In conclusion, components in this class appear as: a) round shape with blurry outline, b) bright dot with large size, c) either a) or b) partially shaded by cloudy noise.

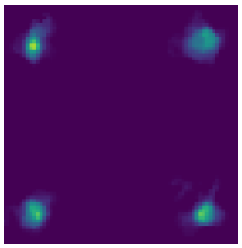

Figure 2: Examples for doubtful neurons

To assess the necessity of the fourth class, we constructed four different datasets: a) combination of all the doubtful neurons with clear neurons, b) combination of all the doubtful neurons with processes, c) combination of all the doubtful neurons with noises, d) exclusion of all the doubtful neurons.

Surprisingly, we found that with doubtful neurons excluded from then entire dataset, we achieved a validation accuracy of 94.46%. This suggests that this class of components confused not only human labeling, but also the training algorithm. Therefore, we introduced the fourth class, $Doubtful Neuron$, to prevent such confusion.

2.2 Cleaning

We encountered several components that have rare spatial features such as honeycomb structure. Since these components have clear structure, we could not label them as noises yet they are apparently not neuron nor dendrites. We decided to remove these rare components out of the dataset.

One interesting finding during cleaning process was that our baseline classifier could correctly classify components that were mistakenly labeled by human annotation. For example, we found that, for some clear neurons, if we labeled them as processes, the baseline model still classified it as clear neuron. This demonstrated that the algorithm were able to outperform human labeling in some cases.

We hypothesized that some indistinguishable dark pixels within the image were not able to be detected by human eyes, but these dark pixels would not cause any problem in computer vision field. Therefore, we concluded that baseline model could already recognize structures or patterns that are easily overlooked by human eyes. Taking this into account, we carefully checked all the misclassified components and relabeled them accordingly.

After labeling and cleaning, we have a dataset of 8981 components in total. 2029 clear neurons, 1840 processes, 2156 noises and 2956 doubtful neurons.

2.3 Data Augmentation

As mentioned previously, we did not have a large dataset to train our model, so we applied data augmentation techniques to prevent overfitting by artificially enlarging the dataset using label-preserving transformations (4). The augmentation techniques we used including rotating components randomly from 0 to 360 degrees, horizontally and vertically shifting or flipping random components etc.

3 Baseline Model

3.1 Architecture

Three types of learning layers are essential in building a CNN model: convolutional layer, pooling layer and fully-connected layer (FC layer). The architecture design is shown in Table 1 [4].

Table 1: Baseline model architecture and hyper-parameter settings. In the table, ReLU is not shown for simplicity.

baseline architecture	parameter size
conv-32	3×3
maxpool	2×2
dropout	0.25
conv-32	3×3
maxpool	2×2
dropout	0.25
conv-64	3×3
maxpool	2×2
dropout	0.25
conv-64	3×3
maxpool	2×2
dropout	0.25
FC	512
dropout	0.5
FC	number of classes

3.2 Layer Parameters

Like other regular neural network, CNN consists of layers of neurons that have learnable weights and biases. Each neuron performs a dot product between the input and weights. For our baseline model, each convolutional layer is followed by a non-linear element-wise activation function ReLU. ReLU takes the form:

$$\sigma(x) = max(0, x)$$

which performs most efficiently for image classification neural network (4).

Regular neural networks cannot take images as input efficiently since large-size images need a large number of weights to represent. For example, a RGB image of size 224×224 would need $224 \times 224 \times 3 = 150,528$ weights for a learning neuron (5). Such huge number of parameters will easily cause overfitting.

To prevent overfitting, it is common to add pooling layers to down-sample parameters periodically between convolutional layers. As shown in Table 1, we employed max-pooling layer of pooling size 2×2 in our baseline model. Another effective regularization methodology is to add dropout. In the model, we set dropout of rate equal to 0.25 following every pooling layer, and added dropout of rate 0.5 after FC layer.

For convolutional layers, we set the kernel size as 3×3, and stride as 1. Given a valid zero-padding, the small receptive window size can not only capture detailed information of the images, but also keep the sizes of feature maps the same as input images' sizes. Fully connected layer (a.k.a "Dense layer") are the last few layers of all CNN-based models. Among all the fully connected layer, the last one follows the form of a softmax function:

$$\sigma(x)_j = \frac{e^{x_j}}{\sum_{j=1}^{K} e^{x_j}}$$

[4] Table 1 shows the architecture and hyper-parameter settings of our baseline model. "conv-n": "convolutional layer-number of filters"; "FC-n": "FC-number of neurons within FC layer". In addition, unless otherwise noted, all convolutional layer has kernel size = 3×3, stride = 1 and a valid zero-padding size.

which calculates probability of each class and hence performs the classification task. We set different numbers of FC layers, depending on the depth of the model.

3.3 Implementations

All of our models are implemented using Keras and ran on cluster GPU. Since the task we performed is a multi-classification problem, we used the categorical cross entropy loss function:

$$L(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} log(p_{ij})$$

where n is the batch size, m is the number of classes. We used RMSprop optimizer to perform stochastic gradient descent. This is a very effective optimization method applying adaptive learning rate (6). During the training stage of the model, we set the batch size to be 256, with learning rate and regularizer of decay equal to 5×10^{-5} and 1×10^{-6}, respectively. For all architectures designed, we ran 1000 epochs to monitor the accuracy and loss change by drawing learning curves. All learning curves are demonstrated in the Appendix A.

