CS230 Project: Protein Location Classification

Laura Cruz-Albrecht
Department of Computer Science
Stanford University
lcruzalb@stanford.edu

Abstract

Effectively identifying the location of proteins in the cell is a valuable task that can help provide deeper insight into the role of the protein within the cell. In this project, we apply various deep learning models to the multi-label, multi-class classification task of determining which cell organelles a protein is present in. We apply ResNet, DenseNet, and most uniquely, Capsule Networks, to this problem, and attain the highest performance (F1 score of 0.309) on the best tuned DenseNet model.

1 Introduction

Proteins are an integral part of cellular processes; identifying in which organelles the protein is present in can shed better insight into the both the role of the protein in the cell and into cellular mechanisms as a whole [1] [2]. This project applies deep learning to solve the following problem: given a microscope image containing a pattern of a protein, classify which organelles it is present in (there are 28 possible organelles). This task is part of the Human Protein Atlas Kaggle challenge [1].

The input to the deep learning models is an RGB image consisting of 3 stacked grayscale microscope images of a cell (one with the protein highlighted, and 2 with a reference organelle highlighted). We then apply various deep learning models - a simple CNN, ResNet, a Capsule Network, and DenseNet - to output the list of organelles that the protein is present in. Thus, this is a multi-class, multi-label classification task.

2 Related work

The realm of biology has seen rich applications of deep neural networks. A similar task of protein classification was tackled previously by [3] in a 2017 challenge; the task was based on the same dataset, but with 13 rather than 28 organelles, and the authors leveraged a fully convolutional neural network (CNN without FC layers).

Capsule networks are a newer type of neural network recently popularized by Hinton et. al [4] [5] [6]. Most notably, they allow neurons to output a vector rather than a scalar, and, unlike CNN's, preserve hierarchical pose relationships between object parts. This could potentially be helpful when looking at the relationship between protein patterns and different organelle structures. In [4], the authors apply the capsule network to MNIST dataset, and implement a variant that is able to classify multiple digits - thus, their loss function is robust to having multiple outputs.They have also been applied to various bioinformatics datasets [7], and to CIFAR [8]. However, while other methods such as CNNs have been applied to the protein organelle classification task [3], to the best of my knowledge Capsule Networks have not yet been applied to this task.

3 Dataset and Features

The dataset, from the Human Protein Atlas Image Classification challenge [1], consists of 31.1K labeled training samples, and 11.7K unlabeled test samples. Each sample has 4 associated 512x512 grayscale images: one is the protein of interest, and the other three are reference photos with a cellular landmark. The training samples are each labeled with a list of or more of 28 possible organelles. The test samples are unlabelled, as the challenge is to accurately generate these. A dataset sample is displayed in Figure 1. For development, we split the provided train set 80/20 into our own training and validation set, resulting in 24,857 training and 6,215 validation samples. Test performance will be calculated by Kaggle by submitting predictions on the provided test samples.

We then preprocessed the samples into an RGB image to feed into the deep learning models. Since many networks expect a 3-channel input, we stacked 3 of the 4 "filters": the one with the protein highlighted, and 2 of the 3 reference filters (for simplicity we chose the ones with the nucleus and microtubules highlighted, though a good future extension would be to try out a different combination, or leverage all 3).

As this is a multi-label, multi-class problem, we investigate two things: 1) the distribution of labels lengths (ie, how many classes are associated with each image); and 2) the distribution of classes (ie, in how many samples is each class present). The results are summarized in Figure 2. The uneven distribution of the label lengths and class counts, and the unknown distribution of the test set, adds a layer of complexity to the task at hand.

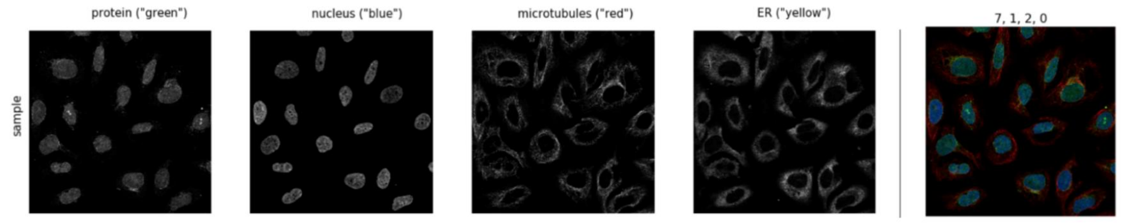

Figure 1: Data sample. Left: the 4 filters (protein + 3 reference images) for a single dataset sample. Right: The "green", "blue", and "red" filters superimposed to create the RGB image used as input to our deep learning models, and the associated class labels.