For baseline model, as our dataset has similar features as CIFAR-10 (7), we started with hyper-parameter settings that are closer to CIFAR-10.

3.4 Conclusion

The basic CNN model reaches an accuracy around 0.801 on validation set. (Result is shown in Table 7). During error analysis, we found some misclassified components. Thus, we believe there are margins to improve the classification results by designing better performing models. In Section 4 and 5, we introduce VGG-like architectures and ResNet that we explored, which gave us more insightful thinking of CNN-base approaches in image classification domain.

4 Very Deep Convolutional Neural Network

4.1 Introduction

During training, the input to our models are fixed-size 50×50 gray-scale images. Motivated by (1), we trained an outperforming classifier to categorize our four targeted classes based on their spatial, contour and other features. The optimal validation accuracy achieved reaches 0.903 with VGG-like 16-layer CNN model. The training and testing procedures were essentially the same as those used in baseline model construction.

4.2 Architecture

For baseline-variant and VGG-like architectures, convolutional, pooling and FC layers have the same parameter settings as those in our baseline model. The only difference is the depth of models.

4.2.1 Baseline-variant Architecture

Our baseline model is constructed with maximum number of 64 filters within convolutional layers of 3×3 kernel size. We increase the number of convolutional filters by factor of 2 to 64, 128, 256 and 512, each followed by a pooling layer and dropout regularization. Holding all hyper parameters the same as in the baseline model, baseline-variant achieved validation accuracy of 0.835. Although it outperforms the baseline model, the improvement is still unsatisfactory.

We conjecture that compared to the 224×224 RGB image dataset used in (1), our input images of single channel and smaller sizes have much fewer features and parameters. Therefore, during the training process convolutional layers with 512 filters may yield too many features that may cause overfitting and lead to accuracy drop. Hence, in later architecture designs, we will not apply any convolutional layers with 512 filters.

4.2.2 VGG-like Architectures

The detailed configurations of our VGG-like architectures are shown in Table 2. To feed images to the model, we applied small 3×3 receptive window to capture images' information, and gradually increased the number of layers from 9 to 21, as well as number of filters, from 32 to 256. In addition, in each architecture, each stack of convolutional layers is equipped with one pooling layer and one dropout. Finally, we set different numbers of FC layers for different architectures, followed by ReLU and dropout.

Table 2: Configurations of VGG-like architectures from 11 layers to 19 layers.

11 layers	13 layers	16 layers	19 layers
conv-32 ReLU	conv-32 ReLU	conv-32 ReLU	conv-32 ReLU
conv-32 ReLU	conv-32 ReLU	conv-32 ReLU	conv-32 ReLU
2× conv-64 ReLU	2×conv-64 ReLU	2×conv-64 ReLU	2×conv-64 ReLU
maxpool & dropout	maxpool & dropout	maxpool & dropout	maxpool & dropout
3×conv-128 ReLU	3×conv-128 ReLU	3×conv-128 ReLU	4× conv-128 ReLU
maxpool & dropout	maxpool & dropout	maxpool & dropout	maxpool & dropout
2×conv-256 ReLU	2×conv-256 ReLU	3×conv-256 ReLU	4×conv-256 ReLU
	maxpool & dropout	maxpool & dropout	maxpool & dropout
	2×conv-256 ReLU	3×conv-256 ReLU	4×conv-256 ReLU
maxpool & dropout	maxpool & dropout	maxpool & dropout	maxpool & dropout
FC-512	FC-512	FC-512	FC-1024
softmax	softmax	FC-512	FC-1024
		softmax	softmax

In all architecture designs, we have two major distinctions from the standard VGG-like architectures in (1), discussed below.

Fewer ReLU. In (1), each hidden layer was equipped with one ReLU, but our convolutional layers(except for input layer) are followed by one ReLU when the number of filters of a convolutional layer increases by a factor of 2 (except when the number of filters exceeds 256). This construction was inspired by the "direct connections" in DenseNet(8). We conjecture that extra non-linear activation functions are impediment in training process for image with fewer features. For neuronal type classification, the dominant feature is contour, so we believe equipping only one ReLU for the stacking layers will improve information flow and accelerate learning. We compare the training results Table 3.

Table 3: Fewer ReLU. Architectures followed by "-relu" indicates that all convolutional layers within the architecture are followed by ReLU.

architecture	validation accuracy	running time
VGG-like 9 layer-relu	0.834	4s/epoch
VGG-like 9 layer	0.885	3s/epoch
VGG-like 16 layer-relu	0.891	19s/epoch
VGG-like 16 layer	0.903	3s/epoch

More Dropout. In (1), each FC layer was followed by one dropout, but none convolutional layers was followed by any dropout. Due to fewer features and parameters, accuracy gets saturated faster in our models, so we believe our architectures will benefit from more random drop. We added dropout preceding to every pooling layer, and both perform effective regularization. In Table 4, "VGG-like 16 layer-nodrop" and "VGG-like 16 layer" have almost identical validation accuracy, but the former

has a much larger training accuracy, so it is reasonable to argue that the former architecture may encounter overfitting, which can be eased by extra dropout. In addition, we found that adding more dropout in the model significantly decreases running time. The detailed results are shown in Table 4.

Table 4: More dropout. Architectures followed by "-nodrop" indicates that dropout regularization is not added after convolutional layers.

architecture	training accuracy	validation accuracy	running time
VGG-like 16 layer-nodrop	0.894	0.900	18s/epoch
VGG-like 16 layer	0.861	0.903	3s/epoch

These two variations of a standard VGG-like architecture have shown us that CNN-based approaches in the image recognition domain has much variability. Different designs of architectures can suit to various scales of dataset. However, this advantage may also cause negative side of a model, for example, low generalizability.

4.3 Problem

4.3.1 Generalizability

One possible problem of our optimal VGG-like 16-layer model is that it is unlikely to fit all datasets. For example, ImageNet (14) has 1000 target classes and each image has a much larger size. ImageNet has completely different features from our dataset, it is reasonable to argue that our model will not perform effective classification on ImageNet. Due to time and hardware usage constraints, we were unable to test this architecture design on other dataset with similar features (sizes, number of classes needs to categorize etc.) and similar-sized input images. This problem is only specific to our dataset, and is still an open question.

4.3.2 Redundant Features Generation

Each convolutional layer of VGG-like models' filter needs to extract different information from preceding layers, for instance, color, edge, curve feature detectors etc., so following layers may produce similar features as previous layers, especially when features within the input is poor. This problem may cause learning layers to output redundant features that are not effective for classification. Although random drop may alleviate this problem, but too much dropping may also cause information loss.

4.3.3 Degradation

Although VGG-like architectures produced impressive results for our input image classification, they became difficult to train with increasing number of layers, which may cause *degradation*(2). The result can be observed in Table 5: when number of layers increases from 11 to 21, the training accuracy reaches 0.867 and started to drop. Furthermore, training models with more layers significantly increases running time. The detailed result is shown in Table 5. An almost seven times jump in running time for 19-layer model to 21-layer model is unacceptable. Fortunately, degradation and lengthy training time can be addressed by recent development of ResNet (discussed Section 6).

Table 5: Training accuracy and running time of VGG-like models from 11 layers to 21 layers, holding other parameters the same.

architecture	training accuracy	running time
VGG-like 11 layer	0.801	3s/epoch
VGG-like 13 layer	0.867	3s/epoch
VGG-like 16 layer	0.861	3s/epoch
VGG-like 19 layer	0.857	4s/epoch
VGG-like 21 layer	0.856	27s/epoch

4.4 Conclusion

In this section, we evaluated various VGG-like models of different depth. The optimal architecture we built is VGG-like 16-layer architecture. We achieved training and validation accuracy of 0.861 and 0.903, respectively with running time 3s/epoch. Section 7 shows all detailed parameter settings and models' results. In next section, we will demonstrate ResNet that we explored, which helps us to alleviate degradation.

5 Residual Learning Neural Network

Although VGG-like models can produce very good results for large scale image classification, training time increases significantly and training error gets saturated with increased depth. This problem can be addressed using a deep residual learning framework. The core idea of Residual Learning Neural Network (ResNet) is to provide shortcut connection between layers, which makes it feasible to train very deep network to gain maximal representation power without degradation problem.

5.1 Vanishing Gradient

Let us consider the case in a N-layer neural network. The empirical cost decreases when the weights are adjusted in the direction according to the gradient. The wights are adjusted as follows:

$$\hat{W_{ij}} = W_{ij} - \alpha \frac{\partial C}{\partial W_{ij}}$$

where C is the cost function, W_{ij} is weight between layer i and layer j. Every layer's weight is updated in this way from the output of the preceding layer, and the update is repeated until $\frac{\partial C}{\partial W}$ approaches zero. In this case, according to chain rule:

$$\hat{W_{ij}^1} = W_{ij} - \alpha \frac{\partial C}{\partial W_{ij}^n} \frac{\partial C}{\partial W_{ij}^{n-1}} \cdots \frac{\partial C}{\partial W_{ij}^2}$$

when the number of layers increases, the gradient decreases, which might lead to a smaller update in following layers. To address problem, we can simply use larger input dataset, but the network will spend much longer time to learn representation from a larger-scale input.

Besides, in deep neural networks, the learning result of bottom layer (a.k.a "high-level feature map" (8)) is largely dependent on the previous layers, so when the top layers produce relatively "low-level" feature maps, such as lines and edges of the input images, the subsequent layers take preceding layers' output as input. As a result, if the previous learning layers take more time to converge due to vanishing gradients (2), the deep training layers will not learn effectively from preceding layers. In other words, regular deep neural network may generate poor classification result.

Another issue caused by vanishing gradients is that if the gradient at each layer is small, then more iterations are going to be added until model finally converges, which indicates the running time of a deeper neural network will increase significantly.

The vanishing gradient might be one of the underlying reason beneath degradation of training accuracy introduced in Section 4. Residual neural networks are proposed to solve this problem and gain maximal representation power.

5.2 Residual Learning

The degradation phenomena shows that it is impractical to find an optimal number of learning layers of a neural network: if the architecture designed has too many layers, the back-propagation of training loss may fail; if the neural network is too shallow, the network may not be able to learn enough representations from input images, thus leading to a unsatisfactory result.

Different from deep VGG-like architectures, adding layers of a neural network with short connection method introduced in (2) avoids adding more parameters, thus adding no extra running time or

computation usage. As a result, we can fairly compare the VGG-like architectures and ResNet models we designed. All the results of ResNet model we trained is shown in the Appendix B.