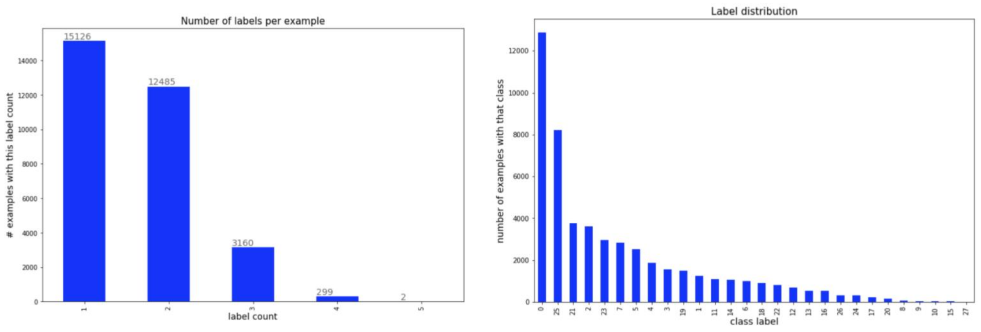

Figure 2: Distribution of label lengths (left) and class counts (right) in the training set. Most samples have one or two labels, and 0 and 25 (nucleoplasm and cytosol) are the most common classes by a large margin.

4 Methods

4.1 Simple CNN

As a baseline, we first tackle this problem with a simple convolutional neural network, consisting of 3 convolutional layers and 2 fully connected layers. Each of the convolutional layers is followed by batch normalization, max pooling, and a ReLU activation function. Dropout is used on the FC layers, and the final layer outputs a 28 x 1 vector. To allow for multi-class, multi-label prediction, we use the multi-label soft margin loss:

$$loss(x,y) = -\frac{1}{C} * \sum_i y[i] * \log((1+\exp(-x[i]))^{-1}) + (1-y[i]) * \log\left(\frac{\exp(-x[i])}{(1+\exp(-x[i]))}\right)$$

4.2 ResNet

Second, we approach this problem using ResNet, an architecture which features residual connections [9]. These residual connections facilitate learning by helping pass information forward through the network, and helps combat the vanishing gradient problem since the gradient can then be backpropagated to earlier layers more easily. We use transfer learning (using weights pretrained on ImageNet), and, to allow for multi-label prediction, use sigmoid for the final layer. We use the binary cross-entropy with logits loss:

$$l_n = -w_n \left[p_n y_n \cdot \log \sigma(x_n) + (1 - y_n) \cdot \log(1 - \sigma(x_n))\right]$$

The losses are averaged over each loss element in the minibatch. Note that the positive elements for each class can be weighted with a term p_n, which helps trade off recall and precision, and can help with the imbalanced nature of the classes. In Experiments, we experiment with different weighting schemes.

4.3 Capsule Network

Next, we approach this problem with Capsule Networks. This relatively new type of network preserves hierarchical pose relationships between object parts, which could potentially be helpful when looking at the relationship between protein patterns and different organelle structures. This network consists of "capsules" that output vectors rather than scalars, and the norm of the final capsule output vectors represents the probability of a class being present in the input (the algorithm uses a "squashing" function to ensure these final output vectors have norm between 0 and 1) [10] [4].

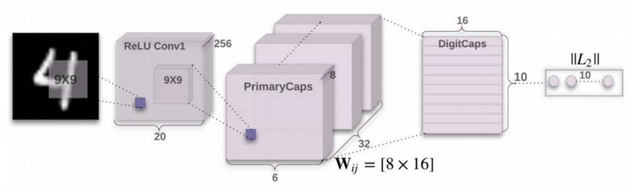

Figure 3: CapsNet architecture

We modify the final layers of the pre-existing 10-class architecture to support our 28-class multi-label classification problem. For the loss function, we use the loss proposed by Hinton et. al. [4]: the margin loss is applied to each output (in our case organelle) capsule k, and then the losses L_k are summed across capsules:

$$L_k = T_k \ \max(0, m^+ - ||\mathbf{v}_k||)^2 + \lambda \, (1 - T_k) \ \max(0, ||\mathbf{v}_k|| - m^-)^2$$

$T_k = 1$ if class k is present, $m+ = 0.9$, $m- = 0.1$, and $\lambda = 0.5$ (down-weight loss of absent classes to prevent initial learning from shrinking vector norms to 0).