5.3 Network Architectures

We added short connections to the VGG-like architectures we constructed in section 4. The detailed configuration is shown in Figure 3. Table 6 demonstrates neural network under residual learning framework, as in (2).

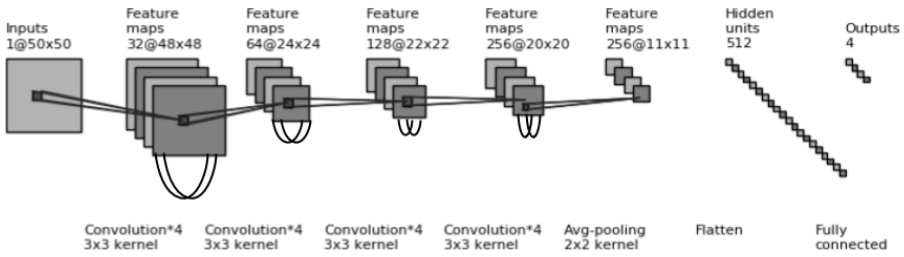

Figure 3: A residual network with 18 learning layers

Table 6: Configurations of ResNet architectures from 18 layers to 101 layers.

18 layers	34 layers	50 layers	101 layers
7,7, 32, stride 2 3,3 max pool, stride 2	7,7, 32, stride 2 3,3 max pool, stride 2	7,7, 32, stride 2 3,3 max pool, stride 2	7,7, 32, stride 2 3,3 max pool, stride 2
$\begin{bmatrix} 3\times 3 & 32 \\ 3\times 3 & 32 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3 & 32 \\ 3\times 3 & 32 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1 & 32 \\ 3\times 3 & 32 \\ 1\times 1 & 128 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1 & 32 \\ 3\times 3 & 32 \\ 1\times 1 & 128 \end{bmatrix} \times 3$
$\begin{bmatrix} 3\times 3 & 64 \\ 3\times 3 & 64 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3 & 64 \\ 3\times 3 & 64 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times 1 & 64 \\ 3\times 3 & 64 \\ 1\times 1 & 256 \end{bmatrix} \times 4$	$\begin{bmatrix} 1\times 1 & 64 \\ 3\times 3 & 64 \\ 1\times 1 & 256 \end{bmatrix} \times 4$
$\begin{bmatrix} 3\times 3 & 128 \\ 3\times 3 & 128 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3 & 128 \\ 3\times 3 & 128 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times 1 & 128 \\ 3\times 3 & 128 \\ 1\times 1 & 512 \end{bmatrix} \times 6$	$\begin{bmatrix} 1\times 1 & 128 \\ 3\times 3 & 128 \\ 1\times 1 & 512 \end{bmatrix} \times 23$
$\begin{bmatrix} 3\times 3 & 256 \\ 3\times 3 & 256 \end{bmatrix} \times 2$	$\begin{bmatrix} 3\times 3 & 256 \\ 3\times 3 & 256 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1 & 256 \\ 3\times 3 & 256 \\ 1\times 1 & 1024 \end{bmatrix} \times 3$	$\begin{bmatrix} 1\times 1 & 256 \\ 3\times 3 & 256 \\ 1\times 1 & 1024 \end{bmatrix} \times 3$
avgpool & dropout	avgpool & dropout	avgpool & dropout	avgpool & dropout
FC-512	FC-512	FC-512	FC-1024
softmax	softmax	softmax	softmax

The identity shortcuts can be directly performed when the input and output are of the same dimensions. As input dimensions increase, the shortcut connection will keep performing identity mapping, with valid extra zero padded to fit dimensions of previous layers' output. We made some changes on number of filters specific to our dataset.

5.4 Implementation

Our implementation for neural classification generally follows the practice in ResNet (2). To train neural nets for image classification, a large amount of image data is required. Therefore, data augmentation, such as shears, rotations, shifts, flips, zooms, etc., are used to create new data and balance training data set.

Our image size is 50×50 with color channel 1. First, we trained ResNet model without batch normalization (9). Then, we made attempt to find the optimal place to add batch normalization. We compared following results: a) add batch normalization layer before convolution layer; b) add batch

normalization layer after convolution layer; c) add batch normalization layer before ReLU; d) add batch normalization layer after ReLU.

We initialized the weights as in (10) and trained all residual nets, with depth of 18, 34, 50, 101 from scratch. The results are shown in Table 8. We did not apply pre-trained weight because our dataset has some special features; for example, the dominant feature of a component is the contour of the object. We used Adam (11) and Stochastic Gradient Decent (12) optimizers with batch sizes of 32 and 64. We used reduced learning rate, which updated every 5 epochs, and the minimal learning rate is 5×10^{-5}. The models are trained for up to 200 epochs with early stopping. We used dropout 0.5.

5.5 Conclusion

To sum up, the core idea of ResNet is to provide shortcut connection (we adopt "Identity mapping" introduced in (2)) between stacked layers and it makes easier to train deeper networks to gain maximal representation without worrying about the degradation and increased training time.

6 Experiment and Result

6.1 Evaluation Method

The evaluation metric we use is accuracy, which shows how often predictions have maximum in the same spot as true values. We evaluate our method on the validation data set that consists of 2246 images. The models are trained on the 6735 training images. We also obtain a final result on the 500 test images.

6.2 Model Results

All layers' parameter settings of VGG-like models are identical as our baseline model for comparison purpose. The detailed result is shown in Table 7.

Table 7: Model results of baseline model, baseline variant model and VGG-like architectures.

architecture	training accuracy	validation accuracy	running time
baseline 6-layer	0.807	0.801	3s/epoch
baseline-variant	0.793	0.835	6s/epoch
VGG-like 9-layer	0.865	0.890	3s/epoch
VGG-like 11-layer	0.856	0.874	3s/epoch
VGG-like 13-layer	0.867	0.895	3s/epoch
VGG-like 16-layer	0.861	0.903	3s/epoch
VGG-like 19-layer	0.857	0.903	4s/epoch
VGG-like 21-layer	0.856	0.891	27s/epoch

To visually detect the effect of our two variations made to the VGG-like models, we demonstrate the following four plots of learning curves.

As shown in Figure 4, in the left plot (our best-performing VGG-like 16-layer model), the blue curve (training accuracy) lies below the red curve (validation accuracy) during the whole training procedure; in the right plot (in which each convolutional layers is followed by ReLU), the two learning curves are overlapping after around 200 epochs. For our dataset, the original architecture design in (1) is clearly sub-optimal. In addition, there is a spike of loss in around 500 epochs, which might be caused by degradation of training accuracy (equivalently, spike in training loss).

As shown in Figure 5, in the left plot (our best-performing VGG-like 16-layer model), the blue curve (training accuracy) lies below the red curve (validation accuracy), whereas in the right plot (the VGG-like 16-layer-nodrop, in which no dropout is added after convolution layers), the validation curve is overlapping with the training curve during the whole training process. We believe this gap shows an inherent regularization effect of dropout as extra dropout added prevents extra learning parameters from fitting into the model.

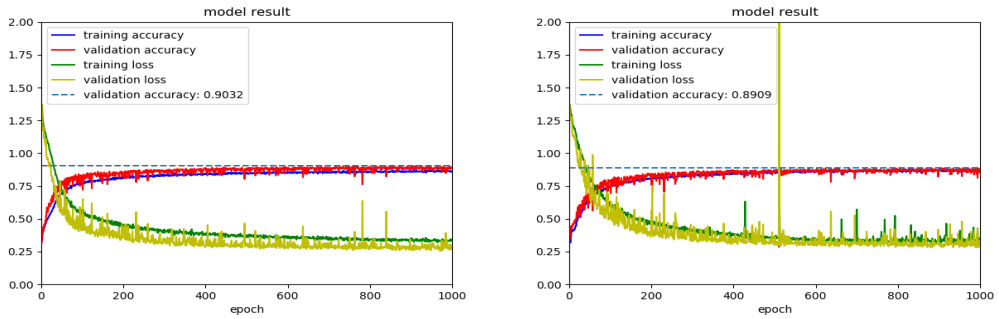

Figure 4: Learning curves for VGG-like 16-layer models. VGG-like 16-layer (left) and VGG-like 16-relu (right)

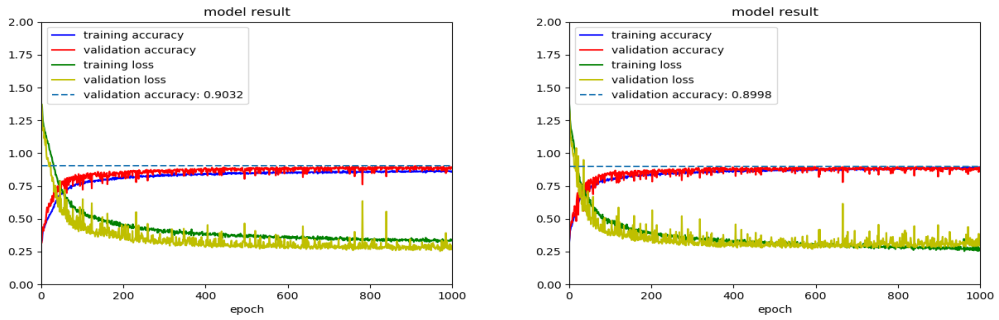

Figure 5: Learning curves for VGG-like 16-layer models. VGG-like 16-layer (left) and VGG-like 16-nodrop (right)

6.2.1 ResNet result

We used a reduced learning rate with patience 5 and early stop. We adopted batch normalization after each convolution layer and before activation layer, following (9). residual nets were trained with depth 18, 34, 50, 101, from scratch. These models are trained with a minibatch size of 32 on one GPU. We used shears, rotations, shifts, flips, zooms, etc., to create new data and balance training data set. Models' result and training time are shown in Table 8.

Table 8: Model results of baseline model, baseline variant model and VGG-like architectures.

architecture	training accuracy	validation accuracy	running time
ResNet-18	0.830	0.836	22s/epoch
ResNet-34	0.857	0.865	48s/epoch
ResNet-50	0.871	0.873	50s/epoch
ResNet-101	0.863	0.867	99s/epoch

All the models are compared using optimal parameter configuration. As we can see from behaviors of ResNet in table 8, the ResNet-34 has the best performance and ResNet demonstrates accuracy gain as depth increases. We further explored full pre-activation ResNet, where batch normalization layer and relu activation layer are both adopted before weight layers (13). Though the accuracy is increased, but the training time is largely increaed in this case, so we do not employed full pre-activation ResNet in this case.