4.4 DenseNet

Finally, we leverage DenseNet, a state-of-the art architecture that connects the output of a layer to all subsequent layers in a feed-forward fashion [11]. Similarly to the case with ResNet, these "skip" connections facilitate learning and combat the vanishing gradient problem within the relatively deep architecture. For this task, we use a similar setup as for ResNet: we use transfer learning (weights pretrained on ImageNet), use sigmoid for the final layer, and leverage the binary cross entropy with logits loss.

5 Experiments/Results/Discussion

For our main metric, we use the macro F1 score: this is the harmonic mean of precision and recall, averaged across the 28 classes. To obtain the F1 score on the test set, we submit to Kaggle, as the true labels are not disclosed. For all models we use the Adam optimizer.

5.1 Simple CNN

We use the architecture described in Methods, derived from the course project tutorial [12]. We resize images to 64x64 for quicker training, and train for 10 epochs over the full training set with a batch size of 32. We perform hyperparameter search over the learning rate (0.01, 0.001, 0.0001) to tune the model. On the test set, using the best learning rate of 0.001, we obtained an F1 score of **0.048**. For reference, submitting predictions of 0 for everything yields F1=0.019.

Overall, while the simple CNN exhibits slight learning, it overall has very low performance. Investigating the predictions on the test set, we find that the model predicts 0 for almost everything, and the only other labels predicted are 25 and 21; thus, the model only predicts the 3 most common labels.

5.2 ResNet

We next turn to the more sophisticated ResNet architecture. We leverage the pytorch ResNet18 (18-layer) implementation pretrained on ImageNet, and retrain only the final fully-connected layer. For this model, we rescale the images to 256x256, then take a random 224x224 crop to match the dimensions expected by ResNet.

For our base ResNet model, we use a batch size of 32, learning rate of 0.001, and train for 10 epochs. Training this for 50 epochs resulted in a test F1 score of 0.089. We then performed hyperparameter search, by searching over a smaller and larger learning rate (0.01, 0.001), and a smaller and larger batch size (16, 64), for 10 epochs. We found that increasing the learning rate and increasing the batch size resulted in a higher F1 score and lower loss.

Next, we experimented with adding positive weights to the binary cross entropy loss function. Due to the highly imbalanced nature of the classes, we hypthesized this would help performance. In particular, we used the following formulation for the positive weight of class n:

$$p_n = \frac{\text{\# neg. examples of class } n}{\text{\# pos. examples of class } n}$$

However, due to the highly imbalanced nature of the classes, the positive weights also proved to be quite imbalanced: the most common labels had an extremely small weight. To strike a middle ground between no weighting (ie, effectively $p_n = 1$) and the full weighting above, we also tried using the natural log of the full weighting ($ln(p_n)$).

We then ran two experiments, one using p_n and one using $ln(p_n)$, using the best learning rate and batch size from the earlier hyperparameter search (lr=0.01, bs=64). Ultimately, using weighting notably boosted performance, with $ln(p_n)$ slightly outperforming the full weights. Our best F1 score with this model on the test set was **0.156**. However, from qualitative analysis of the loss plot (see Figure 4), we find that the validation loss diverges, indicating that the learning rate may be too high. Further retraining with a lower learning rate, but for more epochs, may help improve performance.

It is also interesting to investigate the nature of the predictions under the different weighting schemes. With no weightings, the model predominantly predicts 1-3 labels per sample, and mostly predicts the most common label. But with the full weights, the model predicts many more labels per sample (roughly 12-15 per sample). And with ln, the model predicts roughly 1-5 labels per sample, which is closer to the distribution we would like.