7 Future Work

7.1 Semi-automatic labeling

Even though human annotations are feasible for small dataset, accurately labeling components multiple times is still time consuming, and minor consistency error is also inevitable. Among four classes of components, processes are the most distinguishable from other components, as shown in Figure 1. In future work, we expect to write a script to capture the strongest signal from components within the processes class. The returned signal might be included in the parameters, which can help improve our efficiency and reduce misclassification error.

7.1.1 Pseudo-Labels

Pseudo-Labels are target classes for unlabeled data as if they were true labels. We pick up the class which has maximum estimated probability for each unlabeled sample (15). Namely,

$$y_i^p = \begin{cases} 1, i = argmax_{i'} f_{i'}(x) \\ 0, otherwise \end{cases}$$

where y_i^p is the output data, x is input data, and i is the picked class that has the maximum predicted probability.

In this project, we have a large number of unlabeled data. Usually, we can manually label a part of the raw data using the method we described in Section 2.1, or we can use pseudo label hyper-parameter exploration phase. After applying pseudo-label for unlabeled data, we can train a neural network on unlabeled dataset, and treat the algorithm as 'supervised learning'. The training process with labeled and unlabeled data start at the same time, and the same loss function should be applied to them. The loss is calculated as:

$$L = \frac{1}{n} \sum_{m=1}^{n} \sum_{i=1}^{C} L(y_i^m, f_i^m) - \alpha(t) \frac{1}{n'} \sum_{m=1}^{n'} \sum_{i=1}^{C} L(y_i^{'m}, f_i^{'m})$$

Due to the highly unbalanced dataset, the number of labeled data is much smaller than the number of unlabeled data. Thus, the value of α is crucial. If α is too small, model could not learn from unlabeled image effectively, and vice versa.

Adopting unsupervised learning phase, we can largely improve the classifier's accuracy. Since we estimate that human labeling accuracy is below 80%, it is likely that we may add bias during manual labeling process. Using semi-supervised learning, we can minimized human labeling error, and this pseudo-label based method has been proved to have the state-of-the-art performance in semi-supervised learning for neural networks.

7.1.2 Pseudo code for semi-supervised learning

Below is pseudo code for semi-supervised learning:

Input: unlabeled data x.
f_i are output units for predicting target, $i \in \{1, 2, ..., n\}$, n is number of class

While termination condition not met
For $j = 1, \ldots, m$ (assumes data is randomly permuted)
For i in $\{1, 2, ..., n\}$
If $i = argmax_{i'} f_{i'}(x)$
$y_i^p = 1$
ELSE
$y_i^p = 0$

$L_{unsupervised} = \frac{1}{n} \sum_{m=1}^{n} \sum_{i=1}^{C} L(y_i^m, f_i^m)$
$L_{overall} = \alpha L_{unsupervised} + L_{supervised}$
$\hat{W} = W - \lambda \frac{\partial L}{\partial W}$
End

7.2 Testing on other dataset

In future work, we expect to test our optimal models to perform classifications on other datasets of similar sizes, number of classes etc. We expect to increase the generalizability of our best-performing model. In addition, recent development of DenseNet (8) introduced "direct connections" to ease degradation of model's training accuracy, strengthen feature reuse while producing fewer parameters and requiring less computation power. We will compare DenseNet's accuracy, computing time with our optimal models.

7.3 Activities of neuronal components

Apart from identifying the components' types, we are also very interested in studying the spatial and temporal activities of all the components we classified. From the dynamics of neuronal components monitored by calcium indicator (the fluorescent protein that enables imaging), we will be able to detect the motions of neurons and processes. Most existing approaches in this field simply assume linear dynamics of the protein and concentration, but such assumption is sub-optimal and can lead to incorrect spike estimates.

Contribution Statement

Shangying Jiang: Labeled and cleaned dataset under the guidance of mentors. Explored performance of three-class classifiers by including/excluding the fourth class $DoubtfulNeuron$.
Yang Sun: Explored VGG-like architectures; trained all VGG nets specific to our dataset.
Yidi Zhang: Explored Residual Learning Neural Network. Trained all ResNets specific to our dataset.
Yiran Xu: Labeled and cleaned data with other team members. Explored baseline model.
Zhaopeng Liu: Labeled and cleaned data with other team members.

All five of us contributed in data cleaning and labeling. All five of us contributed in report writing.

Acknowledgements

During the project, we have received a lot of help from Simons Foundation. We sincerely thank them for the support of computing hardware on our models' implementations. We thank our mentors Andrea Giovannucci and Eftychios A. Pnevmatikakis for their comments on our data preprocessing, model construction and attention to details.

References

[1] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition". In ICLR, 2015.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, Jian Sun “Deep Residual Learning for Image Recognition”. Computer Vision and Pattern Recognition (CVPR), 2016.

[3] Pnevmatikakis, E. A., et al., Simultaneous denoising, deconvolution, and demixing of calcium imaging data. 2016

[4] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. In NIPS, 2012.

[5] Karpathy, Andrej. “Convolutional Neural Networks (CNNs / ConvNets).” CS231n Convolutional Neural Networks for Visual Recognition, Stanford University, Jan. 2017, cs231n.github.io/convolutional-networks/.

[6] Karpathy, Andrej. “Neural Networks Part 3: Learning and Evaluation” CS231n Convolutional Neural Networks for Visual Recognition, Stanford University, Jan. 2017, cs231n.github.io/neural-networks-3/ada/.

[7] A. Krizhevsky. Learning multiple layers of features from tiny images. Tech Report, 2009.

[8] G. Huang, Z. Liu, K. Weinberger, L. Maaten. Densely connected convolutional networks. In CVPR, 2017

[9] S. Ioffe and C. Szegedy. "Batch normalization: Accelerating deepnetwork training by reducing internal covariate shift". In ICML, 2015.

[10] K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In ICCV, 2015.

[11] Diederik P. Kingma, Jimmy Ba, "Adam: A Method for Stochastic Optimization", 3rd International Conference for Learning Representations, San Diego, 2015.

[12] Bottou, Léon. "Online Algorithms and Stochastic Approximations". Online Learning and Neural Networks. Cambridge University Press. ISBN 978-0-521-65263-6, 1998.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Identity Mappings in Deep Residual Networks". Computer Vision and Pattern Recognition (CVPR), 2016.

[14] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, et al. Imagenet large scale visual recognition challenge. arXiv:1409.0575, 2014.

[15] Dong-Hyun Lee. "Pseudo-Label : The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks". ICML 2013 Workshop.

Appendix A

Below we demonstrate our learning curves of training procedures for VGG-like architectures from 9 layers to 21 layers. The optimal VGG-like model's learning curve is located in the second row and second column.

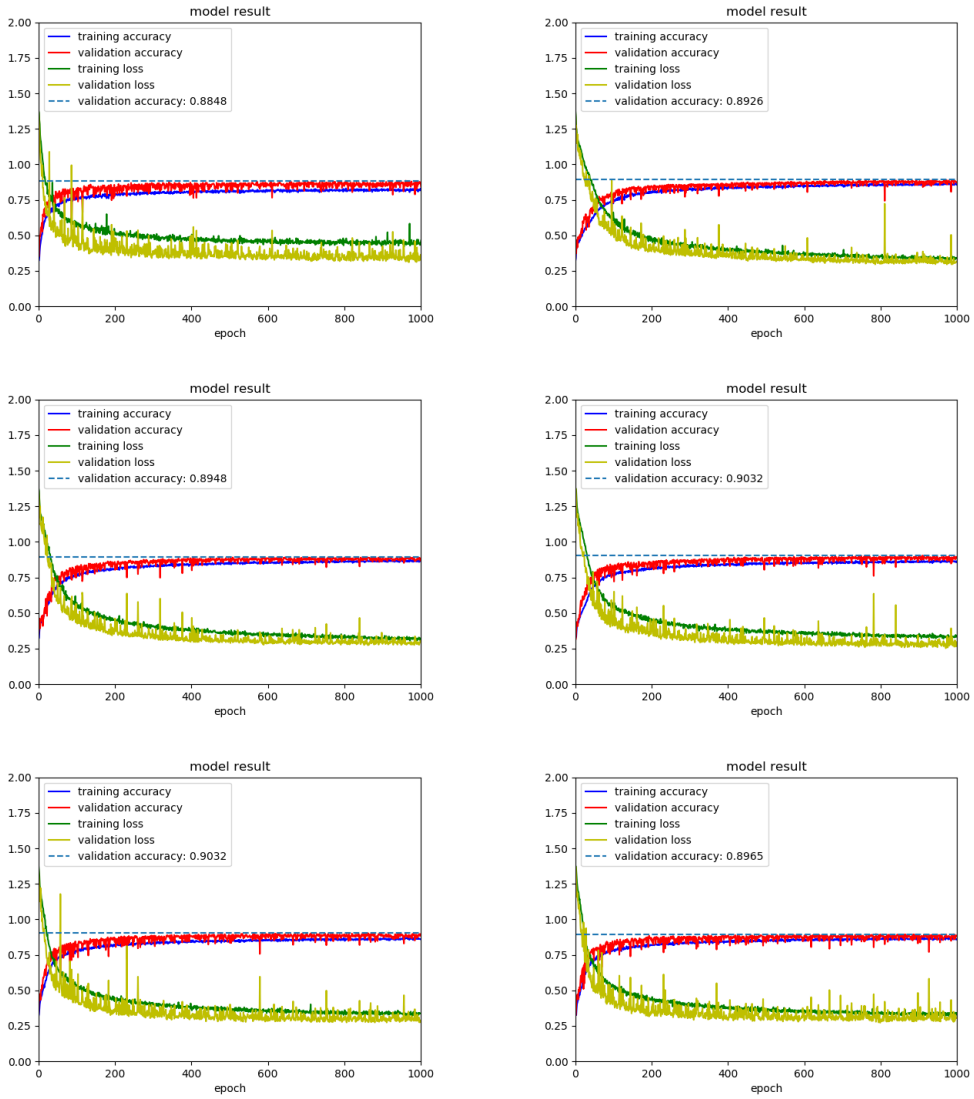

Figure 6: Learning curves for VGG-like models from 9 layers to 21 layers, from top left to bottom right.