5.3 CapsNet

Next, we experiment with the Capsule Network architecture, using the gram-ai pytorch implementation [13] for the 10-class SVHN dataset as a starting point. The number of capsules in the final layer (predictive unit) was expanded to 28. For this model, we rescale images to 32x32. Smaller input was used for this model than what we used for ResNet due to memory constraints (using larger images resulted in a memory error when training). We trained a base model with a learning rate of 0.001 and batch size of 32. We then performed hyperparameter search, by searching over a smaller and larger learning rate (0.01, 0.001), and a smaller and larger batch size (16, 64), for 10 epochs. We found that the largest learning rate (0.01) and smallest batch size (16) was best. We then trained a model for 16 epochs with these hyperparameters, and obtained a final test F1 score of **0.067**.

Unfortunately, CapsNet did not perform very well. This could be due to various reasons. The model appears to overfit heavily, and better hyperparameter tuning may help performance. The model also is taking in lower resolution images, which contain less information, and doesn't incorporate weights in the loss function, which also may be reducing the ability of CapsNet to compete with ResNet.

5.4 DenseNet

Finally, we turn to DenseNet, leveraging the pytorch DenseNet161 implementation pretrained on ImageNet. We used 224x224 images as input.

We tried the following experiments:

- Retrain only final classification layer, use lr=0.01, bs=64 (best hyperparameters from ResNet), and no weights in loss function.
- Retrain final dense block + classification layer, decrease lr to 0.001, and use $ln(p_n)$ weights in BCE loss function. We decreased the learning rate to help combat validation loss divergence.
- Same as above, except anneal learning rate to 0.0001 after 5 epochs, and add L2 regularization (weight decay = 0.001). These changes were made to help further combat overfitting. This performed best, with a test F1 score of **0.260**.

Finally, we briefly experimented with changing the prediction threshold: previously, a threshold of 0.5 was used for all classes (if model outputs probability > 0.5 for a class, we predict that class). However, since the classes are imbalanced, tuning the threshold could boost performance. We bucketed the classes into 4 groups based on prevalence, and used [0.5, 0.4, 0.3, 0.2] as the threshold for each bucket (with lower threshold for rarer classes). [1] Using the same 3rd model from above, but with the modified thresholds, we obtained a test F1 of **0.309**.

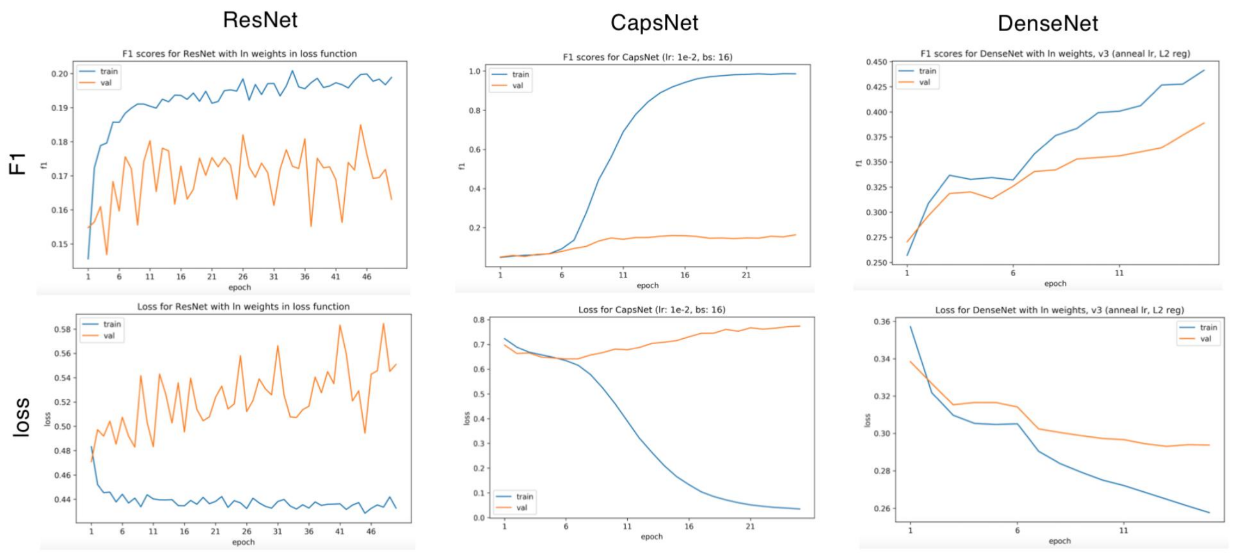

Figure 4: F1 and loss for best version of each model.