Appendix B

Below is the model summary of ResNet 18-layer net that we constructed. The learning curve is shown in Figure 7.

```
_________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         (None, 50, 50, 1)         0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 25, 25, 64)        3200
_________________________________________________________________
```

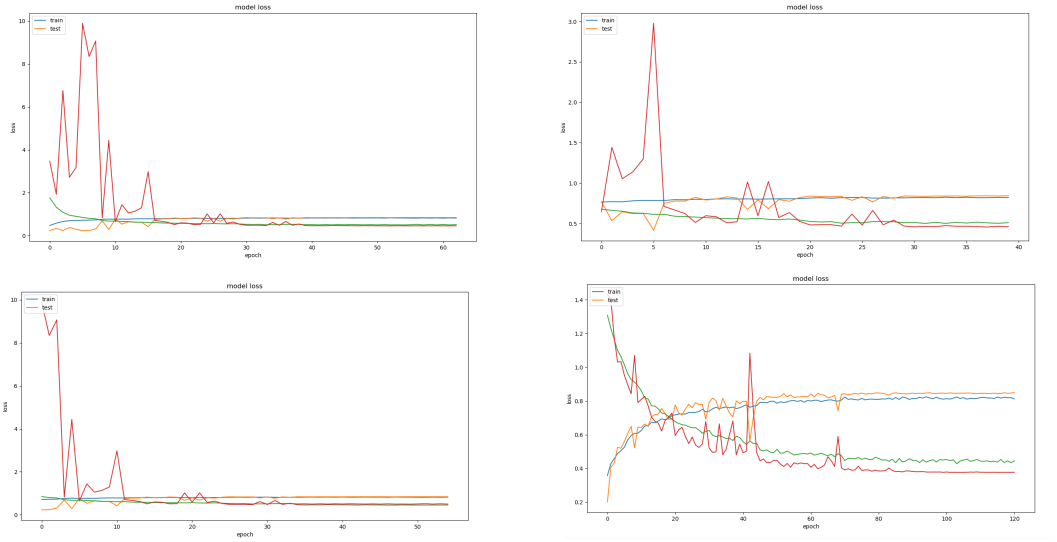

Figure 7: Learning curves for ResNet models 18, 34, 50 and fully pre-activation, from top left to bottom right.

```
batch_normalization_1 (BatchNorm (None, 25, 25, 64)     256
_________________________________________________________________
activation_1 (Activation)        (None, 25, 25, 64)     0
batch_normalization_1[0][0]
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D)   (None, 13, 13, 64)     0
_________________________________________________________________
conv2d_2 (Conv2D)                (None, 13, 13, 64)     36928
max_pooling2d_1[0][0]
_________________________________________________________________
batch_normalization_2 (BatchNorm (None, 13, 13, 64)     256
_________________________________________________________________
activation_2 (Activation)        (None, 13, 13, 64)     0
batch_normalization_2[0][0]
...  ...
...  ...
...  ...
_________________________________________________________________
activation_17 (Activation)       (None, 2, 2, 512)      0
batch_normalization_17[0][0]
_________________________________________________________________
average_pooling2d_1 (AveragePool (None, 1, 1, 512)      0
_________________________________________________________________
flatten_1 (Flatten)              (None, 512)            0
average_pooling2d_1[0][0]
_________________________________________________________________
dense_1 (Dense)                  (None, 4)              2052
=================================================================
Total params: 11,183,108
Trainable params: 11,175,300
Non-trainable params: 7,808
_________________________________________________________________
```

Appendix C

Below we show examples of well-classified and misclassified components produced by our classifier.

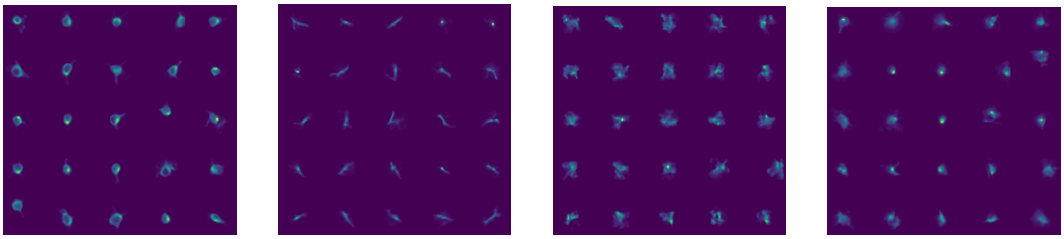

Figure 8: Example of well-classified neurons, processes, noises and doubtful neurons (from left to right).

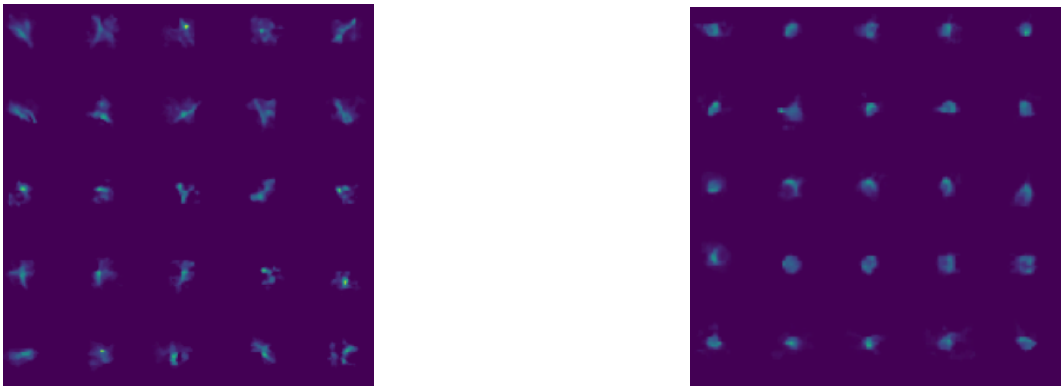

Figure 9: process classified as noise & doubtful neuron classified as clear neuron (from left to right).