Model	Train F1	Train loss	Val F1	Val loss	Test F1
Reference: predict only 0	-	-	-	-	0.019
Simple CNN	0.061	0.146	0.052	0.142	0.048
ResNet (no weights)	0.101	0.139	0.092	0.149	0.089
ResNet (full weights)	0.169	1.544	0.153	10.477	0.155
ResNet (ln weights)	0.199	0.433	0.163	0.551	0.156
CapsNet	0.406/0.985*	0.035	0.084/0.163*	0.774	0.067
DenseNet (ln weights)	0.656/0.956*	0.022	0.299/0.395*	0.727	0.249
DenseNet (ln weights, anneal, l2)	0.355/0.441*	0.258	0.306/0.389*	0.294	**0.260/0.309**$^{+}$

Table 1: F1 score and loss results for various experiments. Note: $*$ indicates F1 score calculated over entire set, rather than by batch. $+$ indicates predictions made with custom threshold.

6 Conclusion and Future Work

In conclusion, we find that our tuned DenseNet model performed best, with a final best F1 score of 0.309, which is solid given the multi-label, multi-class nature of the task, and a good improvement from our starting baseline of 0.048. At the time of writing this, the best test F1 score in the Kaggle competition is 0.629, and the lowest non-zero entry is 0.019 (which was also my starting point as well). Thus, we achieved reasonable performance, though there is still room for improvement.

However, we found that Capsule Networks did not perform as well as hoped on this task. This could be due to a variety of reasons: the model took in smaller resolution input images, did not have pretrained weights, and also the CapsNet loss function doesn't incorporate weights (to account for class imbalance). Addressing these shortcomings, as well as doing more hyperparameter tuning, could potentially boost CapsNet performance and is an area for future work.

A few other areas for future work include: further research into the optimal prediction threshold, further research into weighting different classes differently in the model loss functions (as this helped when applied to ResNet and DenseNet), further hyperparameter tuning, and finding a way to incorporate all 4 sample filters as input.

[1]Inspiration from: `https://www.kaggle.com/c/human-protein-atlas-image-classification/discussion/71648` and conversation with my project mentor

7 Code

Code is available at: `https://github.com/LauraCruz-Albrecht/cs230_project`.

8 Acknowledgements

I would like to thank Ahmad Momeni for his mentorship and valuable suggestions over the course of this project, Pedro Garzon for introducing me to the idea of Capsule Networks, and the CS230 teaching team as a whole.

References

[1] Human protein atlas image classification. https://www.kaggle.com/c/human-protein-atlas-image-classification.

[2] “the organelle proteome”. the human protein atlas. https://www.proteinatlas.org/humancell/organelle.

[3] M. Valkinen P. Ruusuvuori K. Liimatainen, L. Latonent. Cell organelle classification with fully convolutional neural networks. *Image Analysis Challenge of Cyto*, 2017.

[4] N. Fross S. Sabour and G. Hinton. Dynamic routing between capsules. *Neural Information Processing Systems (NIPS)*, 2017.

[5] S. Sabour G. Hinton and N. Fross. Matrix capsules with em routing. *ICLR*, 2018.

[6] A. Krizhevsky G. Hinton and S. Wang. Transforming auto-encoders. *Proceedings of the 21th International Conference on Artificial Neural Networks - Volume Part I*, 2011.

[7] S. Verma and Z. Zhang. Graph capsule convolutional neural networks. *arXiv:1805.08090*, 2018.

[8] S. Bing E. Xi and Y. Jin. Capsule network performance on complex data. *arXiv:1712.03480*, 2017.

[9] Forest et. al. Iandola. Deep residual learning for image recognition. *ICLR 2017 (under review)*, 2017.

[10] Max Pechyonkin. Understanding hinton’s capsule networks. https://medium.com/ai

[11] Gao Huang, Zhuang Liu, and Kilian Q. Weinberger. Densely connected convolutional networks. *CoRR*, abs/1608.06993, 2016.

[12] cs230 project starter package. https://github.com/cs230-stanford/cs230-code-examples/tree/master/pytorch/vision.

[13] gram ai. A pytorch implementation of the nips 2017 paper "dynamic routing between capsules". https://github.com/gram-ai/capsule-networks.